SMARTSDLC - AI - ENHANCED SOFTWARE DEVELOPMENT LIFECYCLE

1.INTRODUCTION

- Team Leader : P.N RAMALAKHSHMI
- Team Member : M THAMIZHARASI
- Team Member : V REKSHANAA
- Team Member : E HARISHMA
- Team Member : C MANIMEGALAI

2.PROJECT OVERVIEW

2.1 PURPOSE :

The purpose of SMARTSDLC – AI-Enhanced Software Development Lifecycle is to integrate Artificial Intelligence into the traditional software development process, making it more efficient, automated, and intelligent.

By leveraging Natural Language Processing (NLP) and advanced language models, the system assists developers, analysts, and project managers throughout the early phases of SDLC. It automatically extracts and organizes requirements from documents or user input, and generates optimized, ready-to-use code in multiple programming languages.

Ultimately, this assistant reduces manual effort, bridges communication gaps between stakeholders and developers, accelerates development cycles, and ensures higher productivity with fewer errors.

It serves as both a requirement analysis partner and a code generation assistant, making the SDLC smarter, faster, and more reliable.

2.2 FEATURES

1.Requirements Analysis

- Key Point: Automated requirement extraction
- Functionality: Analyzes uploaded PDFs or written text to categorize requirements into functional, non-functional, and technical specifications.

2.Code Generation

- Key Point: Multi-language code synthesis
- Functionality: Generates optimized code in languages like Python, Java, C++, C#, and others based on plain-text software requirements.

3.Conversational Interface

- Key Point: Natural language interaction
- Functionality: Allows developers and stakeholders to input requirements in plain English and receive structured outputs.

4.Multimodal Input Support

- Key Point: Flexible requirement handling
- Functionality: Accepts both textual inputs and uploaded PDF documents for requirement analysis.

5.AI-driven Insights

- Key Point: Smart development guidance
- Functionality: Provides AI-enhanced suggestions, ensuring requirements are well-structured before moving into the design or coding phase.

6.User-Friendly Gradio UI

- Key Point: Simplified developer experience
- Functionality: Offers an interactive dashboard with tabs for requirement analysis and code generation, making the process intuitive.

3. ARCHITECTURE

1.Frontend (Gradio UI):

- The frontend is built with Gradio, offering an interactive and user-friendly web interface. It provides multiple tabs including
- Requirement Analysis Tab – for uploading PDFs or entering text-based requirements.
- Code Generation Tab – for selecting programming languages and generating code.

- The interface is lightweight, intuitive, and modular, making it easy for developers and non-technical stakeholders to interact with the system.

2.Backend (Python + Hugging Face Transformers):

The backend is powered by Python and Hugging Face Transformers, enabling seamless integration with large language models. Core responsibilities include:

- Requirement text extraction and preprocessing.
- AI-driven requirement classification (functional, non-functional, technical). Code generation in multiple programming languages.
- Device management for CPU/GPU execution using PyTorch.

3.LLM Integration (IBM Granite Model):

The system integrates the IBM Granite 3.2 instruct model for natural language understanding and generation. Prompts are designed to handle:

- Requirement analysis from natural text.
- Automatic code generation.
- Structured output formatting for improved readability.

4.Document Processing (PyPDF2):

- Uploaded PDF requirement documents are parsed using PyPDF2, and the extracted text is sent to the AI model for analysis.

5.Key Modules:

- Requirement Analysis Module – Organizes requirements into categories.
- Code Generation Module – Produces executable code in the chosen language.
- Gradio Interface Manager – Handles interaction between user inputs, AI processing, and output display.

4.SETUP INSTRUCTIONS

4.1 Prerequisites

- **Python 3.9 or later**
- **pip** (Python package installer)
- **Internet access** (required to download IBM Granite model from Hugging Face)

Required Python libraries:

- `torch` (PyTorch for model execution)
- `transformers` (Hugging Face tokenizer and model)
- `gradio` (for building the web interface)
- `PyPDF2` (for extracting text from PDFs)

4.2 Installation Process

1.Download the Project

- Place the `main.py` file in a folder of your choice.

2. Install Dependencies

- Open a terminal in the project folder and run:

```
pip install torch transformers gradio PyPDF2
```

3. Run the Application

Start the app by running :

```
python main.py
```

Gradio will provide a local URL (e.g., `http://127.0.0.1:7860`).

