AI Code Analysis & Generator

**Project Documentation**

**1. Introduction:**

* Project Title: AI Code Analysis & Generator
* Team Leader: Shalini.K
* Team Member:Priyadharshini.G
* Team Member:Sandhiya.S
* Team Member:Rajeshwari.D

**2. Project Overview:**

**Purpose:**

The AI Code Analysis & Generator project is designed with the purpose of enhancing productivity, accuracy, and efficiency in the early stages of the Software Development Life Cycle (SDLC).

In many software projects, one of the most challenging and error-prone steps is translating requirement documents into clear specifications and eventually into working code. Developers and project managers often spend significant time reading lengthy documents, categorizing requirements, and ensuring nothing is overlooked. Furthermore, writing initial code prototypes based on these requirements can be time-consuming, especially when experimenting with multiple programming languages.

This project addresses these challenges by leveraging AI-powered natural language processing (NLP) and large language models (LLMs) to automatically:

* Extract and analyze requirements from documents or text inputs.
* Organize requirements into well-defined categories: functional requirements, non-functional requirements, and technical specifications.
* Generate code snippets directly from requirement descriptions in the user’s preferred programming language.

**Features:**

**Requirement Analysis from Documents**

Upload a PDF requirement document or manually type text.

Automatically extract content using PyPDF2.

AI-powered classification into functional, non-functional, and technical specifications.

**Code Generation Across Multiple Languages**

Accepts natural language descriptions of software requirements.

Generates corresponding code snippets in various programming languages including Python, Java, C++, C#, JavaScript, PHP, Go, and Rust.

Supports rapid prototyping and language exploration.

**AI-Powered Insights**

Uses the IBM Granite 3.2B Instruct model, a large language model optimized for instruction-following tasks.

Capable of both understanding natural language requirements and generating structured outputs.

**Interactive Web Interface (Gradio)**

Code Analysis: For analyzing and categorizing requirements.

Code Generation: For generating code in the selected programming language.

**Cross-Platform Compatibility**

Runs efficiently on both CPU and GPU environments.

Automatically detects GPU availability for performance optimization.

**Scalability and Extensibility**

Designed with modular functions (requirement\_analysis, code\_generation, etc.) that can be reused or integrated into larger applications.

Future enhancements can extend support to additional file formats, APIs, and collaborative workflows.

**3. Architecture:**

This modular architecture ensures clarity, scalability, and ease of maintenance. Each layer plays a distinct role while working together to create a smooth and interactive experience for the end users .

* **Frontend (Gradio Web Interface)**

Technology Used: Gradio

Purpose: Acts as the entry point for users, providing an interactive web interface that is accessible through a local browser or a public share link.

Key Responsibilities:

Accept user input in the form of PDF uploads or typed text.

Provide an interface to enter requirement descriptions for code generation.

Allow selection of a programming language from a dropdown menu.

Display the AI-generated requirement analysis or code snippets.

UI Design:

Uses tabbed layouts for switching between “Code Analysis” and “Code Generation.”

Contains file upload components, text boxes, dropdown menus, buttons, and large output display areas.

Focused on simplicity and accessibility, ensuring both technical and non-technical users can interact easily.

* **Backend (Application Logic – Python)**

Technology Used: Python, PyTorch, PyPDF2, Transformers

Purpose: Serves as the bridge between the user interface and the AI model. It processes inputs, prepares them for the model, and formats outputs for display.

Key Responsibilities:

Text Extraction: Uses PyPDF2 to extract raw text from uploaded PDF files.

Prompt Preparation: Formats user input into well-structured prompts for the AI model (e.g., converting requirement descriptions into instruction-style prompts).

Model Communication: Interacts with the IBM Granite model via the Hugging Face Transformers library.

Response Handling: Cleans up model outputs (removing duplicate prompts, special tokens, etc.) before displaying them.

Error Handling: Detects and reports errors (e.g., corrupted PDFs, overly large inputs).

