SmartSDLC-AI-Enhanced Software Development Lifecycle

|  |  |
| --- | --- |
| Date | 10 September 2025 |
| Team ID | NM2025TMID00608 |
| **Project Name** | SmartSDLC-AI –Enhanced Software Development Lifecycle |
| **Maximum Marks** |  |

# 1. Introduction

Transforming software requirements into structured specifications and code is a critical yet time-consuming task in modern development. Traditionally done manually, it often leads to inefficiencies and errors.

This project presents an **AI-powered tool** that leverages **Natural Language Processing (NLP)** and **code generation** to automate this process. Users can upload a PDF or enter text to receive:

* A structured breakdown of **functional**, **non-functional**, and **technical requirements**
* **Auto-generated code** in languages like Python, Java, and C++

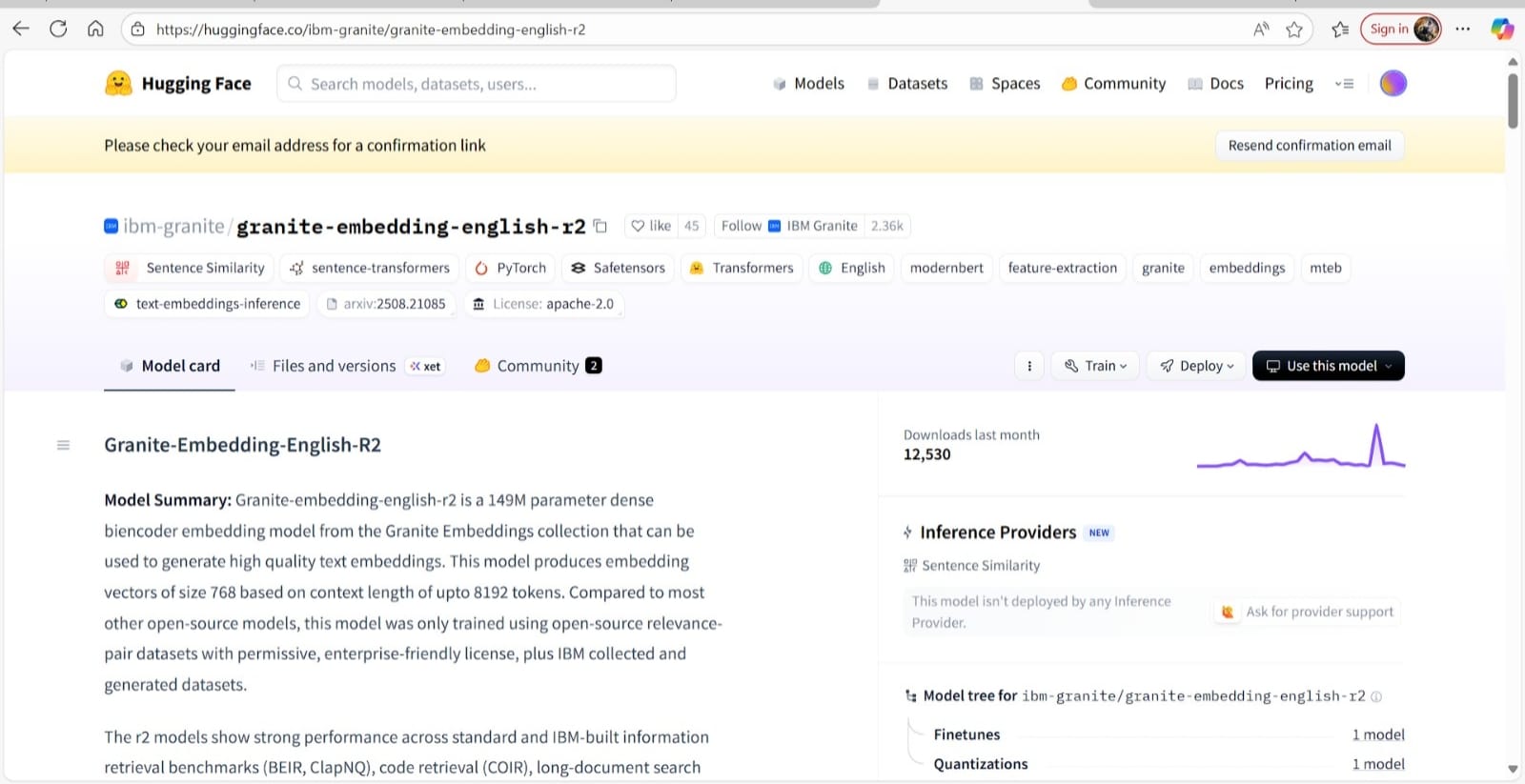
The system uses IBM’s granite-3.2-2b-instruct model via Hugging Face, integrates PyTorch for inference, features a Gradio-based UI, and is designed for future backend expansion with FastAPI.

