SMART SDLC – AI-ENCHANCED SOFTWAREN DEVELOPMENT LIFECYCLE

PROJECT DOCUMENTATION

1. INTRODUCTION

Project Title: Smart SDLC – AI-Enchanced Software Development Lifecycle

Team Leader: Hemalatha.T

Team member: Gunavathy.D

Team member: GomathiPriya.E

Team member:Hemasree.V

2. PROJECT OVERVIEW

- **PURPOSE:**

The purpose of the SmartSDLC – AI-Enhanced software Development Lifecycle is to assist developers, students, and organizations in transforming natural language requirements into structured specifications and functional code. By leveraging large language models (LLMs) and interactive interfaces, the system automates requirement analysis, ensuring that functional, non-functional, and technical aspects are clearly identified. Additionally, it bridges the gap between software documentation and implementation by generating language-specific code directly from user prompts or uploaded documents. This dual functionality not only accelerates the software development lifecycle but also reduces ambiguity, improves accuracy, and enhances productivity. Ultimately, the assistant serves as both a requirement analyst and a coding partner, making modern AI capabilities accessible to developers of all skill levels.

- **FEATURES:**

1. Requirement Analysis

Key Point: Automatic classification of requirements

Functionality: Extracts requirements from uploaded PDF documents or manually entered text and organizes them into functional requirements, non-functional requirements, and technical specifications.

2. Code Generation

KeyPoint: Multi-language code synthesis

Functionality: Converts natural language requirements into working code across multiple programming languages, enabling faster prototyping and reducing manual development effort.

3. PDF Text Extraction

Key Point: Document-driven analysis

Functionality: Reads PDF files and extracts their textual content for automated requirement analysis, ensuring compatibility with existing project documentation.

4. Conversational AI Assistance

Key Point: Natural interaction with developers

Functionality: Uses IBM Granite LLM models to generate meaningful and context-aware responses, providing insights, explanations, and structured outputs.

5. Interactive Web Interface

Key Point: User-friendly Gradio dashboard

Functionality: Offers a tab-based interface with clear workflows for both requirement analysis and code generation, accessible to both technical and non-technical users.

6. Cross-Platform Compatibility

Key Point: Flexible deployment

Functionality: Runs on CPU or GPU environments, leveraging PyTorch optimizations to ensure scalability for both personal use and enterprise adoption

7. Error Handling

Key Point: Reliable system behavior

Functionality: Detects and manages issues such as empty inputs, unreadable PDFs, or invalid prompts, ensuring that the application provides clear error messages instead of breaking.

8. Dual Input Support

Key Point: Flexible requirement entry

Functionality: Accepts both PDF uploads and manually typed text, allowing users to analyze requirements from existing documentation or freeform descriptions.

9. Gradio UI

Key Point: User-friendly interface

Functionality: Provides an intuitive dashboard with tabbed navigation for requirement analysis and code generation, making it easy for both technical and non-technical users to interact with the system.

3. ARCHITECTURE

Frontend (Gradio):

The frontend is built with Gradio, providing an interactive and user-friendly web interface. It includes multiple tabs such as requirement analysis and code generation, allowing users to switch seamlessly between tasks. The design supports both PDF uploads and text input, ensuring flexibility in how requirements are provided. The interface also organizes outputs into clearly structured text boxes, helping users quickly understand the results.

Backend (Python with Hugging Face Transformers):

The backend logic is powered by Python, leveraging the Hugging Face Transformers library and PyTorch for model execution. It handles PDF parsing, prompt engineering, and interaction with the large language model. The backend also manages request processing for both requirement analysis and code generation, ensuring smooth communication between the interface and the AI engine.

LLM Integration (IBM Granite):

The project integrates IBM Granite LLM models for natural language understanding and generation. These models are used to analyze requirement documents, classify them into functional and non-functional categories, and generate code snippets in multiple programming languages. Prompts are carefully structured to maximize output accuracy and maintain context awareness.

Document Processing (PyPDF2):

The system uses PyPDF2 to handle PDF inputs. Uploaded documents are parsed page by page, and their textual content is extracted for requirement analysis. This enables users to upload existing project documents and receive structured insights without manually copying content.