* **AI Model (Intelligence Layer – IBM Granite LLM)**

Technology Used: IBM Granite 3.2B Instruct Model (via Hugging Face Transformers)

Purpose: Provides the intelligence and reasoning behind requirement analysis and code generation.

Key Responsibilities:

Requirement Analysis: Classifies extracted text into functional, non-functional, and technical specifications.

Natural Language Understanding: Interprets user-entered requirement descriptions.

Code Generation: Produces code snippets in different programming languages based on user requests.

Performance Optimization:

Automatically runs on GPU (CUDA) if available, otherwise falls back to CPU.

Uses optimized tensor formats (torch.float16 on GPU for faster inference).

**4. Setup Instructions:**

The setup process for the AI Code Analysis & Generator involves preparing the environment, installing dependencies, and running the application. This section provides detailed prerequisites and step-by-step installation instructions to ensure smooth deployment.

* **Prerequisites**

Before setting up the project, ensure the following prerequisites are met:

**1. System Requirements**

Operating System: Windows 10/11, macOS, or Linux (Ubuntu preferred)

RAM: Minimum 8 GB (16 GB recommended for smooth performance)

Disk Space: At least 5 GB free for model downloads and dependencies

Processor: Multi-core CPU (Intel i5/Ryzen 5 or higher recommended)

GPU: Optional, but a CUDA-compatible GPU (NVIDIA with 6 GB+ VRAM) is recommended for faster AI model inference

**2. Software Requirements**

Python 3.9 or later

pip (Python package manager)

Git (for cloning repositories, optional but recommended)

Virtual environment tool (venv or conda)

**3. Libraries and Frameworks**

PyTorch (with CUDA support if GPU available)

Transformers (for AI model handling)

Gradio (for the web-based frontend)

PyPDF2 (for text extraction from PDFs)

* **Installation Process**

Follow these steps to install and configure the project:

**Step 1: Clone the Repository**

If the project is hosted on GitHub or another platform, clone it:

git clone <your-repository-link>

cd ai-code-analysis-generator

If you have the project files locally, navigate to the project director.

**Step 2: Create a Virtual Environment**

It is recommended to use a virtual environment to isolate dependencies.

On Linux/Mac:

python3 -m venv venv

source venv/bin/activate

On Windows:

python -m venv venv

venv\Scripts\activate

**Step 3: Install Dependencies**

Install all required Python packages:

pip install torch transformers gradio PyPDF2

If you are using GPU, install PyTorch with CUDA (based on your CUDA version):

# Example for CUDA 11.8

pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

**Step 4: Verify GPU Compatibility (Optional)**

To check if PyTorch detects your GPU:

import torch

print(torch.cuda.is\_available())

If True, the model will run on GPU automatically.

If False, the model will run on CPU (slower but still functional).

**Step 5: Run the Application**

Start the application with:

python app.py

Once launched:

A local URL (e.g., http://127.0.0.1:7860) will be displayed in the terminal.

If share=True is enabled in app.py, a public URL will also be generated, allowing remote access.

* **Notes on Setup**

Model Download: The IBM Granite model will be automatically downloaded from Hugging Face on the first run. Ensure you have a stable internet connection.

**Performance Consideration:**

On CPU, response generation may be slower (10–30 seconds for complex prompts).

On GPU, inference speed is significantly improved.

Virtual Environment Best Practices: Always activate your environment before running the project to avoid dependency conflicts.

**5. Folder Structure:**

The project is organized into a modular folder structure to ensure clarity, maintainability, and scalability. Each folder or file serves a specific purpose, and together they provide a well-structured environment for development, testing, and documentation.

**Folder & File Descriptions**

* app.py

The main entry point of the application.

Contains the Gradio UI setup, tab layouts, button handlers, and connections to backend functions.

Integrates requirement analysis and code generation modules.

* requirements.txt

Lists all necessary dependencies for the project (torch, transformers, gradio, PyPDF2, etc.).