# 1. Project Overview

This project focuses on developing a web-based AI tool that enables users to analyze software requirements (either from text or PDF documents) and generate corresponding source code in a variety of programming languages. The application integrates a powerful transformer-based generative model (ibm-granite/granite-3.2-2b-instruct) for natural language processing and code generation.

The tool is built using:

* **Transformers (Hugging Face)**
* **PyTorch**
* **Gradio** for UI
* **PyPDF2** for PDF processing
* **FastAPI** for scalable backend support (in progress)



# Key Features

* **AI Requirement Analysis**: Automatically categorizes requirements into **functional**, **non-functional**, and **technical**.
* **PDF & Text Input**: Analyze uploaded PDFs or manually entered text.
* **Multi-language Code Generation**: Generates code in **Python, Java, JS, C++, C#, PHP, Go, Rust**.
* **Simple UI**: Two-tab interface for **Analysis** and **Code Generation** using Gradio.
* **Fast Inference**: Uses **IBM Granite AI model** with GPU support.
* **Web-Deployable**: Easily launchable via browser for local or cloud use.
* **Modular Design**: Clean separation of UI, backend, and model logic.
* **Future-Ready**: Export to DOCX/PDF, login system, and history tracking coming soon.

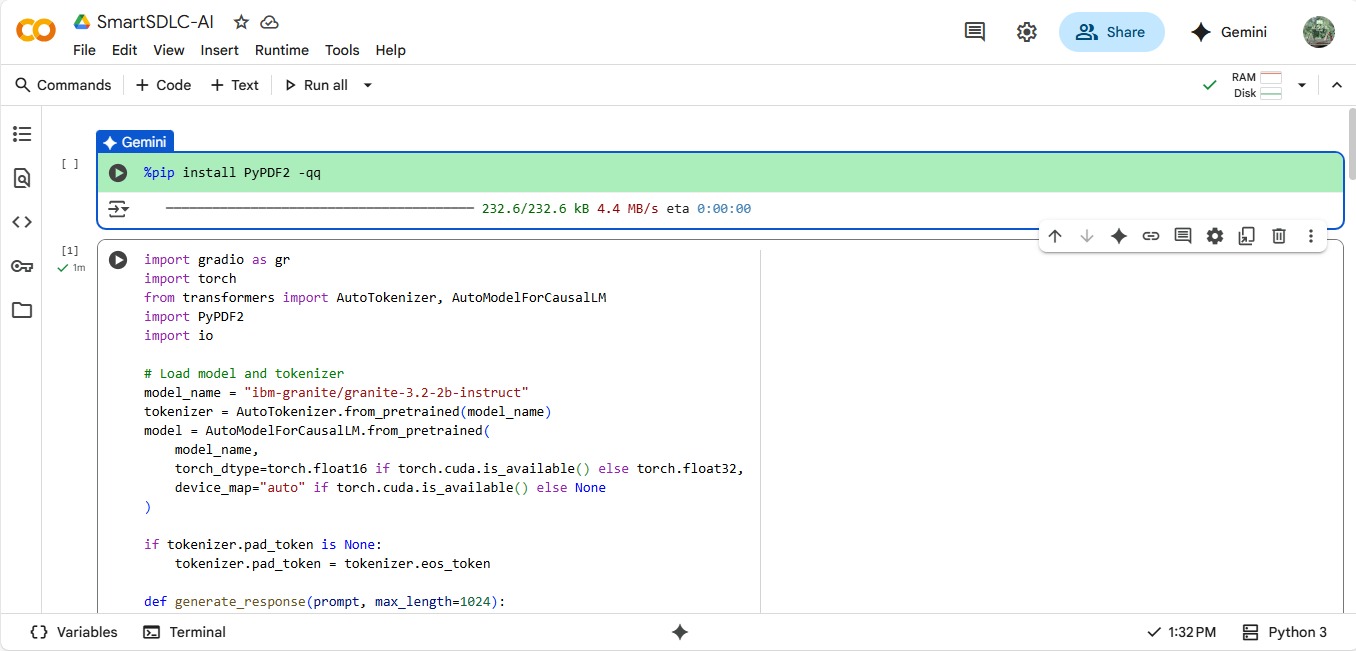
# 2. Milestone 1: Model Selection and Architecture

