Sustainable Smart City Assistant Using IBM Granite LLM

A Project Report
Submitted to the Naan Mudhalvan course for the award of degree of

BACHELOR OF COMPUTER SCIENCE

Submitted By
(**Team-Id: NM2025TMID04713**)

Name	**Reg. No**
MUKESH M	222305214
SIVARAMAKRISHNAN U	222305228
YOKESH C	222305232
RAGHU M	222305217

Department of Computer Science

PACHAIYAPPA'S COLLEGE

(Affiliated to the University of Madras)
CHENNAI – 600030.

November – 2025.

PACHAIYAPPA'S COLLEGE

(Affiliated to the University of Madras)

CHENNAI-600 030.

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to certify that the Naan Mudhalvan project work entitled **"Sustainable Smart City Assistant Using IBM Granite LLM"** is the Bonafide record of work done by **(Mukesh - Sivaramakrishnan - Yokesh - Raghu)** in partial fulfillment for the award of the degree of **Bachelor of Computer Science,** under our guidance and supervision, during the academic year 2025-2026.

Head of the Department
Dr. J. KARTHIKEYAN

Staff-in-Charge
Mr. K. DINESHKUMAR

Submitted for Viva-Voce Examination held on…….a Pachaiyappa's College, Chennai-30.

Internal Examiner

External Examiner

Sustainable Smart City Assistant Using IBM Granite LLM

Table of Contents

S. No.	Title	Page No.
1	Introduction	3
2	Project Overview	4
3	Architecture	6
4	Setup Instructions	9
5	Folder Structure	11
6	Running the Application	12
7	API Documentation	14
8	Authentication	15
9	User Interface	17
10	Testing	19
11	Screenshots	21
12	Known Issues	25
13	Future Enhancement	26

1. Introduction

Project Title: Sustainable Smart City Assistant Using IBM Granite LLM

Team Members:

S. No.	Name	Reg. No
1	MUKESH M	222305214
2	SIVARAMAKRISHNAN U	222305228
3	YOKESH C	222305232
4	RAGHU M	222305217

Program Background:

The IBM Naan Mudhalvan Smart Internz Program provides students with industry-aligned training in AI/ML, cloud computing, and generative AI. It emphasizes solving real-world urban challenges with modern AI-driven solutions.

The Smart City AI project leverages IBM Watsonx Granite LLM integrated with Streamlit, FastAPI, Pinecone, and ML modules to build a digital assistant for sustainable urban management. The system is capable of chatting with citizens and officials, summarizing lengthy policy reports, forecasting city KPIs, and detecting anomalies in city data.

2. Project Overview

Purpose

The Smart City AI Assistant is designed to act as a bridge between citizens, administrators, and urban data systems. In modern cities, vast amounts of data are generated daily from transport, healthcare, energy, water, and environmental monitoring systems. However, this data is often underutilized due to its complexity and lack of accessibility for decision-makers and common citizens.

The project aims to:

- Simplify complex city-level reports and datasets into actionable insights.
- Improve resource management by predicting and optimizing the use of energy, water, and waste disposal.
- Foster citizen engagement through an interactive AI assistant that listens, responds, and provides guidance.
- Empower city officials with real-time anomaly detection to quickly address critical issues (like sudden spikes in pollution or abnormal water wastage).
- Contribute towards the UN Sustainable Development Goals (SDGs), particularly Sustainable Cities and Communities (SDG 11), by integrating AI into urban planning and governance.

Features

1. Conversational Chatbot

- Citizens can ask questions like *"What is today's air quality index in my area?"* or *"What steps can I take to reduce electricity consumption?"*.
- Officials can query the system for summarized insights from large policy documents or reports.
- Built using IBM Watsonx Granite LLM for natural language understanding and contextual responses.

2. Policy Summarization

- Upload lengthy government policies, budget reports, or environmental studies.
- AI condenses them into key points, pros/cons, and recommendations.
- Helps both administrators (fast decision-making) and citizens (easy awareness).

3. Forecasting Module

- Uses historical city data (energy usage, water supply, healthcare demand) to predict future trends.
- Example: If water usage is expected to rise 20% next summer, officials can plan reservoir management in advance.
- Implements time-series forecasting models (e.g., ARIMA, Prophet, or regression).

4. Eco & Healthcare Tips

- Personalized recommendations based on city trends and user queries.
- Examples:
    - For citizens → *"Switch to LED bulbs, it reduces energy usage by 80% compared to CFLs."*
    - For healthcare → *"Increase fluid intake during peak summer; local data shows heatstroke risk is rising."*
- Reinforces sustainability awareness.

5. Feedback Collection

- Citizens can submit opinions, complaints, or improvement ideas.
- Data stored in backend for sentiment analysis (future enhancement).
- Helps administrators understand public mood and prioritize actions.

6. KPI Tracking (Key Performance Indicators)

- Real-time dashboards display indicators like:
    - Air Quality Index (AQI)
    - Electricity consumption per household
    - Water usage per capita
- Provides a health score for the city that officials can monitor.

7. Anomaly Detection

- Identifies sudden unusual patterns in city data.
- Example:
    - A sharp increase in hospital admissions might indicate an outbreak.
    - A sudden dip in electricity usage could suggest a power outage.
- Uses ML-based anomaly detection algorithms (Isolation Forest, DBSCAN).

8. Multimodal Input Support

- Accepts PDFs, CSVs, or plain text.
- Enables document-based Q&A (citizens or officials can upload files for AI-driven answers).
- Integrates Pinecone vector database for semantic search across uploaded data.

9. User Interface (UI)

- Developed in Streamlit for interactivity.
- Features:
    - Sidebar navigation for different modules.
    - Lightweight, responsive design for both desktop and mobile.

3. Architecture

The architecture of Smart City AI follows a modular, layered design, ensuring scalability, flexibility, and maintainability. Each component has a well-defined role, and together they enable seamless interaction between citizens, administrators, and AI-driven insights.

Frontend (Streamlit)

- Provides an **interactive web interface** for both citizens and city officials.
- Designed with a **sidebar-driven navigation system** for easy module access (Chatbot, Forecasting, Policy Summarization, KPI Dashboard).
- **Key Features:**
    - File upload support (PDF, CSV, text).
    - Data visualization (line charts, bar graphs, pie charts using **Matplotlib/Plotly**).
    - Real-time AI chat interaction.
    - Downloadable reports (PDF/CSV).
- **Why Streamlit?**
    - Lightweight, open-source, Python-native.
    - Rapid prototyping for data science apps.
    - Easy integration with FastAPI backend.

3.2 Backend (FastAPI)

- Acts as the **core service layer** that connects frontend requests to AI/ML models.
- Provides a **REST API layer** with endpoints for:
    - /chat/ask → LLM-based chat responses.
    - /upload-doc → Document ingestion + embedding.
    - /forecast → Predictive analysis.
    - /anomaly-check → Anomaly detection.
    - /summarize → Policy/document summarization.
- **Why FastAPI?**
    - High-performance, async-enabled web framework.
    - Built-in support for auto-generated **Swagger UI** and **OpenAPI docs**.
    - Lightweight compared to Django/Flask, making it ideal for AI services.