Makes installation easier by allowing users to run:

pip install -r requirements.txt

* README.md

Provides a high-level overview of the project.

Contains quick setup instructions and usage details.

Useful for anyone new to the project.

* /data/

Stores sample PDFs used for testing the requirement analysis feature.

Example: sample1.pdf might contain a mock software requirement specification (SRS).

Users can place their own documents here for local testing.

* /docs/

Contains documentation files such as project reports, manuals, and reference guides.

Useful for maintaining project deliverables (like this documentation).

* /models/ (Optional)

Stores locally downloaded AI models for offline usage.

If the Hugging Face model is cached here, the project can run without re-downloading each time.

Example: granite\_model/ stores the IBM Granite LLM.

* /tests/

Contains testing scripts for unit and integration testing.

Example test files:

test\_pdf\_extraction.py – Verifies PDF text extraction.

test\_code\_generation.py – Tests correctness of generated code.

test\_requirements\_analysis.py – Ensures proper categorization of requirements

* /utils/

Contains helper modules that make the codebase cleaner and modular.

The core application logic remains in app.py.

Supporting modules are neatly separated into /utils/.

Tests and documentation are organized for maintainability.

The design supports scalability if new features (e.g., database integration, API endpoints) are added later.

**6. Running the Application:**

The AI Code Analysis & Generator application is designed to run as a standalone interactive web app powered by Gradio. Once launched, it provides users with a browser-based interface where they can perform requirement analysis and generate code in real time.

* Launching the Application

**Activate your virtual environment (if created):**

Linux/Mac:

source venv/bin/activate

Windows:

venv\Scripts\activate

**Run the main script:**

python app.py

3. Access the application:

The terminal will display a local URL (e.g., http://127.0.0.1:7860).

Open this URL in a web browser to access the interface.

If share=True is enabled in the code, Gradio will also provide a public URL, which can be shared with others for remote access.

* **Application Workflow**

The system operates in two main modes:

**Requirement Analysis Mode**

Navigate to the Code Analysis tab.

Step 1: Upload a PDF file containing requirement specifications OR type requirements manually into the text box.

Step 2: Click the Analyze button.

Step 3: The backend extracts and processes the input, then sends a formatted prompt to the AI model.

Step 4: The AI model categorizes the requirements into Functional, Non-Functional, and Technical Specifications.

Step 5: The structured results are displayed in the output box.

**Code Generation Mode**

Navigate to the Code Generation tab.

Step 1: Type a requirement description in the text box (e.g., “Create a Python function that calculates factorial using recursion”).

Step 2: Select the target programming language from the dropdown (Python, Java, C++, etc.).

Step 3: Click the Generate Code button.

Step 4: The backend prepares a code-generation prompt and sends it to the AI model.

Step 5: The AI model returns a code snippet in the selected language.

Step 6: The code is displayed in the output box.

* **Frontend–Backend Interaction**

**Frontend (Gradio)**

Provides two tabbed sections for different workflows.

Handles file uploads, text input, dropdown selections, and output display.

**Backend (Python Logic)**

Extracts text (if PDF is uploaded).

Prepares AI prompts (requirement analysis or code generation).

Cleans and formats model outputs.

**AI Model (Granite LLM)**

Executes NLP-based reasoning for requirement analysis.

Generates code snippets based on natural language input.

Returns structured results back to the backend.

* **Execution Summary**

Start Command: python app.py

Access via Browser: Localhost (127.0.0.1:7860) or public link (if enabled)

Two Modes of Use:

Requirement Analysis (PDF/Text → Categorized Requirements)

Code Generation (Text Prompt + Language → AI-Generated Code)

Output Display: Structured text and code snippets in large output boxes.

**7. API Documentation:**

Although the AI Code Analysis & Generator project is primarily designed as a Gradio web application, its backend functions can be treated as internal APIs. These functions can also be extended into REST APIs in the future for broader integration.