**Activity 1.1: Research and Select the Appropriate Generative AI Model**

* Researched various open-source transformer models capable of both text understanding and code generation.
* Compared models like **CodeGen**, **CodeLlama**, and **Granite** based on size, inference cost, instruction-following capability, and license.
* Selected **ibm-granite/granite-3.2-2b-instruct** due to:
  + Instruction-tuned performance
  + Lightweight architecture (2B parameters) for efficient inference
  + Good balance between generation quality and resource usage
* Integrated with **Hugging Face Transformers** using AutoTokenizer and AutoModelForCausalLM.

**Architecture Highlights:**

* Transformer-based model
* Torch-based inference pipeline
* Dynamic allocation to CUDA or CPU based on availability
* PDF parsing module and code/text prompt handling logic



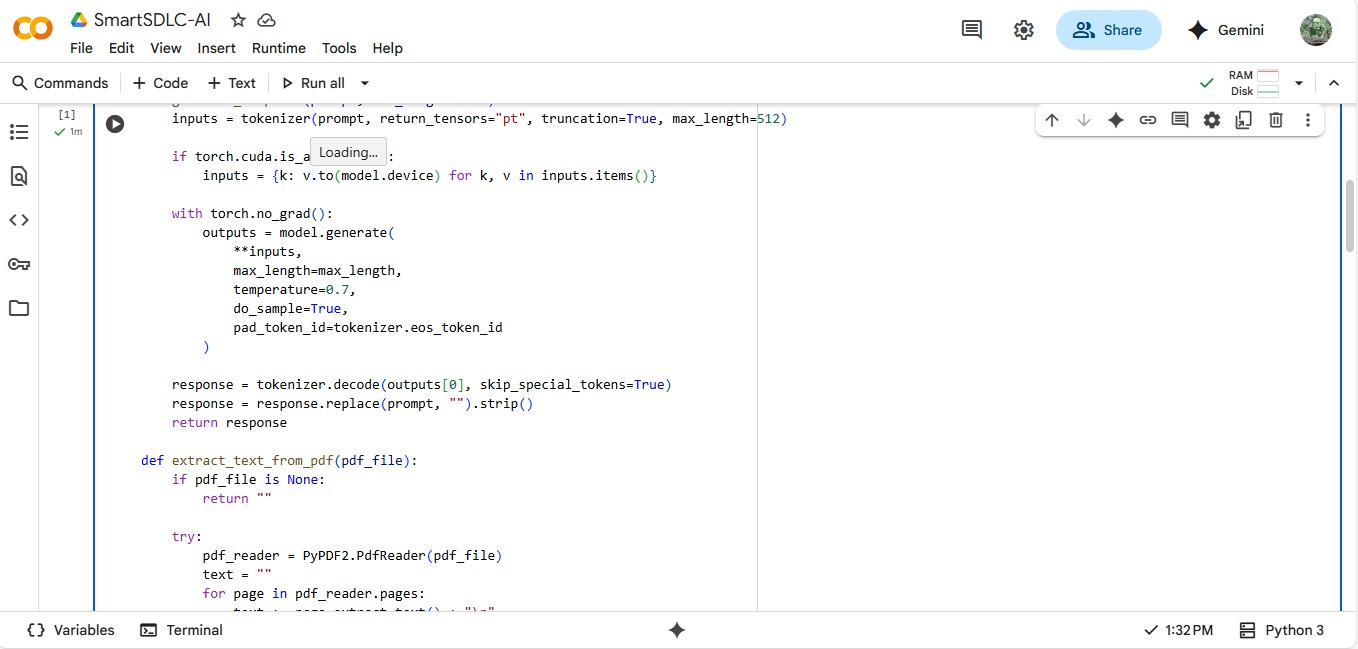
# 3. Milestone 2: Core Functionalities Development