3.3 Large Language Model (LLM) – IBM Watsonx Granite

- The **core intelligence layer** powering natural language interactions.
- Functions:
  - Generates **human-like responses** to citizen queries.
  - Summarizes long government reports and documents.
  - Provides personalized **eco/healthcare tips**.
- **Advantages:**
  - Domain adaptability – Granite can be fine-tuned on city-related datasets.
  - Enterprise-ready – optimized for governance and business use cases.

3.4 Vector Database (Pinecone)

- Stores **document embeddings** for semantic search.
- **Process Flow:**
  1. User uploads a document (PDF/CSV).
  2. Document is chunked and embedded using **Granite embeddings**.
  3. Embeddings stored in Pinecone for fast retrieval.
  4. User query → converted to embedding → nearest match retrieved → passed to LLM for contextual answer.
- **Why Pinecone?**
  - High-performance vector database with millisecond-level search.
  - Handles scalability for thousands of documents.
  - Cloud-hosted, fully managed.

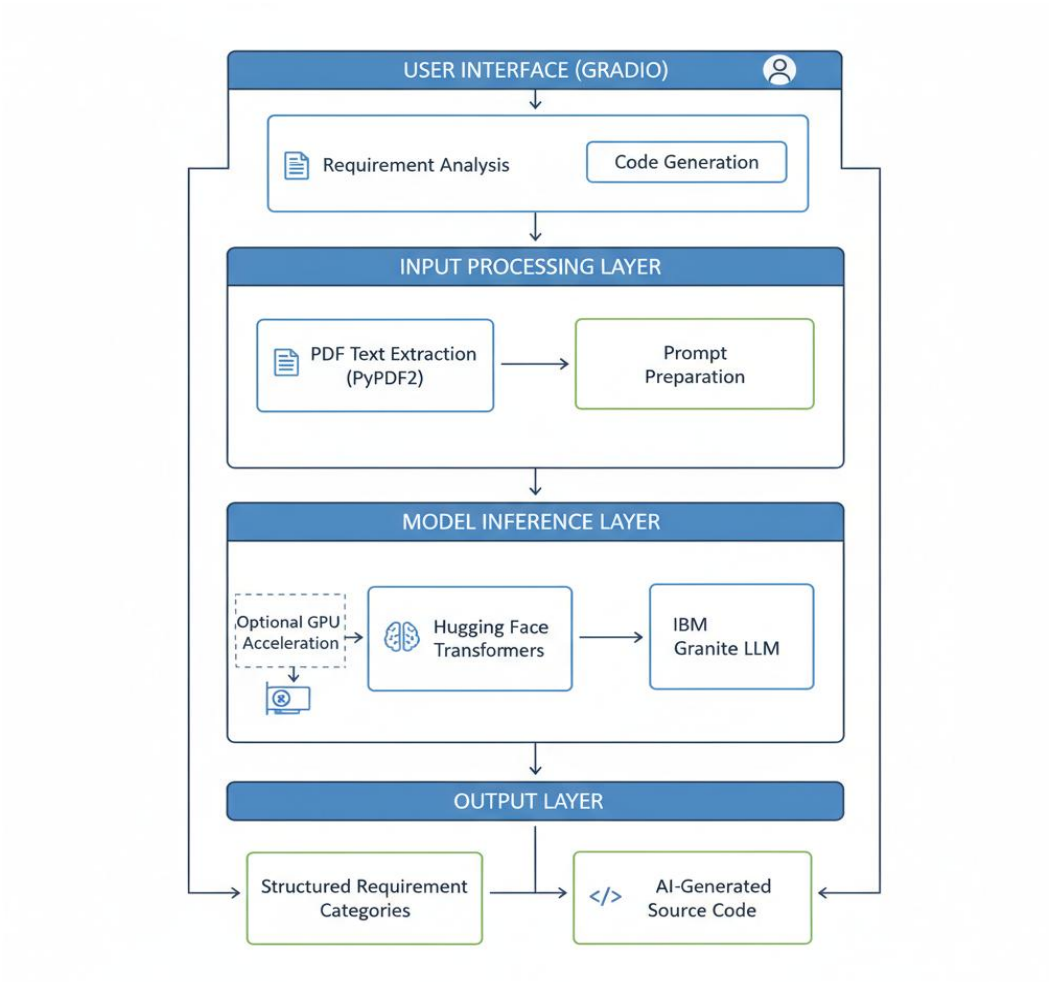
3.5 Machine Learning Modules

- Responsible for **predictive analysis** and **anomaly detection**.
- **Forecasting Module:**
  - Uses **time-series models (ARIMA, Prophet, regression models)**.
  - Predicts future city resource demands (water, energy, healthcare).
- **Anomaly Detection Module:**
  - Implements **Isolation Forest, Z-Score, or DBSCAN** for irregularity detection.
  - Identifies spikes in pollution, sudden drop in electricity usage, or abnormal hospital admissions.
- **Visualization:** Forecasting results displayed in **interactive charts** for better interpretability.

3.6 Data Flow (End-to-End)

1. **Citizen/Admin Interaction:** User uploads a file or sends a query via the **Streamlit frontend**.
2. **Request Handling:** Frontend forwards the request to the **FastAPI backend**.
3. **Processing Layer:**
    - If it's a query → processed by **Granite LLM**.
    - If it's a document → embeddings generated and stored in **Pinecone**.
    - If it's forecasting → ML model predicts future values.
    - If anomaly detection → ML algorithms identify irregular patterns.
4. **Response Generation:** Backend compiles results (LLM output, forecast graph, anomaly report).
5. **User Delivery:** Results displayed on Streamlit as **chat responses, visual charts, or downloadable reports**.

3.7 Architecture Diagram



4. Setup Instructions

4.1 Prerequisites

Before running the application, ensure the following are installed:

- Python: Version 3.9 or higher.
- pip: Python package manager.
- Virtual Environment (venv): Recommended for dependency isolation.
- Git: For cloning the repository.
- Node.js & npm (Optional): Only if integrating additional frontend components.
- API Keys Required:
    - IBM Watsonx Granite API Key.
    - Pinecone API Key.

4.2 Cloning the Repository

git clone https://github.com/username/smart-city-ai.git

cd smart-city-ai

4.3 Creating a Virtual Environment

```
# Create virtual environment
python -m venv
# Activate (Linux/Mac)
source venv/bin/activate
# Activate (Windows)
venv\Scripts\activate
```

4.4 Installing Dependencies

Install the required Python libraries from **requirements.txt**:

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

Key Dependencies:

- **fastapi** – Backend framework.
- **pinecone-client** – Vector database integration.
- **ibm-watsonx-ai** – Granite LLM access.
- **scikit-learn, pandas, matplotlib** – ML, data handling, visualization.

4.5 Environment Configuration

Create a .env file in the root directory and add your credentials:

```
IBM_API_KEY=your_ibm_watsonx_api_key
PINECONE_API_KEY=your_pinecone_api_key
PINECONE_ENV=us-east1-gcp
```

This ensures your sensitive credentials are not hardcoded into the source code.