* **Function: generate\_response(prompt, max\_length=1024)**

Description: Generates a natural language or code-based response from the AI model based on a given prompt.

Parameters: prompt (str) – Input text describing the requirement or question.

max\_length (int, optional) – Maximum number of tokens to generate (default: 1024).

Returns: (str) – AI-generated response (plain text or code).

* **Function: extract\_text\_from\_pdf(pdf\_file)**

Description: Extracts text content from an uploaded PDF file using PyPDF2.

Parameters: pdf\_file (file object) – PDF file uploaded by the user.

Returns: (str) – Extracted plain text from the PDF.

* **Function: requirement\_analysis(pdf\_file, prompt\_text)**

Description: Analyzes requirements either from a PDF file or from manually entered text. The requirements are then categorized into Functional Requirements, Non-Functional Requirements, and Technical Specifications.

Parameters: pdf\_file (file object or None) – PDF document containing requirements (optional).

prompt\_text (str) – Manually entered requirement text (used if no PDF is provided).

Returns: (str) – Structured requirement analysis output.

* **Function: code\_generation(prompt, language)**

Description: Generates code in the specified programming language based on a natural language requirement description.

Parameters: prompt (str) – Requirement description.

language (str) – Target programming language (Python, Java, C++, C#, PHP, Go, Rust, JavaScript).

Returns: (str) – AI-generated code snippet.

* **API Integration Possibility**

Currently, these functions are integrated into the Gradio interface. However, they can be extended into REST APIs using frameworks such as FastAPI or Flask to expose endpoints for:

Requirement Analysis

Code Generation

PDF Upload and Text Extraction

**8. Authentication:**

Currently, the AI Code Analysis & Generator runs in an open-access environment, meaning any user who accesses the application through the Gradio interface can fully utilize its features without needing credentials. This approach makes it simple for demonstration, academic use, and prototyping. However, it does not enforce any security restrictions, user tracking, or usage control.

* **Current Implementation**

No login or registration mechanism is required.

All users have equal access to requirement analysis and code generation features.

The application operates entirely in a stateless manner – user interactions are not stored or linked to user identities.

This setup is ideal for testing, classroom demonstrations, and local use, but it is not suited for production environments where multiple users may need differentiated access.

* **Limitations of the Current Setup**

No User Authentication: Anyone with access to the link can use the system.

No Role-Based Access Control (RBAC): There is no distinction between normal users and administrators.

No Usage Monitoring: User actions and usage statistics are not logged.

No Data Privacy Controls: Since users are not uniquely identified, session history cannot be secured

* **Future Authentication Enhancements**

For secure deployments in enterprise or cloud environments, the following authentication mechanisms can be integrated:

**1. Token-Based Authentication (JWT / API Keys):**

Secure API endpoints using tokens or API keys.

Ensures only authorized applications or users can interact with the backend functions.

**2. OAuth2 Integration:**

Allow login via trusted identity providers such as Google, GitHub, or enterprise Single Sign-On (SSO).

Suitable for team or organization-level deployments.

**3. Role-Based Access Control (RBAC):**

Define roles such as Admin, Developer, and Viewer.

Admins could manage settings, while users would only be able to analyze requirements or generate code.

**4. User Session Management:**

Track individual user sessions.

Enable personalized features like saving past analyses, generating usage reports, or resuming previous work.

**5. Audit and Logging Mechanisms:**

Record user interactions for monitoring, debugging, and compliance purposes.

Useful in educational or enterprise scenarios where accountability is important.

**9. User Interface:**

The AI Code Analysis & Generator features a web-based user interface (UI) built using Gradio. The design prioritizes simplicity, clarity, and accessibility, ensuring that both technical and non-technical users can easily interact with the application. The interface is structured around two main functional areas: Requirement Analysis and Code Generation.

* **Design Principles**

Minimalist Layout: The interface avoids unnecessary complexity, focusing on essential inputs and outputs.