4.Access the Interface

- Open the URL in your browser.
- Use the **Code Analysis tab** to upload a PDF or type requirements manually.
- Use the **Code Generation tab** to enter prompts and generate code in your selected programming language.

5. FOLDER STRUCTURE

Since this version of the project is **single-file**, the folder structure is very simple:

```
project_folder/
|
├ main.py              # Contains all code (UI, requirement analysis, code generation)
├ sample_pdfs/         # Optional: place PDF files here for testing
└ requirements.txt     # Optional: list dependencies (torch, transformers, gradio, PyPDF2)
```

Key points:

- All functionality is implemented in main.py.
- You can add a folder like sample_pdfs/ to organize test PDF files.
- If desired, a requirements.txt file can be added for easy installation of dependencies.

6.RUNNING THE APPLICATION

- This step is about executing the project after all files are arranged and installed.
- It ensures the application loads the AI model, starts the Gradio interface, and is ready for use.
- The project is run from the main.py file or the main script you created.
- Before running, you must install all required libraries mentioned in requirements.txt.
- The application can be run locally on your system or shared via a link (Gradio share).

- After running, the app will open a browser window where you can access:
  - Code Analysis Tab → Upload PDF or type requirements to get analysis.
  - Code Generation Tab → Type requirements and get generated code.

Steps to run the Application

- Open the project folder on your computer.
- Open a terminal / command prompt in that folder.

Install the required packages by running:

```
pip install -r requirements.txt
```

- Run the main script by typing:
  python main.py
- Wait for the Gradio interface to launch — it will give you a local URL (like http://127.0.0.1:7860).
- Click the URL to open the web interface in your browser.
- Use the interface to perform requirement analysis or generate code.
- (Optional) Enable share=True in Gradio to get a public link you can share with others.

7. API DOCUMENTATION

- API Documentation explains how the functions, classes, and modules work in your project.
- It helps developers understand how to use and extend your code.
- Provides clear details about input parameters, output values, and usage examples.
- Makes maintenance, debugging, and integration easier for other developers.
- Can be written manually or generated automatically using tools like Sphinx or pdoc.
- Should be kept updated whenever code changes, so it always reflects the latest version.

- Identify all major modules and functions in your project (like analysis.py, code_generation.py, utils.py).

- Write docstrings inside your Python files — describe what each function does, its parameters, and return values.

- Add usage examples in the docstrings to show how to call the function.

- Create a separate docs/ folder in your project to store documentation files.

- Use an auto-documentation tool (optional) like Sphinx, MkDocs, or pdoc to generate HTML documentation from your docstrings.

- Link your API documentation in the README.md file so users know where to find it.

- Review and update the documentation whenever you change or add features to the project.

8. AUTHENTICATION

Authentication is not included in the current version of this application.

- The application runs locally with Gradio and does not require login.
- Anyone with the local or shared Gradio link can access and use the tool.
- This makes it simple for testing and demonstrations, but not secure for production environments.

If needed in the future, authentication can be added using:

- Gradio's built-in authentication (`app.launch(auth=[("username", "password")])`)
- API keys or tokens for controlled access

- OAuth / Single Sign-On (SSO) for enterprise integration

At present, the tool is open access, intended for local use and experimentation only.

9.User Interface

The project uses **Gradio** to build an interactive and user-friendly interface that connects users with the AI model.
The interface is designed with a **tab-based layout**, making it simple and organized for different tasks:

1. Home Display:

- A title section (AI Code Analysis & Generator) welcomes the user.
- The interface is built using gr.Blocks(), which allows a clean and modular layout.

2. Code Analysis Tab:

Inputs:

- File upload option (.pdf) for requirement documents.
- A text box to manually enter requirements.
- A button to start the analysis process.

Output:

- A large text box displaying extracted and categorized requirements (functional, non-functional, technical).

3. Code Generation Tab:

Inputs:

- A text box to describe code requirements.
- A dropdown menu to select the target programming language (Python, Java, C++, etc.).
- A button to generate code.

Output:

- A text box that displays the generated code clearly for the user.

4. Interactive Features:

- Buttons (Analyze, Generate Code) trigger AI functions.
- Real-time interaction: once the user clicks, the AI responds instantly with results.
- The interface is simple, intuitive, and requires no technical expertise.

5. Accessibility:

- Web-based interface (app.launch(share=True)), allowing users to access it from anywhere without extra installation.
- Clear labeling, placeholders, and multi-line text boxes enhance usability.