**Activity 2.1: Develop the Core Functionalities**

Core Features Implemented:

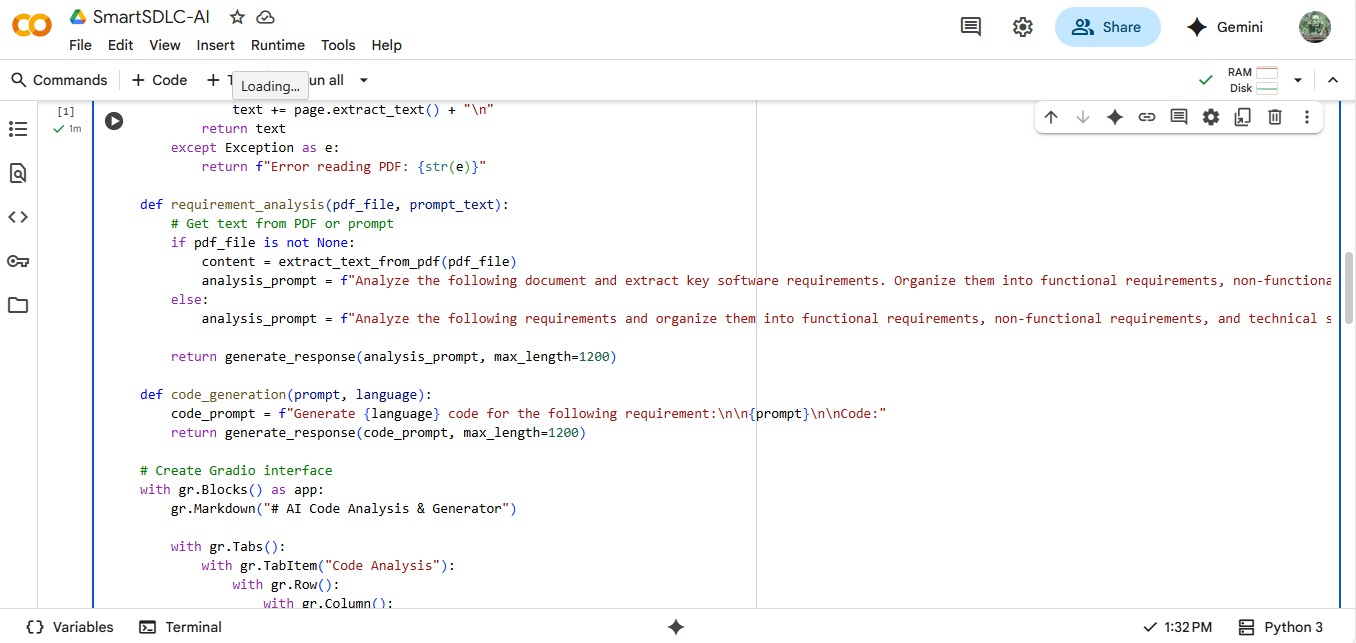
* **Requirement Analysis**
  + Users can upload a PDF or enter text describing software requirements.
  + The system extracts functional, non-functional, and technical specifications.
* **Code Generation**
  + Generates executable source code in a selected language from natural language input.
  + Supports multiple languages (Python, Java, C++, etc.).

Key Backend Logic:

* PDF parsing using PyPDF2
* Prompt formulation and response parsing
* Dynamic max-length and sampling settings for model generation
* 

**Activity 2.2: Implement the FastAPI Backend**

* A FastAPI backend will be created to manage all model inference and PDF processing routes.
* This decouples the frontend (Gradio) from the model, improving scalability.
* Key Endpoints:
  + /analyze: Accepts text or PDF for requirement analysis
  + /generate-code: Accepts prompt and language for code generation
* Includes asynchronous I/O handling and CORS management