Tabbed Navigation: Functions are divided into separate tabs to clearly distinguish between analysis and generation workflows.

Interactive Components: File uploads, text boxes, dropdown menus, and buttons provide a smooth, interactive experience.

Real-Time Display: Results are displayed instantly in large text boxes, allowing users to view outputs without delays.

Cross-Platform Accessibility: The UI can be accessed via any modern web browser, on both desktop and mobile devices.

* **Layout Overview**

The UI is organized into two main tabs:

**1. Code Analysis Tab**

File Uploader: Allows users to upload a PDF requirement document.

Text Input Box: Alternative option to type requirements manually.

Analyze Button: Initiates requirement analysis.

Output Display Box: Shows categorized requirements as Functional, Non-Functional, and Technical Specifications.

**2. Code Generation Tab**

Text Input Box: Allows users to describe the required functionality (e.g., “Write a program to calculate factorial”).

Programming Language Dropdown: Provides a list of supported languages (Python, Java, C++, C#, PHP, Go, Rust, JavaScript).

Generate Button: Initiates AI-driven code generation.

Output Display Box: Shows the generated code snippet.

* **User Interaction Flow**

1. **Requirement Analysis Flow:**

User uploads a PDF or types **requirements** manually.

Clicks the Analyze button.

Output is displayed in the right-hand panel, categorized into requirement types.

**2. Code Generation Flow:**

User enters a requirement description in text form.

Selects a target programming language from the dropdown.

Clicks the Generate Code button.

Generated code is displayed instantly in the output panel.

* **Usability Features**

Error Handling Messages: If a PDF cannot be read or if input is missing, the UI displays an error message instead of crashing.

Flexible Input Options: Users can either upload files or directly type input text, increasing convenience.

Wide Output Boxes: Large multi-line text boxes are used to ensure readability of long AI-generated content.

Copy-Paste Friendly: Outputs can easily be copied from the result box for further use in IDEs or documentation.

* **Planned UI Enhancements**

Syntax Highlighting: Future versions can highlight generated code for better readability.

Downloadable Reports: Add buttons to export requirement analysis or code snippets as PDF/Word files.

Dark Mode Support: Improve accessibility by offering theme customization.

Multi-Language UI: Option to provide instructions and labels in multiple natural languages.

**10. Testing:**

Testing is a critical part of ensuring that the AI Code Analysis & Generator functions reliably, delivers accurate outputs, and provides a smooth user experience. The project underwent multiple levels of testing, including unit testing, integration testing, and user testing. This section describes the testing strategy, coverage, and outcomes.

* **Testing Objectives**

Verify the correctness of core functions such as requirement analysis, PDF text extraction, and code generation.

Ensure smooth integration between the frontend (Gradio UI) and the backend functions.

Validate that the AI model outputs are accurate, readable, and properly categorized.

Test the usability of the interface to confirm it is intuitive and accessible for different types of users.

* **Unit Testing**

Unit tests were developed to verify the functionality of individual components.

PDF Text Extraction (extract\_text\_from\_pdf)

Tested with valid PDFs containing structured text.

Tested with empty, corrupted, or scanned PDFs to validate error handling.

Response Generation (generate\_response)

Tested with short requirement prompts to ensure the model responds without errors.

Validated maximum token handling (max\_length parameter).

Requirement Categorization (requirement\_analysis)

Verified that functional, non-functional, and technical requirements were correctly separated.

Code Generation (code\_generation)

Checked if code was generated in the selected programming language.

Ensured syntax was consistent with the chosen language (Python indentation, Java class structure, etc.).

* **Integration Testing**

Integration tests ensured that different parts of the system worked seamlessly together.

Frontend–Backend Communication

Confirmed that file uploads, text inputs, and dropdown selections in the UI were correctly passed to backend functions.

Validated that outputs were displayed without truncation or formatting errors.