10.TESTING

The testing phase ensures that the AI-powered **Code Analysis & Generator** system works reliably and produces accurate results. The following are the manual testing approaches were applied to validate the project:

Testing in this version is performed **manually**, as no automated test scripts are included.

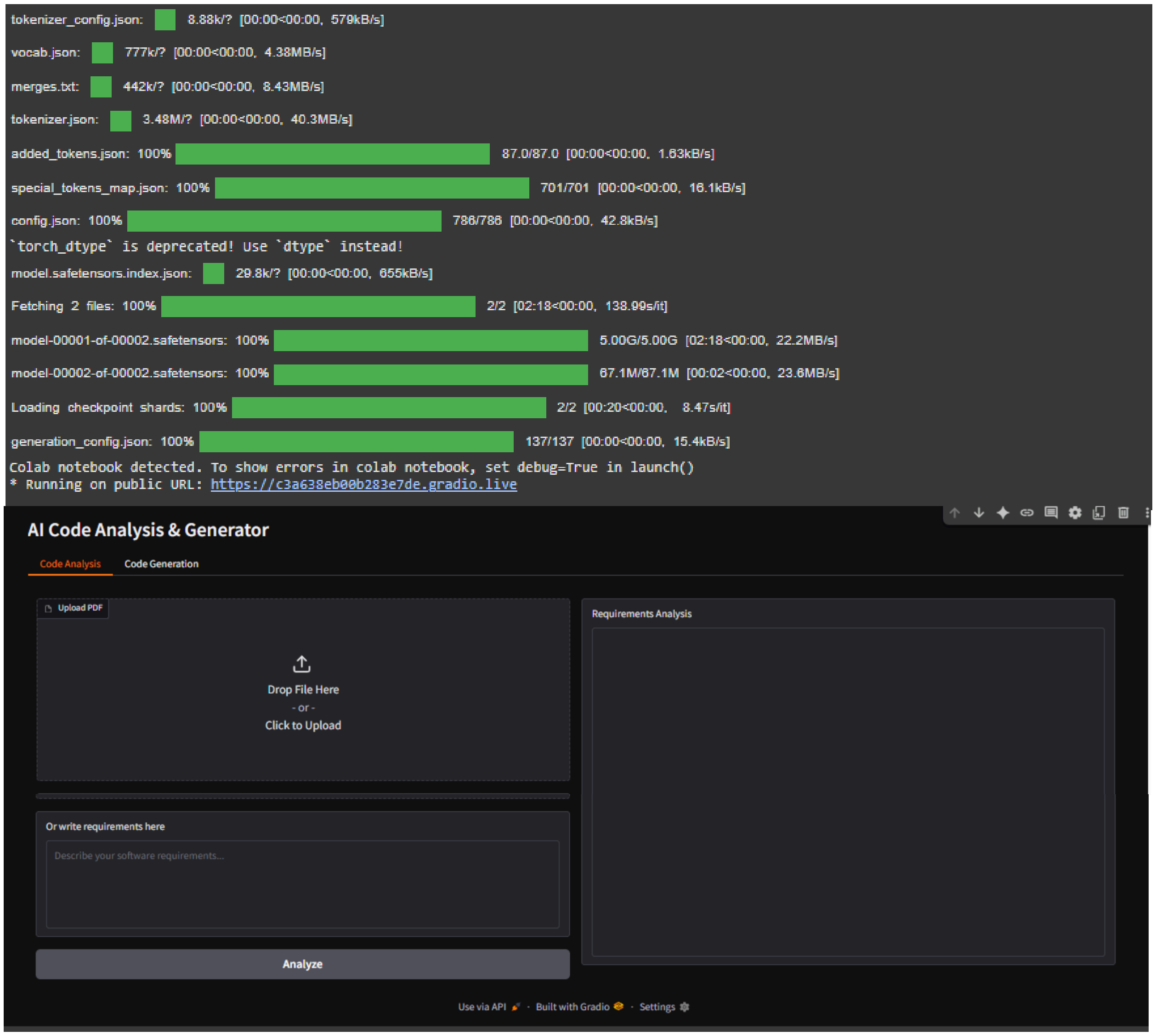
1. **Requirement Analysis Testing**
    - Upload a sample PDF with requirements or enter text manually.
    - Verify that the output categorizes requirements into **functional**, **non-functional**, and **technical**.
    - Check that empty or corrupted PDFs return a clear error message.
2. **Code Generation Testing**
    - Enter a prompt describing desired functionality (e.g., “Build a calculator that adds two numbers”).
    - Select a programming language from the dropdown.
    - Verify that valid code is generated and displayed in the output box.
3. **User Interface Testing**
    - Ensure all UI components (file upload, textboxes, buttons, dropdowns) work as intended.
    - Confirm outputs appear in the correct sections immediately after interaction.
4. **Error Handling Testing**
    - Test with incomplete or ambiguous prompts to see if the AI returns meaningful responses.
    - Verify that PDF reading errors are handled gracefully.