Code Generation Engine:

A dedicated module constructs prompts for code generation based on user-selected programming languages. The model then generates executable code, which is displayed in the interface. This feature supports multiple languages, making the tool versatile across different development environments.

Error Handling and Validation:

To ensure reliability, the architecture incorporates mechanisms to handle malformed PDFs, empty inputs, and excessively large prompts. Instead of system crashes, users receive informative error messages, allowing them to correct issues and continue working smoothly.

4. SETUP INSTRUCTIONS

- **PRE REQUISITES:**
    - Python 3.9 or later
    - pip and virtual environment tools
    - PyTorch installed with CPU or GPU support

- Hugging Face Transformers library
- Gradio for web interface
- PyPDF2 for PDF document processing
- Internet access for downloading models from Hugging Face

INSTALLATION PROCESS:

- Clone the repository containing the project files
- Install dependencies using requirements.txt or pip install -r requirements.txt
- Ensure PyTorch is properly configured (with CUDA support if using GPU)
- Launch the application by running the main script (e.g., python app.py)
- Access the Gradio interface through the provided local or shareable URL
- Upload requirement documents (PDFs) or enter text and interact with the modules for requirement analysis and code generation

5. FOLDER STRUCTURE

app/ – Contains all backend AI logic. Breaking into separate files like model_loader.py, pdf_processor.py, requirement_analysis.py, and code_generation.py ensures modularity. Each file handles one specific functionality, making the project easier to maintain.

ui/ – Contains frontend components. app_interface.py hosts the Gradio UI with tabs for Code Analysis and Code Generation. This separates presentation logic from backend AI logic.

requirements.txt – Lists all dependencies needed to run the project (transformers, torch, gradio, PyPDF2, etc.).

run.py – Entry point to launch the Gradio app. It imports the interface from ui/app_interface.py and runs app.launch().

README.md – Provides project documentation, setup instructions, and usage details.

6. RUNNING THE APPLICATION

To start the project:

- Run the Python script containing the Gradio interface to launch the web application.
- Open the local URL provided by Gradio in a browser to access the interactive UI.
- Navigate between the Code Analysis and Code Generation tabs.
- Upload PDF documents or enter requirement descriptions, select programming language (for code generation), and view outputs such as organized requirements or generated code.
- All interactions are real-time and rely on the backend AI model to dynamically process inputs and update the frontend **interface.**

Frontend (Gradio):

The frontend is built with Gradio, providing a clean and interactive web interface. It consists of multiple components including file upload widgets, text input boxes, dropdown menus for language selection, buttons to trigger analysis or code generation, and output text boxes to display results. The interface is tab-based, with separate tabs for Requirement Analysis and Code Generation, making it easy for users to switch between functionalities. Each component is modularized for scalability and future enhancements.

Backend (AI Model & Processing):

The backend uses the IBM Granite AI model, loaded through the Transformers and PyTorch libraries. It handles PDF text extraction, requirement analysis, and code generation. PDF files are processed using PyPDF2 to extract text, which is then sent to the Granite AI model for semantic analysis. For code generation, the model takes user-written requirements and produces code in the selected programming language. The system is optimized to use GPU if available for faster inference and ensures all requests are processed efficiently in real-time.

7. Functionality Documentation

The AI Code Analysis & Generator application provides two main functionalities: Requirement Analysis and Code Generation. These are accessible through the Gradio frontend, where each button click triggers the corresponding function in the backend.

Main Functional Components:

1. Requirement Analysis (requirement_analysis)

Input: A PDF document uploaded by the user or a textual description of software requirements.

Process: If a PDF is uploaded, the text is extracted using PyPDF2. The extracted text or the typed input is then sent to the IBM Granite AI model, which organizes it into functional requirements, non-functional requirements, and technical specifications.

Output: A structured requirement analysis displayed in the Gradio textbox.

2. Code Generation (code_generation)

Input: A textual description of the software functionality and a programming language selected from a dropdown.