End-to-End Workflows

**Requirement Analysis Workflow:** PDF → Extract Text → Model Analysis → Categorized Output.

Code Generation Workflow: Requirement Text → Prompt Preparation → Model Inference → Code Output.

* **User Testing**

User testing was conducted to validate the interface and overall user experience.

Test Audience: Students, software developers, and academic users.

**Focus Areas:**

Ease of uploading PDFs and entering text.

Clarity of categorized requirement outputs.

Accuracy and usefulness of generated code snippets.

Responsiveness of the application across browsers (Chrome, Firefox, Edge).

* **Edge Case Testing**

Large PDF files with 50+ pages tested for performance.

Special characters and formatting in PDFs (tables, diagrams) checked for graceful handling.

Long prompts tested for token overflow prevention.

Code generation tested for unsupported languages to ensure proper error messages were shown.

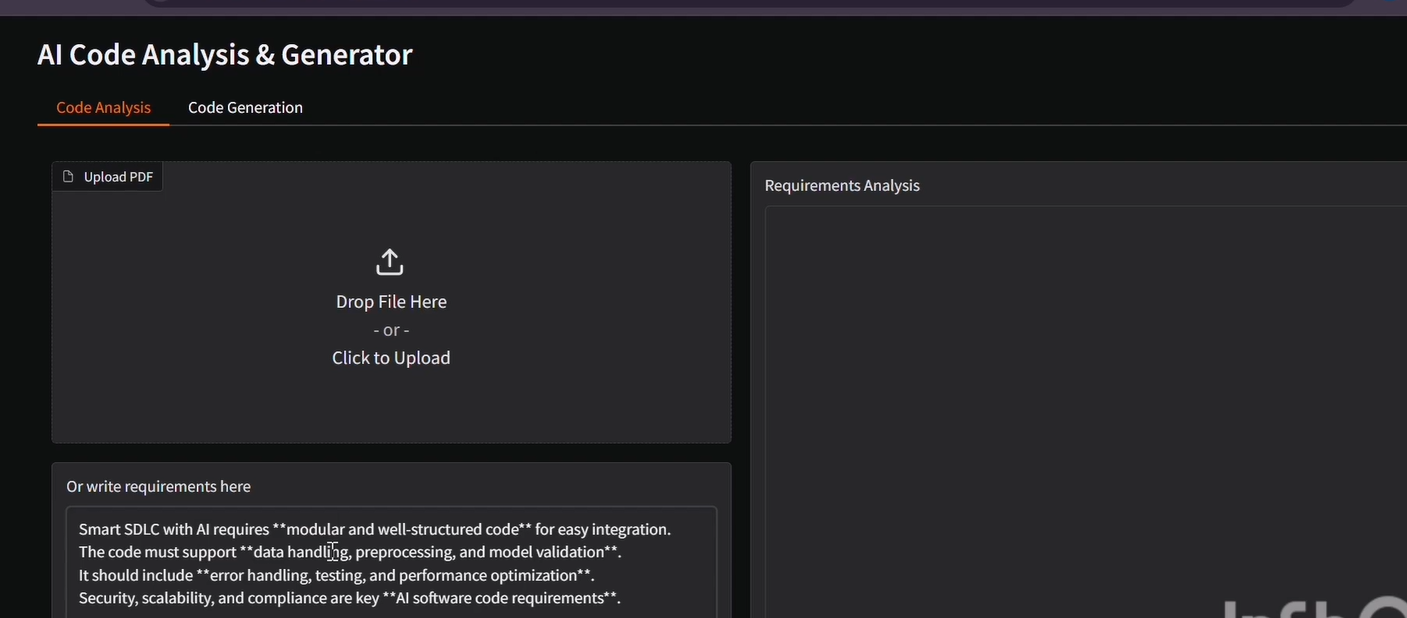
* **Results**

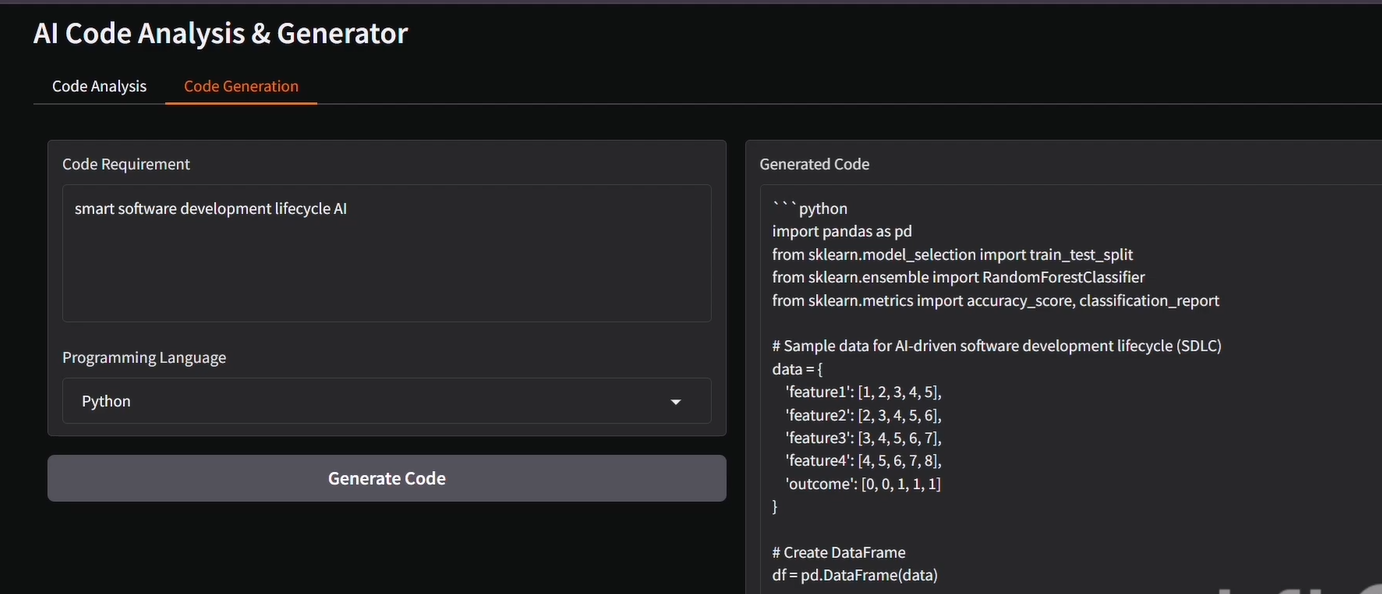
Core features (requirement analysis, code generation) worked reliably for the majority of test cases.

The application was stable under normal use but slower when handling very large PDFs or running only on CPU.

User interface testing confirmed that the design is accessible and functional.

**11.Screenshots:**

****

****

**12. Known Issues:**

While the AI Code Analysis & Generator performs its intended functions effectively, several limitations and known issues have been identified during testing and usage. These issues do not prevent the application from functioning but may impact performance, usability, or accuracy under certain conditions.

**12.1 Performance Issues**

Slow Response on CPU:

When running without a GPU, the AI model inference is slower, especially for complex prompts or large PDF documents.

Long prompts may take up to 30 seconds to process on CPU-only systems.

High Memory Usage:

The model consumes a significant amount of RAM/VRAM. On systems with limited resources, this can cause lag or process termination.

**12.2 PDF Handling Limitations**

Scanned PDFs Not Supported:

Since the project uses PyPDF2, only text-based PDFs can be processed.

Scanned PDFs or image-based documents cannot be analyzed unless OCR (Optical Character Recognition) is integrated.

Complex Formatting Issues:

Tables, diagrams, or highly formatted text in PDFs may not be extracted properly.

This can result in partial or inconsistent requirement extraction.