5. **Performance Observation**

   ○ Observe model response times on CPU-only and GPU-enabled systems.

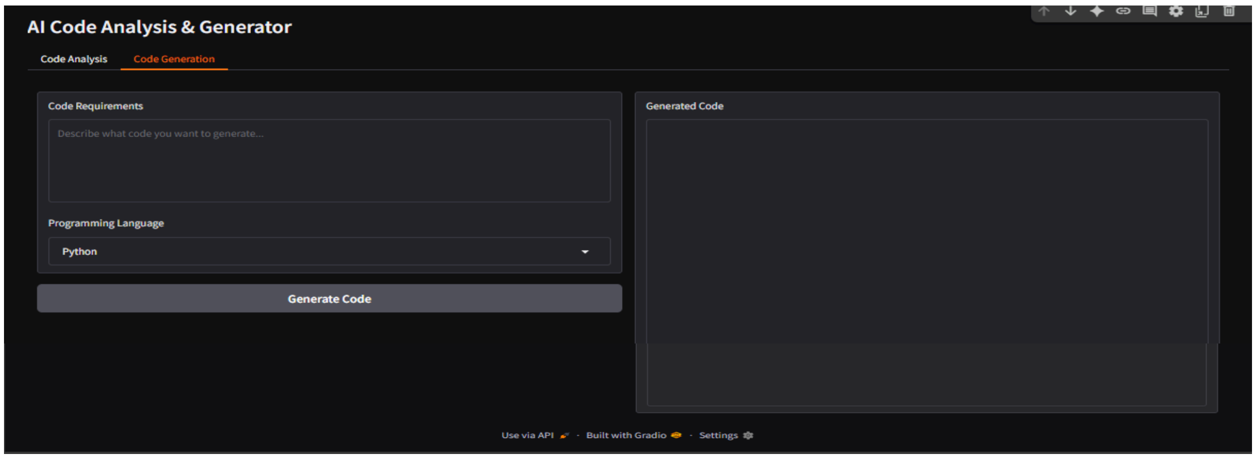
   ○ Ensure outputs are generated without crashing or freezing, even for longer prompts