Process: The input prompt is formatted and sent to the Granite AI model, which generates executable code in the chosen language.

Output: Generated code displayed in the Gradio textbox.

3. PDF Text Extraction (extract_text_from_pdf)

Input: PDF file uploaded by the user.

Process: Reads each page of the PDF and extracts text content.

Output: Raw text string, used internally for requirement analysis.

How the Functions Work Together:

- When the user clicks Analyze, the requirement_analysis function is called. It internally uses extract_text_from_pdf if a PDF is provided.
- When the user clicks Generate Code, the code_generation function is called with the user's prompt and selected language.

- The Gradio interface handles the input and output display, giving the user an interactive experience.

8. User Access & Interaction

The AI Code Analysis & Generator provides an intuitive and fully accessible interface for users to interact with AI-powered tools without any login or authentication. The application is designed for immediate use, making it ideal for demonstrations, testing, and learning purposes.

Core Capabilities Accessible to Users:

Requirement Analysis: Upload PDFs or enter textual software requirements to automatically classify them into functional, non-functional, and technical specifications.

Code Generation: Enter requirement descriptions and select a programming language to generate executable code instantly.

Real-Time Feedback: All inputs are processed in real-time, providing immediate results in the interface.

Key Notes:

The application is open-access, so all features are available without credentials.

No user data, session information, or generated outputs are stored, ensuring privacy and simplicity.

9. USER INTERFACE

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

- Tabbed layouts for Requirement Analysis and Code Generation
- File upload for PDFs and text input boxes for requirements
- Dropdown menu to select programming language for code generation
- Buttons to trigger analysis or code generation

- Output text boxes to display results in real-time

The design prioritizes clarity, speed, and user guidance, with clear labels, placeholder texts, and intuitive flows for a smooth user experience.

10. TESTING

Testing was carried out in multiple phases:

Unit Testing: For core functions such as requirement_analysis, code_generation, and extract_text_from_pdf

Functional Testing: Ensuring PDF uploads, text inputs, and dropdown selections work as expected

Manual Testing: Validating real-time outputs for requirement analysis and generated code

Edge Case Handling: Tested with empty inputs, large PDF files, and unusual requirement descriptions