4.6 Running the Backend

Use **Uvicorn** to launch the FastAPI backend:

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

- Runs on: http://127.0.0.1:8000
- Auto-generated API docs: http://127.0.0.1:8000/docs

4.7 Running the Frontend

Launch the **Streamlit dashboard**:

```
streamlit run ui/dashboard.py
```

Runs on: http://localhost:8501

4.8 Verifying the Setup

- Open **Streamlit UI** in the browser.
- Test chatbot by asking: *"What is today's air quality index?"*.
- Upload a sample **policy PDF** and check the summarization.
- Upload a **CSV file** and test forecasting + anomaly detection.

4.9 Troubleshooting

- **Issue: Module not found** → Run pip install -r requirements.txt again.
- **Issue: API errors** → Check if API keys are correctly set in .env.
- **Issue: Pinecone connection failed** → Ensure correct environment region is used (us-east1-gcp or as per your Pinecone dashboard).
- **Issue: Streamlit not launching** → Kill any process on port 8501 and retry:

```
kill -9 $(lsof -t -i:8501)

streamlit run ui/dashboard.py
```

5. Folder Structure

The Smart City AI project follows a modular and layered directory structure to separate responsibilities and improve maintainability.

```
ai-code-analyzer/
│
├── app.py                  # Main Gradio application (backend + frontend)
│
├── models/                 # Custom or fine-tuned Hugging Face models
│   └── (optional checkpoints or weights)
│
├── utils/                  # Helper functions and reusable modules
│   ├── pdf_utils.py        # PDF text extraction using PyPDF2
│   ├── prompt_utils.py     # Prompt templates for analysis & code generation
│   └── model_utils.py      # Model loading and inference helpers
│
├── notebooks/              # Google Colab or Jupyter notebooks for experiments
│   └── experiments.ipynb
│
├── requirements.txt        # Python dependencies
├── .env                    # API keys (e.g., Hugging Face token)
├── README.md               # Project overview and instructions
└── reports/                # Generated outputs and exported results
    └── sample_report.txt
```

6. Running the Application

This section explains how to set up and run the **AI Code Analysis & Generator** application, which leverages **IBM Granite LLM models (via Hugging Face)**, **Gradio** for the user interface, and **PyPDF2** for document parsing. The system provides two main capabilities: **requirement analysis from PDFs/text** and **automatic code generation** in multiple languages.

6.1 Environment Setup

Python Environment

Ensure **Python 3.9 or higher** is installed. Create a virtual environment and install dependencies:

```
python -m venv
source venv/bin/activate       # On Windows: venv\Scripts\activate
pip install torch transformers gradio pypdf2
```

Dependencies

- **transformers** – Loads Hugging Face's ibm-granite/granite-3.2-2b-instruct model.
- **torch** – Accelerates model inference (GPU if available).
- **gradio** – Provides a user-friendly web interface.
- **PyPDF2** – Extracts text content from uploaded PDFs.

Model Configuration

The application uses **IBM Granite 3.2-2B Instruct** hosted on Hugging Face. The model is automatically downloaded when first executed.

- GPU support is enabled if available (for faster response generation).
- The tokenizer is configured with an EOS token as padding for stability.

6.2 Running the Backend and Frontend

Unlike a traditional split system, the backend (AI logic) and frontend (user interface) are combined within the **Gradio Blocks** app.

Start the application with:

```
python app.py
```

(Replace app.py with the filename of your script.)

- This command loads the Granite LLM, initializes the tokenizer, and starts the Gradio server.
- The console will display a local URL (e.g., http://127.0.0.1:7860) and a public **share link** for external access.

6.3 Using the Application

Once running, open the provided **local URL** in a web browser. The interface has two main tabs:

1. **Code Analysis**
    - Upload a PDF containing software requirements **or** manually enter requirements in the text box.
    - Click **"Analyze"** to receive structured requirements categorized as:
        - Functional requirements
        - Non-functional requirements
        - Technical specifications
    - Output appears in the **Requirements Analysis** text area.
2. **Code Generation**
    - Enter a natural language description of the desired functionality.
    - Select the programming language from the dropdown (Python, Java, JavaScript, C++, etc.).
    - Click **"Generate Code"** to receive AI-generated implementation code.
    - Output appears in the **Generated Code** panel.

6.4 Integration with Experiments

Experiments may be run in **Google Colab** to fine-tune models or test prompts before deployment.

- Save results as custom model weights.
- Update the model_name parameter in the script to point to your fine-tuned model.
- Relaunch the Gradio app to integrate the new model.

7. API Documentation

The application exposes two main functional modules: **Requirement Analysis** and **Code Generation**. While these are accessed through the Gradio interface, they are implemented as functions that can be exposed as REST APIs in future iterations (via FastAPI).

7.1 Endpoints Overview

Endpoint	Method	Description
/analyze-requirements	POST	Extracts and classifies software requirements from text or uploaded PDF.
/generate-code	POST	Produces source code in the requested programming language based on a textual requirement.

7.2 API Specifications

1. Requirement Analysis

Endpoint: /analyze-requirements
Method: POST

Request Parameters:

- pdf_file *(optional)*: PDF file containing requirements.
- prompt_text *(optional)*: Plain text describing requirements.

2. Code Generation

Endpoint: /generate-code
Method: POST

Request Parameters:

- prompt *(string)*: Requirement description for code generation.
- language *(string)*: Target programming language (e.g., "Python", "Java", "C++").

7.3 Gradio Interaction

In the current implementation, these APIs are accessed **via the Gradio interface**:

- **Tab 1 (Code Analysis):** Calls /analyze-requirements under the hood.
- **Tab 2 (Code Generation):** Calls /generate-code with user-selected language.

Users interact through **forms, file upload widgets, and text areas**, with responses displayed instantly in the web dashboard.

8. Authentication

8.1 Current State

The current implementation of the **AI Code Analysis & Generator** system (developed with **Gradio**) does not include a dedicated authentication layer. All functionality — including **requirement analysis** and **code generation** — is publicly accessible through the local server (http://127.0.0.1:7860) or via the Gradio share link.

This design was chosen for **rapid prototyping and experimentation**, ensuring that users can immediately test model outputs without the overhead of credentials or user accounts. However, for deployment in enterprise or academic environments, robust authentication is essential.

8.2 Risks of Open Access

- **Unauthorized Usage**: Anyone with access to the share link can interact with the system.
- **Excessive API Calls**: Potential misuse could lead to resource exhaustion, especially if hosted on GPU servers.
- **Data Sensitivity**: Uploaded PDF files may contain confidential requirement documents.

8.3 Planned Authentication Mechanisms

Future versions will integrate one or more of the following authentication strategies:

1. Token-Based Authentication (JWT)

- Each user is issued a **JSON Web Token (JWT)** upon login.
- Subsequent API calls (e.g., /analyze-requirements, /generate-code) require the token in the header.
- Example request:
- POST /generate-code
- Authorization: Bearer <jwt_token>
- Provides **stateless and scalable authentication** suitable for cloud deployment.

2. OAuth2 (Third-Party Integration)

- Users authenticate via trusted identity providers (Google, IBM Cloud, GitHub).
- Ensures secure and familiar login for enterprise/academic users.
- Reduces need to manage passwords within the application.