11.SCREENSHOTS OF OUTPUT

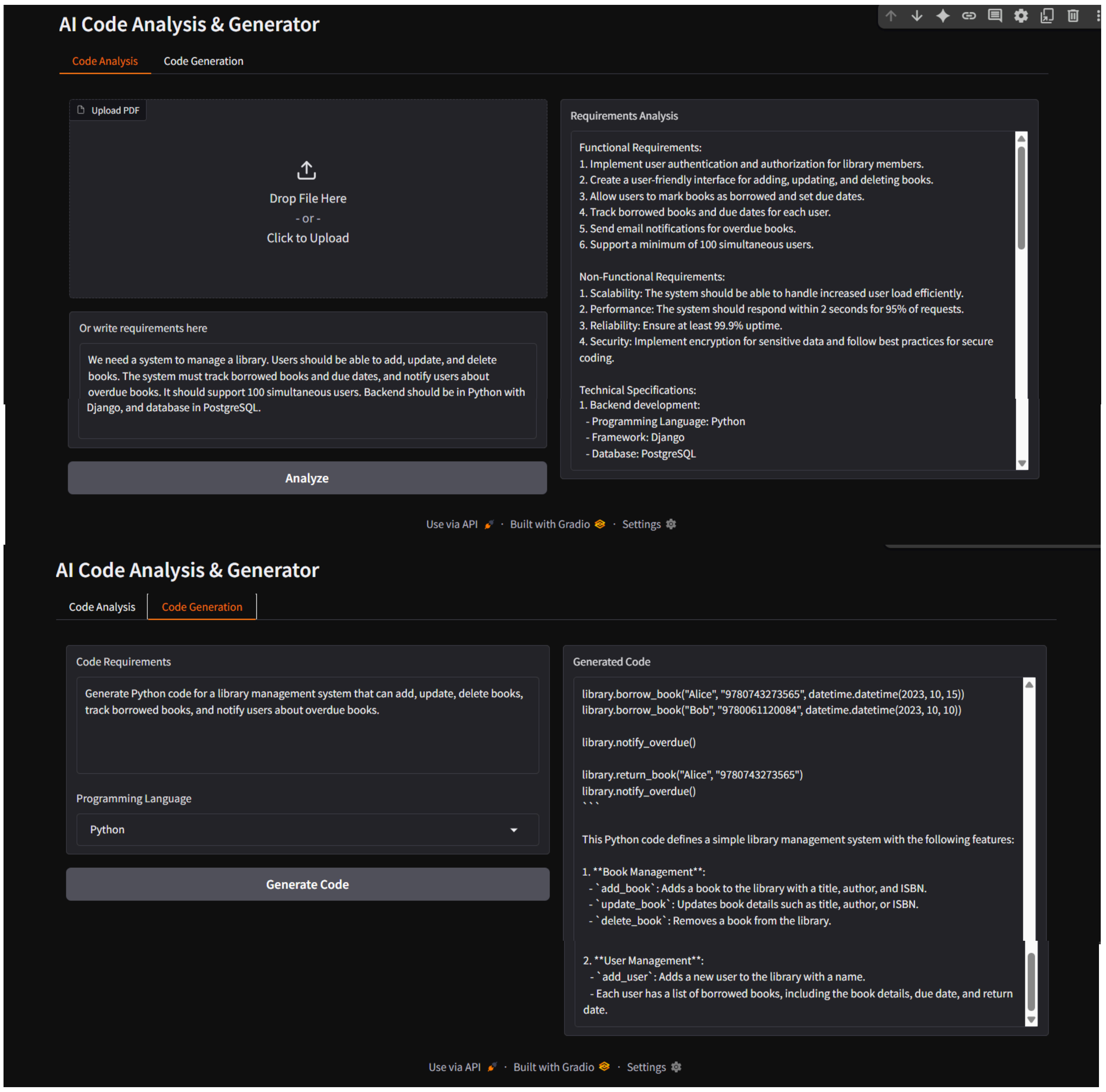


```
tokenizer_config.json:  8.88k/? [00:00<00:00, 579kB/s]
vocab.json:  777k/? [00:00<00:00, 4.38MB/s]
merges.txt:  442k/? [00:00<00:00, 8.43MB/s]
tokenizer.json:  3.48M/? [00:00<00:00, 40.3MB/s]
added_tokens.json: 100%  87.0/87.0 [00:00<00:00, 1.63kB/s]
special_tokens_map.json: 100%  701/701 [00:00<00:00, 16.1kB/s]
config.json: 100%  786/786 [00:00<00:00, 42.8kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json:  29.8k/? [00:00<00:00, 655kB/s]
Fetching 2 files: 100%  2/2 [02:18<00:00, 138.99s/it]
model-00001-of-00002.safetensors: 100%  5.00G/5.00G [02:18<00:00, 22.2MB/s]
model-00002-of-00002.safetensors: 100%  67.1M/67.1M [00:02<00:00, 23.6MB/s]
Loading checkpoint shards: 100%  2/2 [00:20<00:00,  8.47s/it]
generation_config.json: 100%  137/137 [00:00<00:00, 15.4kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://c3a638eb00b283e7de.gradio.live
```

AI Code Analysis & Generator
Code Analysis
Code Generation
Upload PDF
Drop File Here
- or -
Click to Upload
Requirements Analysis
Or write requirements here
Describe your software requirements...
Analyze
Use via API · Built with Gradio · Settings


AI Code Analysis & Generator

Sample Input :


AI Code Analysis & Generator

We need a system to manage a library. Users should be able to add, update, and delete books. The system must track borrowed books and due dates, and notify users about overdue books. It should support 100 simultaneous users. Backend should be in Python with Django, and database in PostgreSQL.

Functional Requirements:
1. Implement user authentication and authorization for library members.
2. Create a user-friendly interface for adding, updating, and deleting books.
3. Allow users to mark books as borrowed and set due dates.
4. Track borrowed books and due dates for each user.
5. Send email notifications for overdue books.
6. Support a minimum of 100 simultaneous users.