11. SCREEN SHOTS

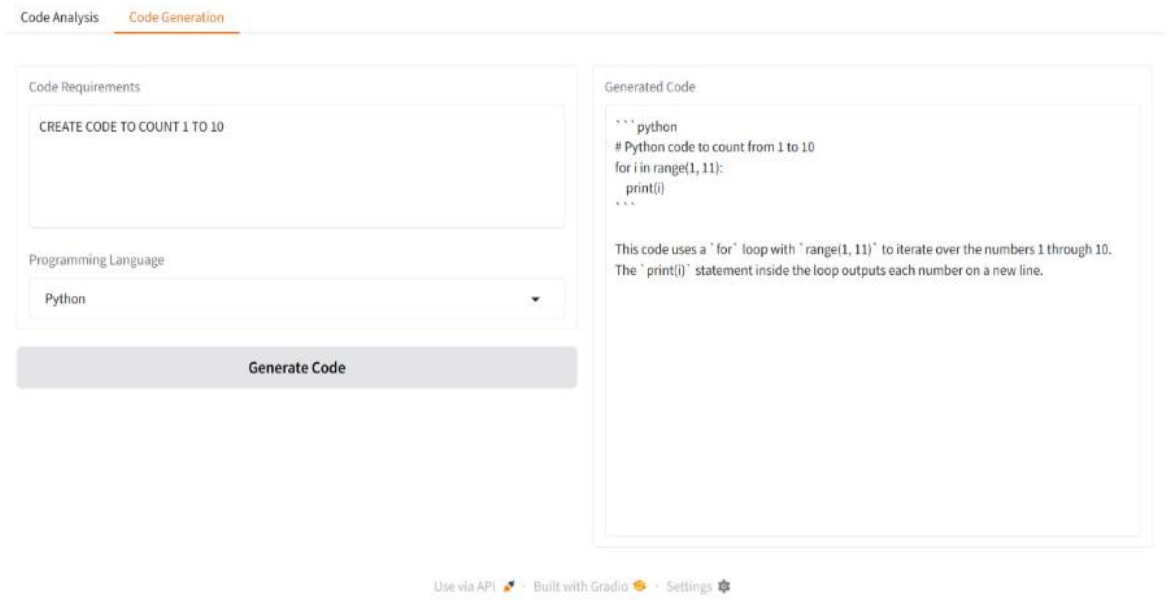
Input Options

1. **PDF Upload** – Upload a requirements document in PDF format.
2. **Text Box** – Type requirements manually.
3. **Programming Language Dropdown** – Choose language for code generation.

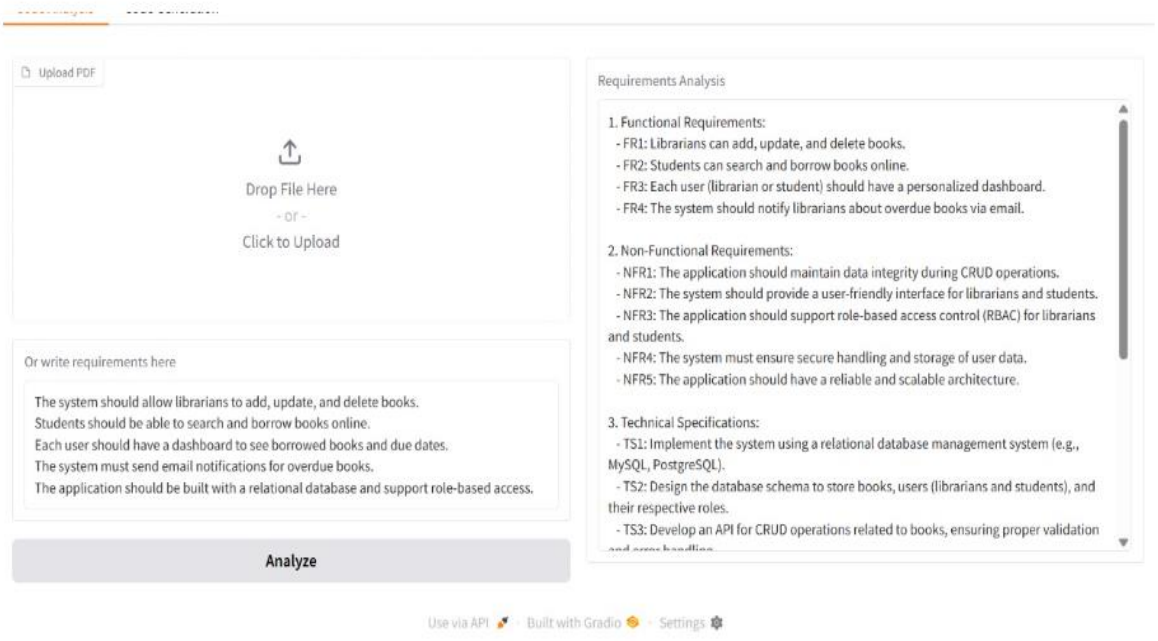
Output

1**. Requirement Analysis Tab** – Displays requirements categorized into functional, non-functional, and technical.

2. **Code Generation Tab** – Displays AI-generated code in the chosen language.



CODE GENERATOR



CODE ANALYSIS

12. KNOWN ISSUES

Large PDF Handling: Very large PDF files may cause slower processing or memory issues.

No Persistent Storage: Uploaded documents, inputs, and generated outputs are not saved; closing the app clears all data.

Limited Error Handling: Malformed PDFs or unexpected input formats may cause errors or incomplete text extraction.

AI Model Limitations: Granite AI may sometimes generate code that requires manual debugging or refinement.

13. FUTURE ENHANCEMENTS

Secure Access: Integration of authentication mechanisms such as API keys or OAuth2 for production deployment.

Role-Based Access Control: Restrict access to specific features based on user roles (e.g., admin, developer, viewer).

Session Management: Track and store user sessions, inputs, and generated outputs for auditing or analytics purposes.