3. API Key Access (Developer Mode)

- Developers request an API key, stored in the .env file or user profile.
- Useful for restricting access to a limited group during beta testing.
- Example .env entry:
- APP_API_KEY=example_key_12345
- Requests include:
- x-api-key: example_key_12345

4. Gradio Authentication (Immediate Integration)

Gradio provides a lightweight **built-in authentication decorator** for prototypes:

app.launch(auth=("username", "password"))

- Requires users to log in with credentials before accessing the interface.
- Useful for **internal demos** and **educational use cases**.
- Can be replaced later with more robust JWT or OAuth2 authentication.

8.4 Roadmap for Secure Deployment

1. **Short-term**:
    - Enable **Gradio username/password authentication** for shared links.
    - Store credentials in .env file.
2. **Mid-term**:
    - Transition to **API key validation** for developers and testers.
    - Log requests for monitoring usage.
3. **Long-term**:
    - Integrate **JWT authentication** for scalable deployments.
    - Support **OAuth2 login** for enterprise identity management.
    - Implement **role-based access control (RBAC)**:
        - *Admin*: Manage users, view logs.
        - *Developer*: Test and integrate APIs.
        - *User*: Access requirement analysis and code generation.

9. User Interface

9.1 Design Philosophy

The user interface (UI) is designed to be **minimalist, intuitive, and accessible**. Built using **Gradio Blocks**, it enables both technical and non-technical users to interact with advanced AI models without requiring programming expertise. The UI balances functionality and simplicity, ensuring smooth navigation between features while maintaining a professional look suitable for academic and enterprise use.

9.2 Layout and Navigation

The interface is organized into two primary **tabs**, accessible via a top-level navigation bar:

1. **Code Analysis Tab**
    - **Inputs:**
        - File upload widget (.pdf) for requirement documents.
        - Multiline text area for manually entering requirements.
        - "Analyze" button to trigger requirement extraction.
    - **Outputs:**
        - A large text panel displaying **categorized requirements**:
            - Functional requirements
            - Non-functional requirements
            - Technical specifications

2. **Code Generation Tab**
    - **Inputs:**
        - Multiline text box for describing required functionality.
        - Dropdown menu to select the target programming language (Python, Java, JavaScript, C++, etc.).
        - "Generate Code" button to request AI-generated implementation.
    - **Outputs:**
        - A scrollable text panel displaying the AI-generated code snippet.
        - Supports co**py-paste ready** output for immediate integration into IDEs.

9.3 Accessibility and Usability Features

- **Web-based Deployment**: Runs entirely in a browser, no installation required for end users.
- **File Upload Support**: Direct PDF upload for automatic requirement extraction.
- **Real-Time Interaction**: AI responses generated dynamically, displayed within seconds.
- **Copy-Friendly Outputs**: Generated code and analysis can be easily copied into external tools.
- **Responsive Design**: Gradio ensures the interface adapts to various screen sizes (desktop, laptop, tablet).

9.4 Enhancements for User Experience (Planned)

- **Dark/Light Mode Toggle** for improved readability.
- **Syntax Highlighting** for generated code outputs.
- **Export Options**: Ability to download results as **PDF, TXT, or Markdown**.
- **Interactive Editing**: Allow users to refine AI outputs by adding feedback loops.
- **Role-Based Dashboards**: Separate views for students, developers, and administrators.

10. Testing

10.1 Introduction

Testing is a critical component in ensuring the reliability, accuracy, and usability of the **AI Code Analysis & Generator** system. Given that the application integrates **Hugging Face models, PDF parsing (PyPDF2), and a Gradio-based user interface**, testing was carried out at multiple levels: **unit testing, functional/API testing, integration testing, and manual exploratory testing**.

10.2 Unit Testing

Unit tests were implemented for core functional modules to validate correctness under controlled inputs:

- **Model Response Function (generate_response)**
    - Ensures the model returns a valid string when provided with a prompt.
    - Checks that responses do not include repeated prompts or empty outputs.
- **PDF Extraction (extract_text_from_pdf)**
    - Verifies successful extraction from valid PDFs.
    - Tests handling of empty, malformed, or scanned (non-text) PDFs.
- **Requirement Analysis (requirement_analysis)**
    - Confirms that outputs contain structured categories: *Functional Requirements*, *Non-Functional Requirements*, *Technical Specifications*.
- **Code Generation (code_generation)**
    - Validates output format as a plausible code block.
    - Ensures language selection influences the style of generated code.

10.3 Functional & API Testing

Although the system is accessed via **Gradio**, its core logic can be abstracted as API functions. Functional tests were performed as if calling API endpoints:

- /analyze-requirements
    - Input: Sample requirements text.
    - Output: JSON structure with categorized requirements.
- /generate-code
    - Input: Natural language requirement + programming language.
    - Output: Valid code snippet in selected language.

10.4 Integration Testing

Integration testing validated the interaction between:

- **Gradio Frontend** (user inputs, file uploads, UI rendering).
- **Backend Logic** (Hugging Face model inference, PDF parsing).
- **Outputs** (display of results in text boxes).

Key tests included:

- Uploading PDF + text input simultaneously.
- Switching between tabs without restarting the session.
- Generating large code outputs (>200 lines).

10.5 Manual Exploratory Testing

Manual testing was conducted to simulate real-world user workflows:

- Uploading policy requirement documents.
- Requesting multi-language code outputs (Python, Java, C++).
- Testing random or ambiguous prompts (e.g., "build something secure").

10.6 Edge Case Testing

Special scenarios were tested to evaluate system robustness:

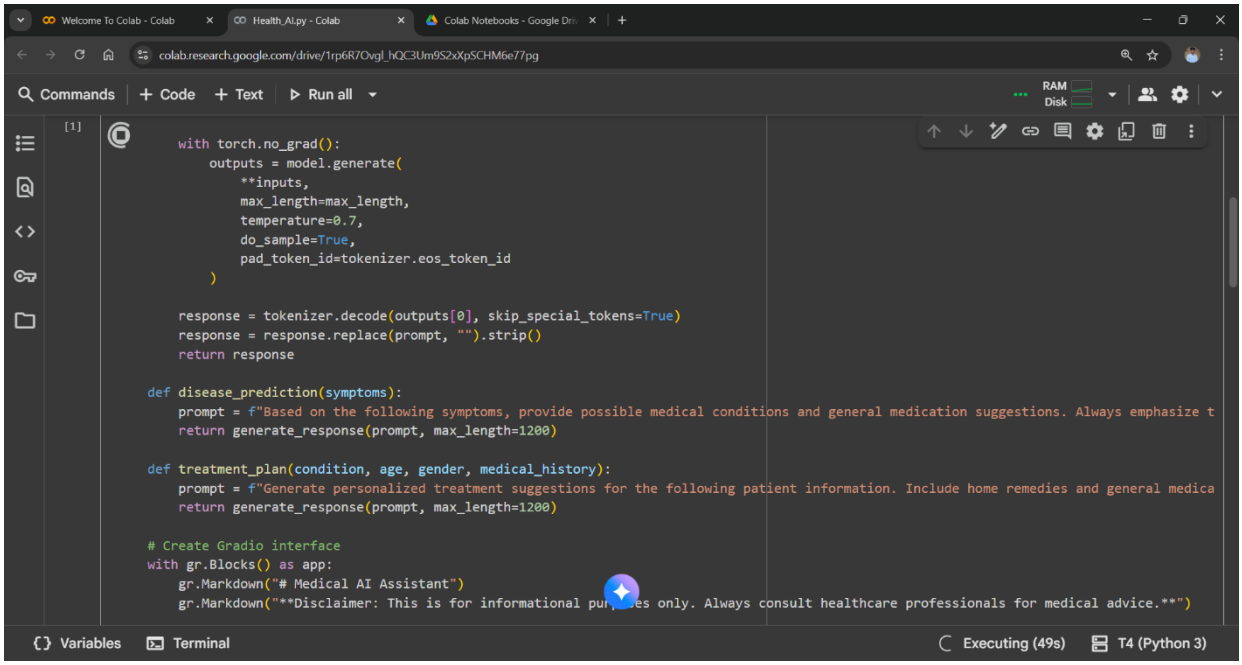
Test Case	Expected Behavior	Result
Empty input	Return an error message or no output	Handled correctly
Large PDF (>50 pages)	Extract and analyze text	Slower, but functional
Non-text PDF (scanned)	Return error	Error message displayed
Unsupported language (e.g., COBOL)	Fallback to generic output	Returned generic code
Invalid file format (e.g., .jpg)	Reject input	Correctly rejected