Non-Functional Requirements:
1. Scalability: The system should be able to handle increased user load efficiently.
2. Performance: The system should respond within 2 seconds for 95% of requests.
3. Reliability: Ensure at least 99.9% uptime.
4. Security: Implement encryption for sensitive data and follow best practices for secure coding.

Technical Specifications:
1. Backend development:
 - Programming Language: Python
 - Framework: Django
 - Database: PostgreSQL

AI Code Analysis & Generator

Generate Python code for a library management system that can add, update, delete books, track borrowed books, and notify users about overdue books.

```
library.borrow_book("Alice", "9780743273565", datetime.datetime(2023, 10, 15))
library.borrow_book("Bob", "9780061120084", datetime.datetime(2023, 10, 10))

library.notify_overdue()

library.return_book("Alice", "9780743273565")
library.notify_overdue()
```

This Python code defines a simple library management system with the following features:

1. **Book Management**:
 - `add_book`: Adds a book to the library with a title, author, and ISBN.
 - `update_book`: Updates book details such as title, author, or ISBN.
 - `delete_book`: Removes a book from the library.

2. **User Management**:
 - `add_user`: Adds a new user to the library with a name.
 - Each user has a list of borrowed books, including the book details, due date, and return date.

12.KNOWN ISSUES

During the development and testing of the **AI Code Analysis & Generator**, some limitations and issues were identified:

1. Model Accuracy:

- The AI model may sometimes generate incomplete or irrelevant requirements or code if the input prompt is vague or ambiguous.
- Generated code may require manual debugging before execution.

2. Performance Limitations:

- On systems without GPU support, response generation is slower due to heavy computation.
- Large PDF files with many pages may take longer to process, sometimes leading to memory usage spikes.

3. PDF Text Extraction Issues:

- Some PDF files, especially scanned documents or those with complex formatting, may not extract text correctly using PyPDF2.
- In such cases, missing or garbled text may affect requirement analysis quality.

4. User Interface Constraints:

- The Gradio interface is simple but may not handle very large outputs gracefully (long generated code may exceed textbox readability).
- Limited customization of layout and styling options.

5. Error Handling:

- Although basic error messages are shown (e.g., for unreadable PDFs), some technical errors are not fully explained to the end user.
- Unexpected exceptions from the AI model may cause the application to freeze or restart.

6. Language Support:

- While multiple programming languages are supported (Python, Java, C++, etc.), the generated code is more reliable for Python than for other languages.

- Advanced language-specific libraries or frameworks are not always handled correctly.

7. Internet Dependency:

- Model and tokenizer need to be downloaded from Hugging Face initially, which requires a stable internet connection.
- Running in offline environments can be problematic without pre-downloaded models.

13.FUTURE ENHANCEMENTS

To further improve the functionality and usability of the **AI Code Analysis & Generator**, the following enhancements are proposed:

1. Advanced PDF Handling:

- Integrate **OCR (Optical Character Recognition)** tools like Tesseract to process scanned or image-based PDFs.
- Improve text extraction for complex document layouts.

2. Improved Code Generation:

- Enhance model training with domain-specific datasets for more accurate and optimized code outputs.
- Add support for framework-based code generation (e.g., Django for Python, Spring Boot for Java).
- Implement an option to generate **unit test cases** along with the main code.

3. User Interface Upgrades:

- Provide a **download option** for generated code and analysis reports.
- Add syntax highlighting for better readability of generated code.
- Introduce a **dark/light mode** toggle for user preference.

4. Error Handling Improvements:

- Provide detailed and user-friendly error messages.
- Implement automatic retries for model inference failures.

5. Performance Optimization:

- Enable caching of frequently used models to reduce load time.
- Optimize response time for large PDF documents and long prompts.
- Support distributed or cloud-based deployment for handling multiple users simultaneously.

6. Extended Language and Framework Support:

- Add support for additional programming languages (e.g., Kotlin, Swift, Ruby).
- Enable multi-language code generation in a single request.

7. Offline Mode:

- Allow pre-downloading of the model and tokenizer so the system can function without internet connectivity.

8. Collaboration Features:

- Add options for multiple users to work together on the same project in real time.
- Provide version control integration (e.g., GitHub, GitLab) for directly storing generated code.