# 4. Milestone 3: Main Application Logic

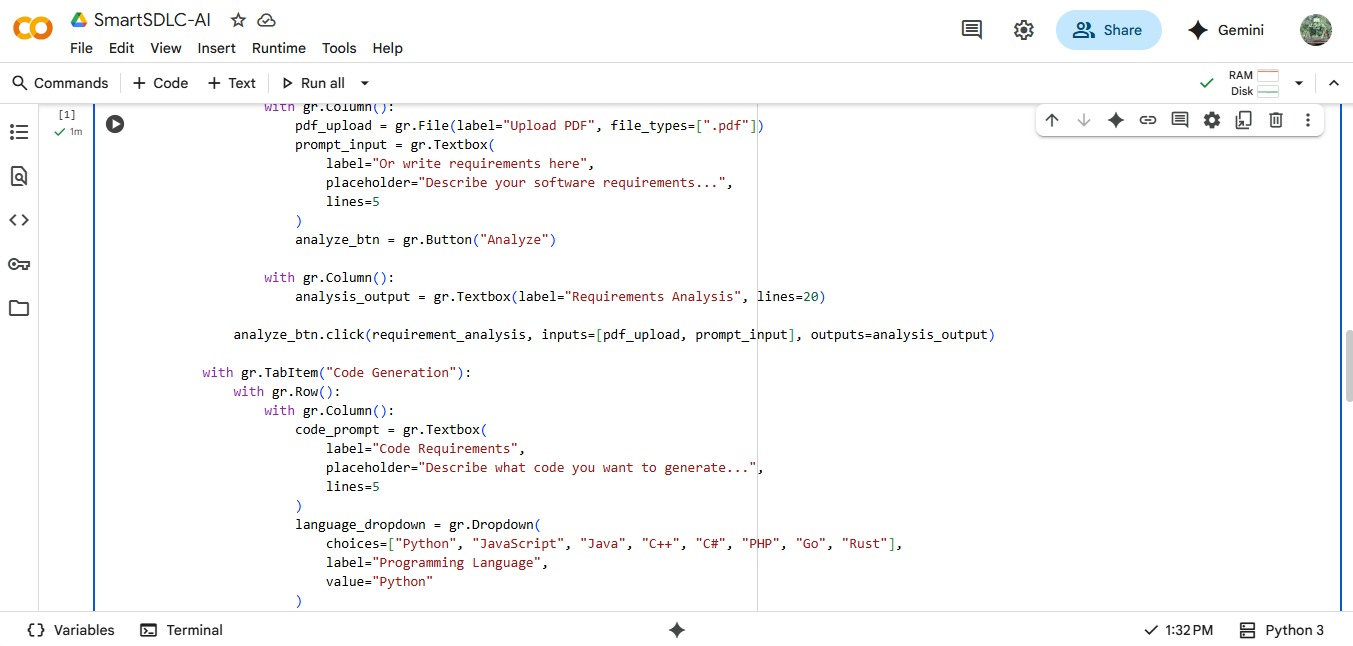
**Activity 3.1: Writing the Main Application Logic in main.py**

The main application script handles:

* Model loading and tokenizer setup
* Conditional GPU/CPU support
* Prompt engineering for analysis and code generation
* PDF file reading and content extraction
* Response formatting and UI output rendering

Highlights:

* Efficient use of torch.no\_grad() for inference
* Automatic device mapping
* Pad token fallback logic
* Sampled generation (temperature=0.7, do\_sample=True) for creative output



# 5. Milestone 4: Frontend Development

**Activity 4.1: Designing and Developing the User Interface**

* Built using **Gradio Blocks** for modularity and UI control
* Tabbed UI for clear separation of tasks:
  + **Code Analysis Tab**
    - Upload PDF or type prompt
    - Click to analyze and extract structured requirements
  + **Code Generation Tab**
    - Input text + select language
    - Click to generate corresponding source code

**Activity 4.2: Creating Dynamic Interaction with Backend**

* Gradio UI buttons (click) are linked to backend functions.
* Inputs and outputs are seamlessly connected to respective functions:
  + PDF file or prompt → analysis result
  + Code request + language → generated code

Planned:

* FastAPI-Gradio integration for deployment
* Replacing direct inference in UI with API calls