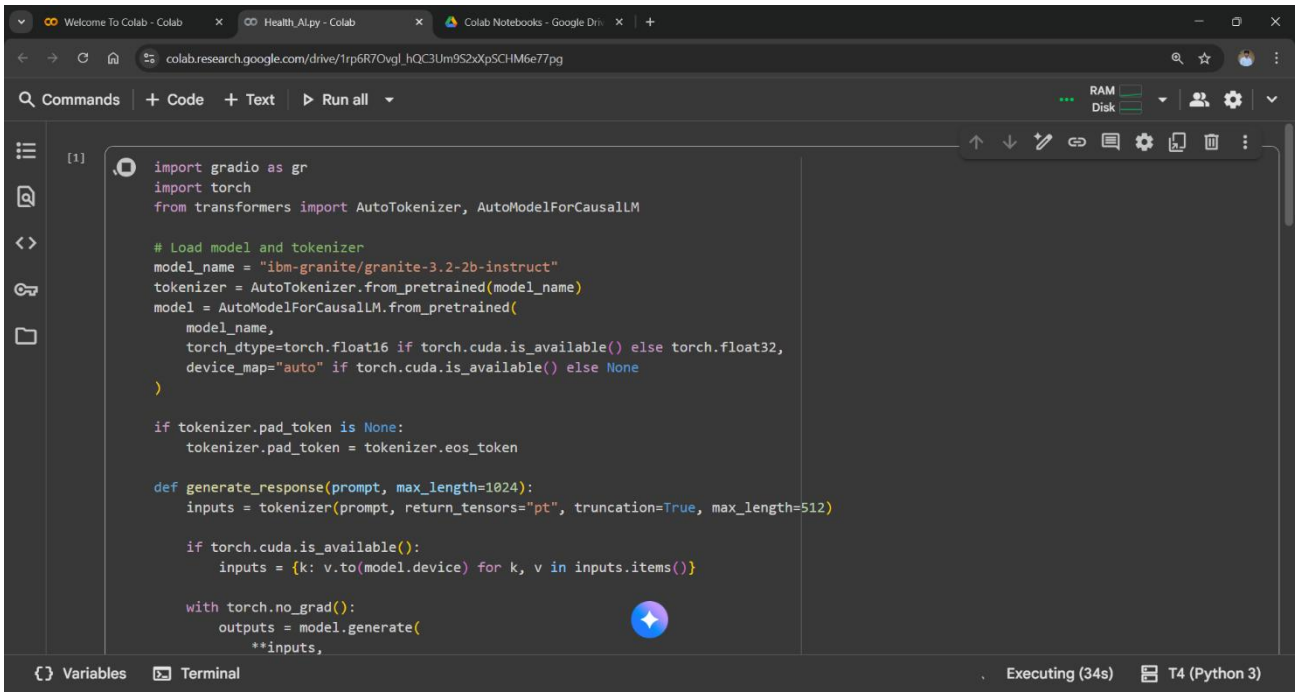
10.7 Evaluation Metrics

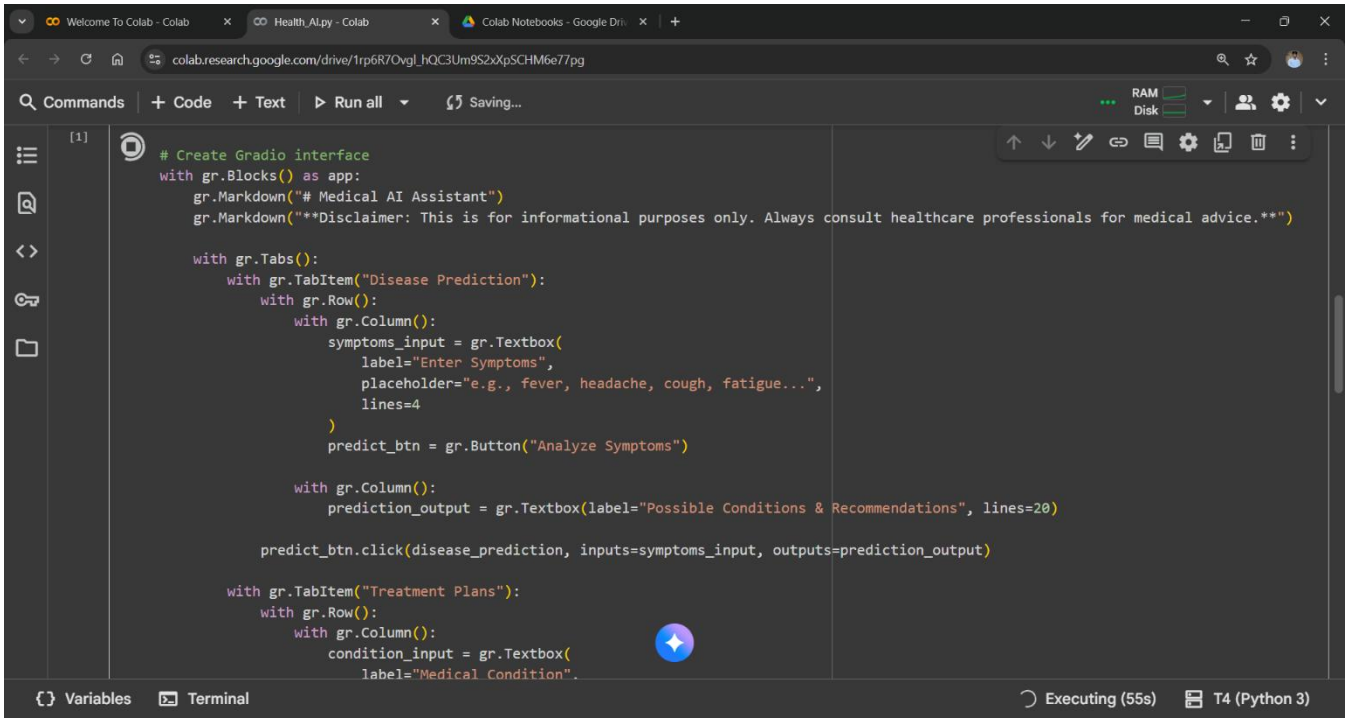
To assess effectiveness, the following metrics were applied:

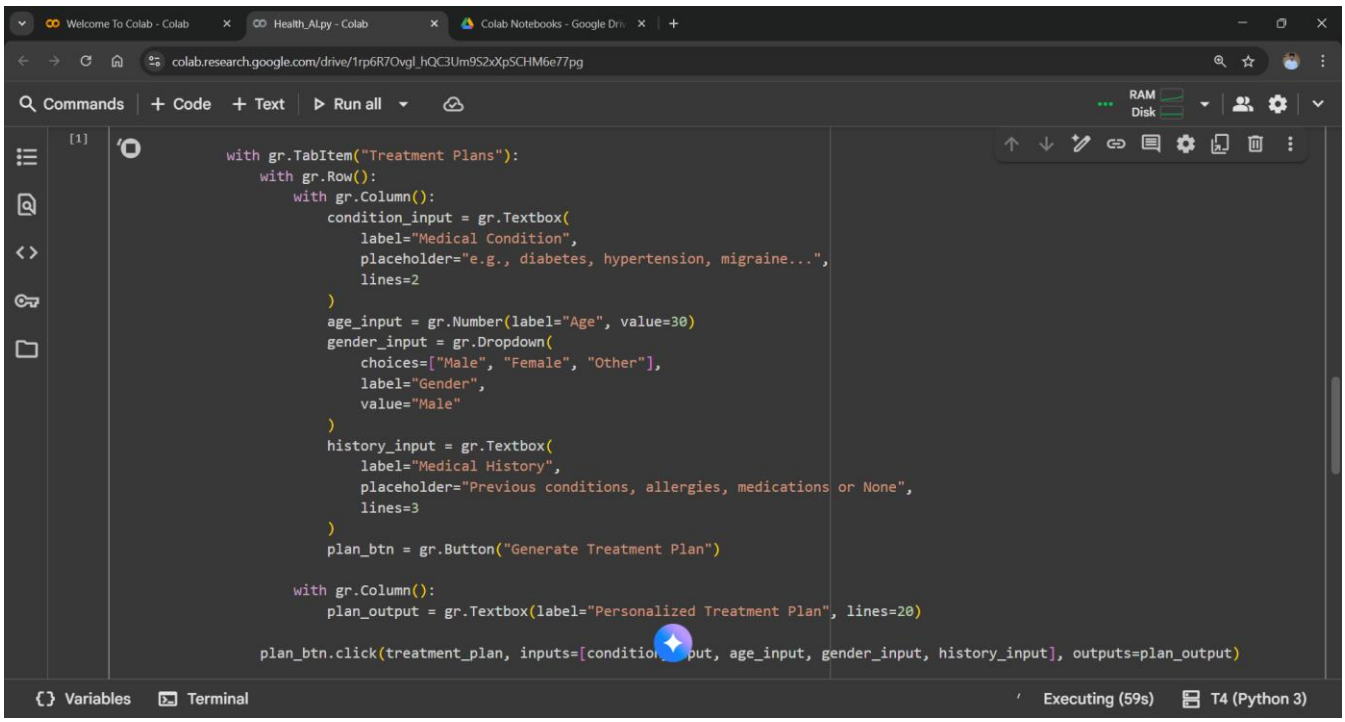
- **Accuracy of Requirement Categorization** – Human evaluators checked alignment of AI analysis with true requirement types.
- **Code Plausibility** – Developers validated whether generated code was syntactically correct and relevant.
- **Usability Feedback** – Test users rated interface clarity and ease of navigation.