**12.3 AI Model Limitations**

Inconsistent Categorization:

While most requirements are classified correctly, the AI model occasionally misclassifies requirements between functional, non-functional, and technical categories.

Code Hallucination:

The model sometimes generates incomplete or logically incorrect code snippets.

For advanced or ambiguous requirements, manual debugging and corrections may be required.

Limited Context Retention:

Very long input documents may exceed the model’s token limit, leading to truncated or incomplete outputs.

**12.4 User Interface Constraints**

Lack of Code Formatting:

The generated code is displayed as plain text without syntax highlighting, making it harder to read in the UI.

No Save/Export Option:

Outputs cannot yet be downloaded directly as PDF or Word reports. Users need to manually copy and paste results.

Single-User Mode:

The application does not currently support multi-user sessions. Multiple users accessing the app simultaneously may face performance slowdowns.

**12.5 Authentication and Security**

Open Access Mode:

The system has no built-in authentication or access control.

Any user with the application link can fully access the features.

**13. Future Enhancement:**

The AI Code Analysis & Generator has proven effective in automating requirement analysis and code generation. However, there are several opportunities to enhance its capabilities, usability, and scalability. These enhancements can transform the tool from a proof-of-concept into a more robust, production-ready system.

**13.1 Enhanced File Support**

**OCR** **Integration** **for** **Scanned** **PDFs**:

Incorporate OCR (Optical Character Recognition) libraries such as Tesseract to process image-based or scanned PDFs.

Expand input compatibility beyond text-based documents.

**Additional** **File** **Formats**:

Support for Microsoft Word (.docx), plain text (.txt), and Markdown (.md) files.

Allow batch file uploads for analyzing multiple documents at once.

**13.2 Improved AI Capabilities**

**Advanced** **Requirement** **Classification**:

Use fine-tuned models for more accurate categorization of functional, non-functional, and technical requirements.

Provide structured outputs in tabular or bullet-point formats for clarity.

**Code** **Validation** **and** **Debugging**:

Integrate a code execution sandbox to automatically test generated code.

Highlight errors and suggest corrections in real time.

**Context**-**Aware** **Generation**:

Enable the model to handle larger documents by splitting them into sections and maintaining context.

Summarize requirements before categorization to reduce token overflow issues.

**13.3 User Interface Enhancements**

Syntax Highlighting: Display generated code with proper indentation and color-coded syntax for better readability.

Allow users to export requirement analysis and code outputs directly as PDF, Word, or CSV reports.

**Collaboration** **Features**:

Support multiple users in a shared workspace.

Enable team members to annotate requirements or edit generated code collaboratively.

**Customizable** **Themes**:

Add support for dark mode and customizable layouts to enhance user experience.

**13.4 Authentication and Security**

**User** **Authentication**:

Add login functionality with OAuth2 or JWT-based authentication.

Differentiate between roles (e.g., Admin, Developer, Viewer).

**Usage** **Tracking** **and** **Quotas**:

Monitor user activity to track resource consumption.

Implement usage quotas for free vs. premium access.

**13.5 Deployment and Integration**

**Cloud** **Deployment**:

Deploy as a web service using platforms like AWS, Azure, or IBM Cloud.

Provide a scalable **architecture** capable of handling multiple concurrent users.

**API Exposure:**

Extend backend functions as REST APIs using FastAPI or Flask.

Allow integration with third-party applications, IDEs, or CI/CD pipelines.

Database Integration:

Store analyzed requirements, generated code, and user sessions in a relational or NoSQL database.

Enable search and retrieval of past analyses for long-term projects.

**13.6** **Educational** **and** **Enterprise** **Extensions**

Learning Mode for Students

Provide explanations alongside generated code to help learners understand the logic.

Enterprise Mode for Professionals:

Integrate with project management tools like JIRA, Trello, or Confluence.

Allow exporting requirement summaries directly into project documentation workflows.