# 6. Milestone 5: Deployment

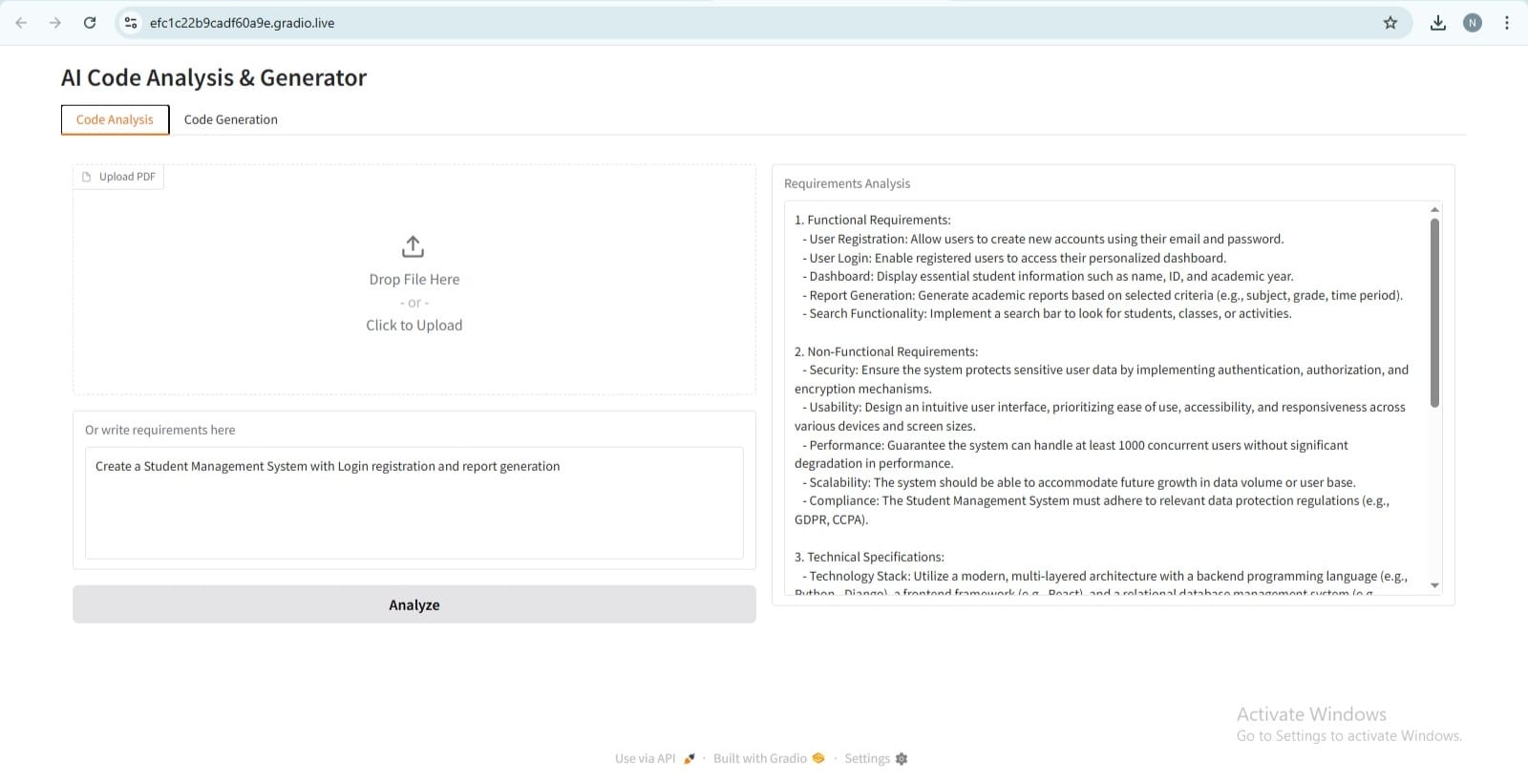
**Activity 5.1: Preparing the Application for Local Deployment**

* Dependencies are managed via requirements.txt or pip
* Gradio launches locally on the user’s machine using app.launch(share=True)
* GPU support included when available Planned Improvements:
* Dockerization for environment consistency
* Configurable .env files



**Activity 5.2: Testing and Verifying Local Deployment**

* Functional tests done for:
  + Model loading
  + Prompt analysis and decoding
  + PDF extraction accuracy
  + UI responsiveness
* Unit testing for modular functions
* Load testing with larger PDFs



# 7. Milestone 6: Conclusion

The project successfully demonstrates the capability of using an instruction-tuned generative model to:

* Extract structured software requirements from unstructured inputs
* Generate usable code across multiple languages
* Offer an interactive UI for both technical and non-technical users

Future Work:

* Extend to web deployment (e.g., using FastAPI + Docker)
* Add authentication for usage control
* Implement formatting options (Markdown, Table, JSON output)
* Improve PDF parsing using OCR fallback or pdfplumber