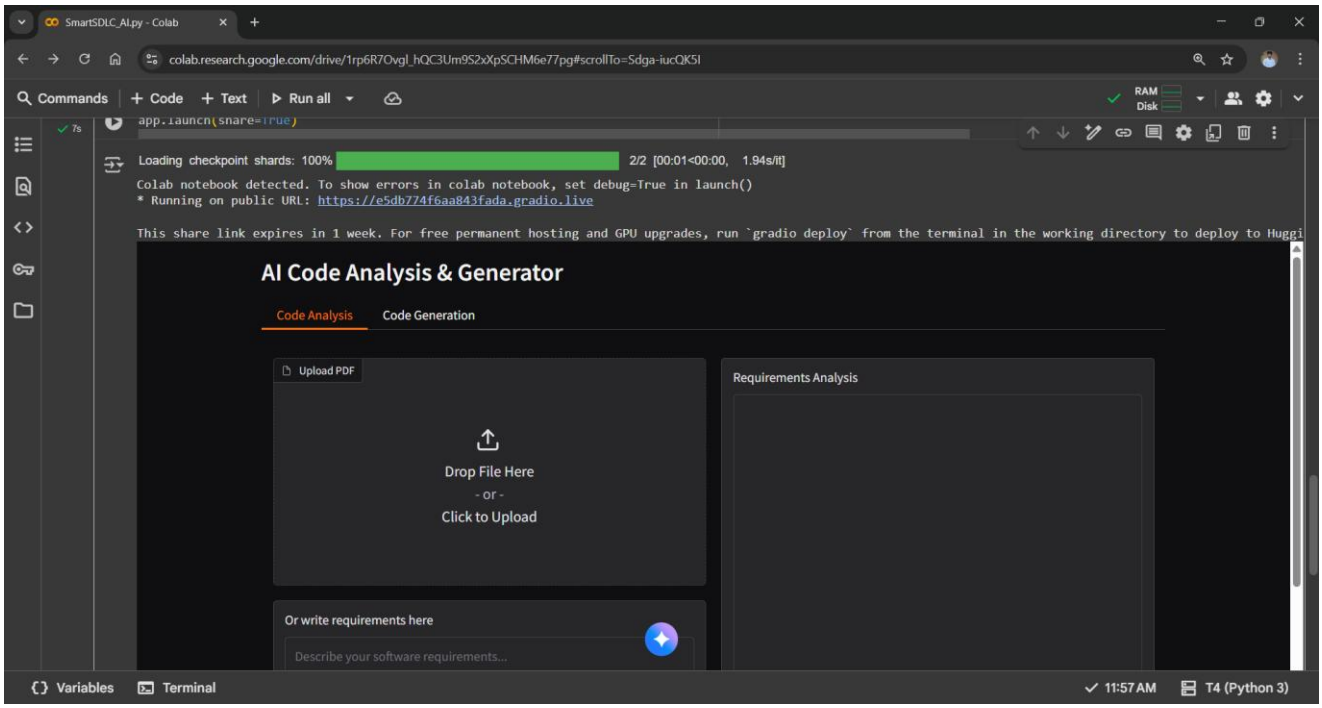
11. Screenshots

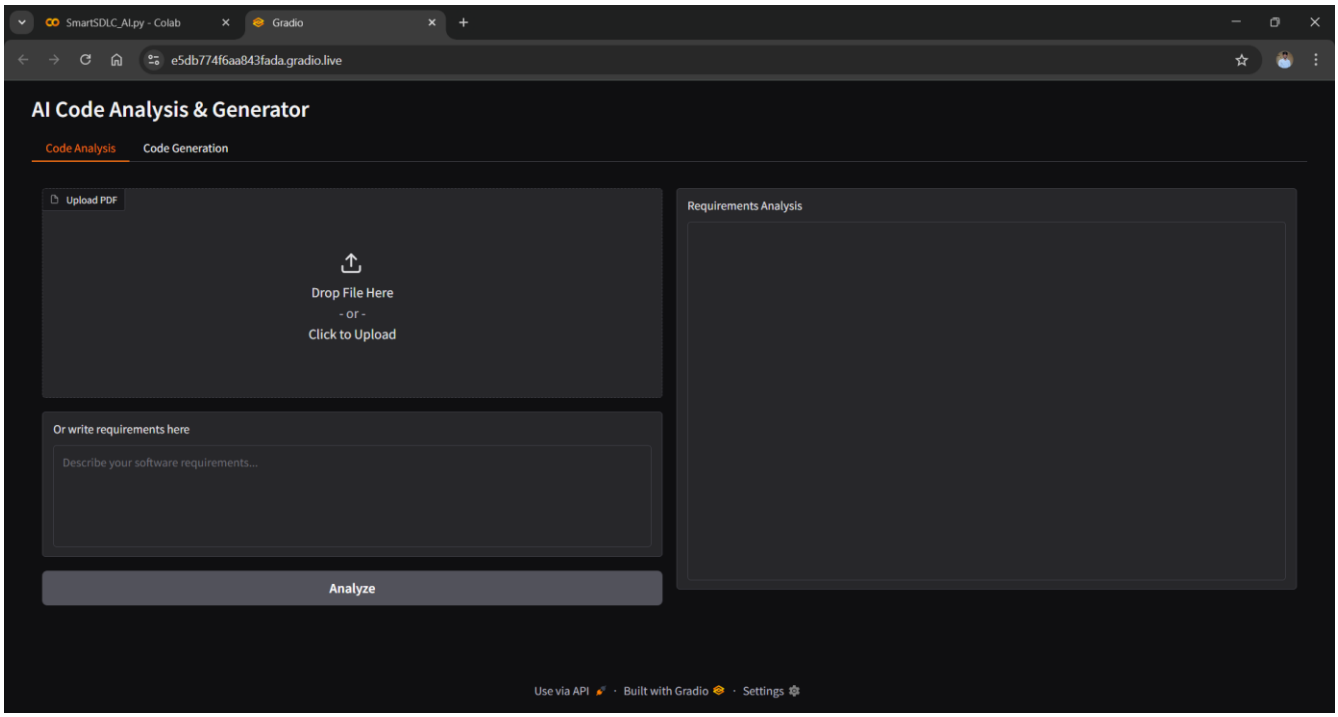


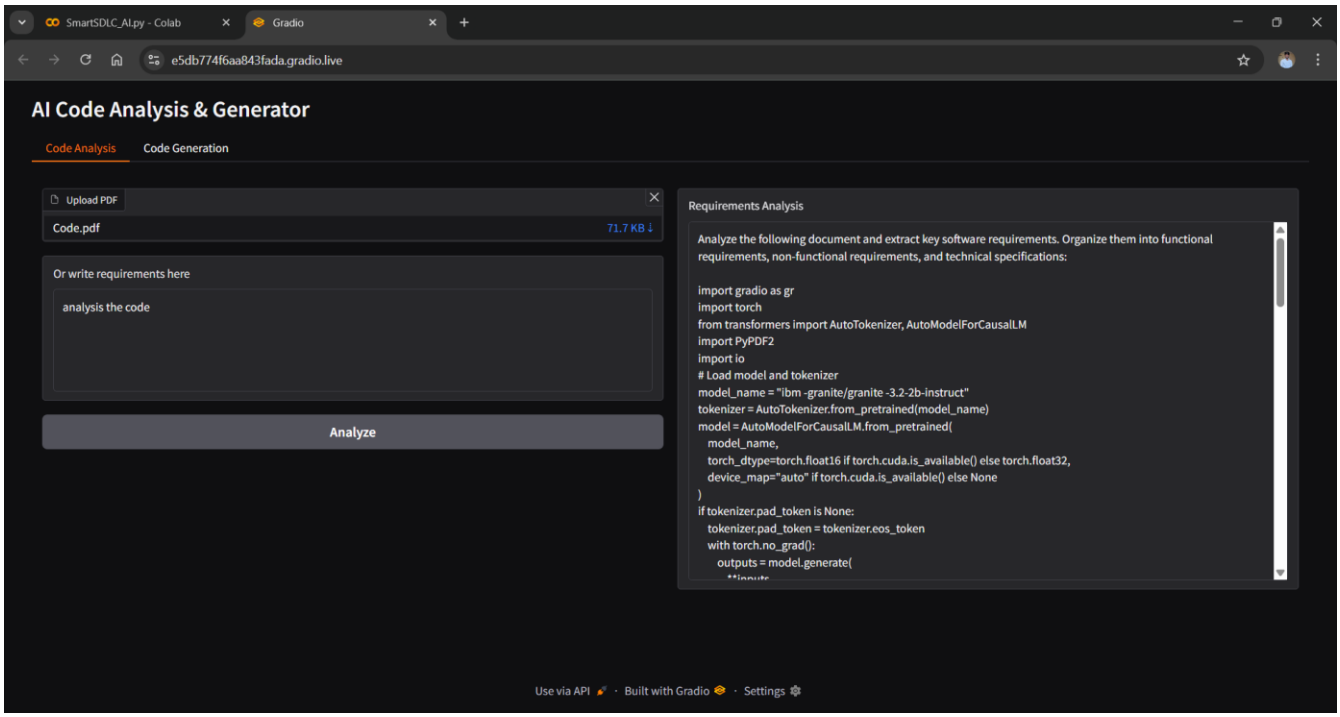



Welcome To Colab - Colab
Health_AI.py - Colab
Colab Notebooks - Google Dri
colab.research.google.com/drive/1rp6R7Ovgl_hQC3Um9S2xXpSCHM6e77pg
Commands
+ Code
+ Text
Run all
Saving...
RAM
Disk
[1]
Create Gradio interface
with gr.Blocks() as app:
gr.Markdown("# Medical AI Assistant")
gr.Markdown("**Disclaimer: This is for informational purposes only. Always consult healthcare professionals for medical advice.**")
with gr.Tabs():
with gr.TabItem("Disease Prediction"):
with gr.Row():
with gr.Column():
symptoms_input = gr.Textbox(
label="Enter Symptoms",
placeholder="e.g., fever, headache, cough, fatigue...",
lines=4
)
predict_btn = gr.Button("Analyze Symptoms")
with gr.Column():
prediction_output = gr.Textbox(label="Possible Conditions & Recommendations", lines=20)
predict_btn.click(disease_prediction, inputs=symptoms_input, outputs=prediction_output)
with gr.TabItem("Treatment Plans"):
with gr.Row():
with gr.Column():
condition_input = gr.Textbox(
label="Medical Condition",
Variables
Terminal
Executing (55s)
T4 (Python 3)


Welcome To Colab - Colab
Health_AI.py - Colab
Colab Notebooks - Google Dri
colab.research.google.com/drive/1rp6R7Ovgl_hQC3Um9S2xXpSCHM6e77pg
Commands
+ Code
+ Text
Run all
RAM
Disk
[1]
with gr.TabItem("Treatment Plans"):
with gr.Row():
with gr.Column():
condition_input = gr.Textbox(
label="Medical Condition",
placeholder="e.g., diabetes, hypertension, migraine...",
lines=2
)
age_input = gr.Number(label="Age", value=30)
gender_input = gr.Dropdown(
choices=["Male", "Female", "Other"],
label="Gender",
value="Male"
)
history_input = gr.Textbox(
label="Medical History",
placeholder="Previous conditions, allergies, medications or None",
lines=3
)
plan_btn = gr.Button("Generate Treatment Plan")
with gr.Column():
plan_output = gr.Textbox(label="Personalized Treatment Plan", lines=20)
plan_btn.click(treatment_plan, inputs=[condition
put, age_input, gender_input, history_input], outputs=plan_output)
Variables
Terminal
Executing (59s)
T4 (Python 3)


SmartSDLC_AI.py - Colab
colab.research.google.com/drive/1rp6R7Ovgl_hQC3Um9S2xXpSCHM6e77pg#scrollTo=Sdga-iucQK5I
Commands
+ Code
+ Text
Run all
RAM
Disk
app.launch(share=True)
Loading checkpoint shards: 100%
2/2 [00:01<00:00, 1.94s/it]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://e5db774f6aa843fada.gradio.live
This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Huggi
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...
Variables
Terminal
11:57 AM
T4 (Python 3)


SmartSDLC_AI.py - Colab
Gradio
e5db774f6aa843fada.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


SmartSDLC_AI.py - Colab
Gradio
e5db774f6aa843fada.gradio.live
AI Code Analysis & Generator
Code Analysis
Code Generation
Upload PDF
Code.pdf
71.7 KB
Or write requirements here
analysis the code
Analyze
Requirements Analysis
Analyze the following document and extract key software requirements. Organize them into functional requirements, non-functional requirements, and technical specifications:
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
with torch.no_grad():
outputs = model.generate(
Use via API
Built with Gradio
Settings

12. Known Issues

Despite successful implementation and testing, the **AI Code Analysis & Generator** application currently has several limitations that must be acknowledged for transparency and to guide future improvements.

12.1 Model Limitations

- **Context Length Constraint**: The IBM Granite model, as integrated via Hugging Face Transformers, has a maximum token limit. Very large documents or overly long prompts may be truncated, leading to incomplete analysis or code generation.
- **Generic Outputs**: In some cases, especially with ambiguous prompts, the model produces generic or overly broad requirements/code instead of highly specific solutions.
- **Code Validity**: While generated code is usually syntactically correct, it is not guaranteed to be functionally complete or optimized for real-world deployment.

12.2 PDF Processing Issues

- **Scanned/Non-Text PDFs**: Since the application relies on PyPDF2, image-based PDFs (scans or photocopies) cannot be processed. An OCR layer would be required for such cases.
- **Complex Formatting Loss**: Tables, diagrams, and special formatting within PDFs are not always preserved during text extraction, which may affect requirement analysis accuracy.

12.3 Performance Constraints

- **GPU Dependency**: On CPU-only environments, response latency increases significantly, especially for longer prompts.
- **Memory Usage**: Running large models such as ibm-granite/granite-3.2-2b-instruct can cause high RAM/VRAM consumption, occasionally leading to out-of-memory errors on limited hardware.

12.4 Usability Gaps

- **Lack of Authentication**: The current version runs without access control, making it unsuitable for production environments where sensitive documents may be uploaded.
- **No Syntax Highlighting**: Generated code is presented as plain text, making it harder for users to review large outputs.
- **Limited Export Options**: Results cannot yet be directly exported as PDF, DOCX, or Markdown reports, reducing convenience for academic and enterprise users.

12.5 Integration Challenges

- **No API Exposure**: At present, the system is only accessible through the Gradio interface. It cannot yet be consumed programmatically by third-party applications.
- **Limited Multilingual Support**: Requirement analysis and code generation are optimized for English input. Non-English texts may result in inconsistent outputs.

13. Future Enhancements

The current version of the **AI Code Analysis & Generator** serves as a functional prototype for requirement analysis and automated code generation. However, to make the system more robust, scalable, and industry-ready, several enhancements are planned. These improvements focus on addressing known limitations, expanding capabilities, and ensuring usability in academic and enterprise contexts.

13.1 Model and NLP Improvements

- **Extended Context Support**: Integrate models with longer token windows (e.g., 32k context) to handle large documents and complex prompts.
- **Domain-Specific Fine-Tuning**: Fine-tune models on software engineering datasets to improve precision in requirement categorization and technical specification extraction.
- **Multilingual Support**: Expand input/output capabilities to multiple languages, enabling global usability.

13.2 Document Processing Enhancements

- **OCR Integration**: Add an OCR module (e.g., Tesseract or AWS Textract) to process scanned/image-based PDFs.
- **Advanced Formatting Preservation**: Enhance parsing to better retain tables, lists, and diagrams during requirement analysis.
- **Batch Document Support**: Allow multiple PDF uploads and parallel analysis for large-scale use cases.

13.3 Performance and Scalability

- **Model Optimization**: Implement quantization, pruning, and knowledge distillation to reduce memory usage and improve inference speed.
- **Cloud Deployment**: Deploy on scalable platforms (IBM Cloud, AWS, or GCP) for broader access and real-time collaboration.
- **Asynchronous Processing**: Use task queues (Celery, Redis) for handling long-running tasks efficiently.

13.4 Security and Authentication

- **User Authentication**: Implement robust authentication using JWT or OAuth2 for secure access.
- **Role-Based Access Control (RBAC)**: Differentiate permissions for Admins, Developers, and General Users.
- **Secure File Handling**: Add encryption for uploaded PDFs to ensure confidentiality of sensitive requirements.

13.5 User Interface Enhancements

- **Syntax Highlighting for Code**: Present generated code with color-coded formatting for better readability.
- **Export Options**: Enable direct export of analysis and code outputs in PDF, DOCX, and Markdown formats.
- **Interactive Feedback Loop**: Allow users to refine outputs by marking "useful," "irrelevant," or suggesting corrections, enabling adaptive improvements.
- **Dark/Light Themes**: Add theme customization to improve accessibility.

13.6 API and Integration Capabilities

- **REST API Exposure**: Extend backend functions as FastAPI endpoints for programmatic integration with external systems.
- **IDE Plugins**: Develop extensions for Visual Studio Code and IntelliJ, bringing requirement analysis and code generation directly into developer workflows.
- **CI/CD Integration**: Automate requirement validation and code generation pipelines in software development lifecycles.

13.7 Long-Term Research Directions

- **Automated Test Case Generation**: Extend code generation module to also produce corresponding unit tests.
- **Requirement-to-Architecture Mapping**: Evolve analysis into system-level design recommendations (e.g., UML diagrams).
- **Self-Learning Feedback Loop**: Implement reinforcement learning from user feedback to continuously refine model accuracy.