**SmartSDLC AI Enhanced Software Development Lifecycle**

1. **Introduction**

• **Project title** : SmartSDLC AI Assistant

• **Team member** : C.Ranjith

• **Team member** : A.Rajesh Kumar

• **Team membe**r : C.Raj Kumar

• **Team membe**r :R.Sanjay

1. **Project overview**

• **Purpose:**

 The purpose of this program is to provide an AI-powered tool that assists software developers and analysts in streamlining the software development process. It allows users to upload requirement documents (PDFs) or input raw requirement text, and then automatically analyzes and categorizes them into functional requirements, non-functional requirements, and technical specifications. Additionally, the program can generate ready-to-use source code snippets in multiple programming languages based on user-provided requirements. By combining requirement analysis and code generation in one platform with an easy-to-use Gradio interface, the program reduces manual effort, speeds up development, and improves clarity in software projects

• **Fearture:**

1. PDF Requirement Extraction – Upload PDF documents and automatically extract requirement text.

2. Requirement Analysis – Categorizes input into functional, non-functional, and technical specifications.

3. AI-Powered Code Generation – Generates code snippets in multiple languages (Python, Java, C++, JavaScript, etc.) based on requirements.

4. Dual Input Support – Accepts both direct text input and PDF uploads for flexibility.

5. Interactive UI with Gradio – Easy-to-use tab-based interface for switching between analysis and code generation.

6. Customizable Code Output – Users can choose the target programming language for generated code.

7. GPU/CPU Support – Optimized to run on both GPU (if available) and CPU environments.

8. Error Handling – Provides meaningful error messages when PDFs cannot be read.

9. Real-time Results – Instant display of requirement analysis and generated code within the interface.

1. Shareable Application – Can be shared via public Gradio link for collaborative use.
2. **Architecture**

1.Input Layer

PDF Upload Module → Uses PyPDF2 to extract raw text from uploaded PDF requirement documents.

Text Input Module → Allows users to manually type or paste software requirements.

2. Processing Layer

Tokenizer & Model Loader → Loads the IBM Granite AI model (AutoTokenizer, AutoModelForCausalLM) for NLP tasks.

Requirement Analysis Module

Takes extracted text/prompt.

Constructs an analysis prompt for the model.

AI organizes requirements into functional, non-functional, technical categories.

Code Generation Module

Takes user requirement + selected language.

AI generates code snippet accordingly.

3. Model Execution Layer

PyTorch Runtime → Executes the Granite AI model either on GPU (CUDA) or CPU depending on availability.

Response Generator → Post-processes AI output (removes prompt text, formats response).

4. Output Layer

Requirements Analysis Output → Displays organized requirements in a textbox.

Code Generation Output → Displays generated code snippet in the chosen programming language.

5. User Interface Layer (Gradio)

Tabs Layout

Code Analysis Tab → For uploading PDFs / entering requirements and viewing categorized analysis.

Code Generation Tab → For requirement input + language selection + generated code display.

Interactive Components → File upload, textbox, dropdown, and buttons connected to backend functions.

1. **Setup Instructions**

1. System Requirements

OS: Windows, macOS, or Linux

Python: 3.9 or later

Hardware:

CPU (works fine)

GPU (recommended for faster inference, supports CUDA)

Internet connection (required to download model + run Gradio app)

2. Install Required Packages

Open a terminal (or Google Colab cell) and run:

pip install torch transformers gradio -q

torch → Deep learning framework for running models.

transformers → Hugging Face library to load IBM

Granite model.

gradio → To create the web-based interface.

**5. Folder Structure**

app.py → Single entry point for running the Gradio app.

modules/ → Keeps core logic (analysis, code generation, PDF handling) separate and reusable.

tests/ → Ensures correctness of functions.

docs/ → For proper documentation (can be used in reports/presentations).

assets/ → Sample files & screenshots.

**6. Running the Application**

5. User Interface Layer (Gradio)

Tabs Layout

Code Analysis Tab → For uploading PDFs / entering requirements and viewing categorized analysis.

Code Generation Tab → For requirement input + language selection + generated code display.

Interactive Components → File upload, textbox, dropdown, and buttons connected to backend functions.

3. AI Model Layer (Core Inference)

IBM Granite Model (granite-3.2-2b-instruct) loaded via transformers.

Runs on GPU (float16) if available, else CPU (float32).

Uses generate() to create responses with controlled sampling (temperature=0.7).

1. **API Documentation**

1. requirement\_analysis(pdf\_file, prompt\_text)

Description

Analyzes software requirements provided via PDF document or manual text input, and organizes them into categories:

Functional Requirements

Non-Functional Requirements

Technical Specifications

Parameters

Name Type Description

pdf\_file file / None PDF file containing requirements. If None, prompt\_text is used.

prompt\_text str Raw requirement text input. Used only if pdf\_file is not provided.

2. code\_generation(prompt, language)

Description

Generates source code in a specified programming language based on the provided requirement description.

Parameters

Name Type Description

prompt str Requirement or feature description for which code is needed.

language str Target programming language (Python, Java, JavaScript, C++, etc.).

3. extract\_text\_from\_pdf(pdf\_file)

Description

Extracts text from a given PDF file.

Parameters

Name Type Description

pdf\_file file PDF file to extract text from.

4. generate\_response(prompt, max\_length=1024)

Description

Core function that interacts with the Granite AI model to generate responses based on prompts.

Parameters

Name Type Default Description

prompt str required Input text for the AI model.

max\_length int 1024 Maximum token length for generated response.

1. **Authentication**

For personal use / testing → Gradio’s auth is enough.

For team use → Use a custom authentication layer (tokens, JWT).

For production / enterprise → Deploy behind a secure API gateway (NGINX, FastAPI with OAuth2, etc.

1. **User Interface**

1. Main Layout

Header → Title: “AI Code Analysis & Generator” (Markdown heading).

Two Tabs:

1. Code Analysis

2. Code Generation

2. Code Analysis Tab

Left Column (Inputs):

File Upload → Upload a PDF document containing requirements (.pdf).

Textbox → Enter requirements manually if no PDF is available.

Button: “Analyze” → Runs requirement analysis.

Right Column (Output):

Textbox (Large) → Displays analyzed requirements organized into:

Functional requirements

Non-functional requirements

Technical specifications

3. Code Generation Tab

Left Column (Inputs):

Textbox → Write the requirement/feature you want code for.

Dropdown Menu → Choose target programming language (Python, Java, C++, JavaScript, C#, PHP, Go, Rust).

Button: “Generate Code” → Runs AI-based code generation.

Right Column (Output):

Textbox (Large) → Displays generated code snippet in the chosen language.

4. Interaction Flow

1. User either uploads a PDF or types requirements.

2. Clicks Analyze → AI organizes requirements.

3. Switch to Code Generation tab.

4. Enters requirement + selects language.

5. Clicks Generate Code → AI outputs code.

5. UI Characteristics

Tabbed Layout → Clear separation between analysis and generation tasks.

Responsive Design → Works in browser, no installation needed.

Shareable Link → If share=True, UI can be accessed by others remotely.

1. **Testing**

1. extract\_text\_from\_pdf(pdf\_file)

Input: valid PDF → should return correct extracted text.

Input: corrupted/empty PDF → should return error message.

2. generate\_response(prompt, max\_length)

Input: short prompt → should return valid AI-generated text.

Input: very long prompt → should truncate correctly without crashing.

3. requirement\_analysis(pdf\_file, prompt\_text)

Input: sample text → should categorize into functional, non-functional, technical.

Input: only PDF → should work without prompt\_text.

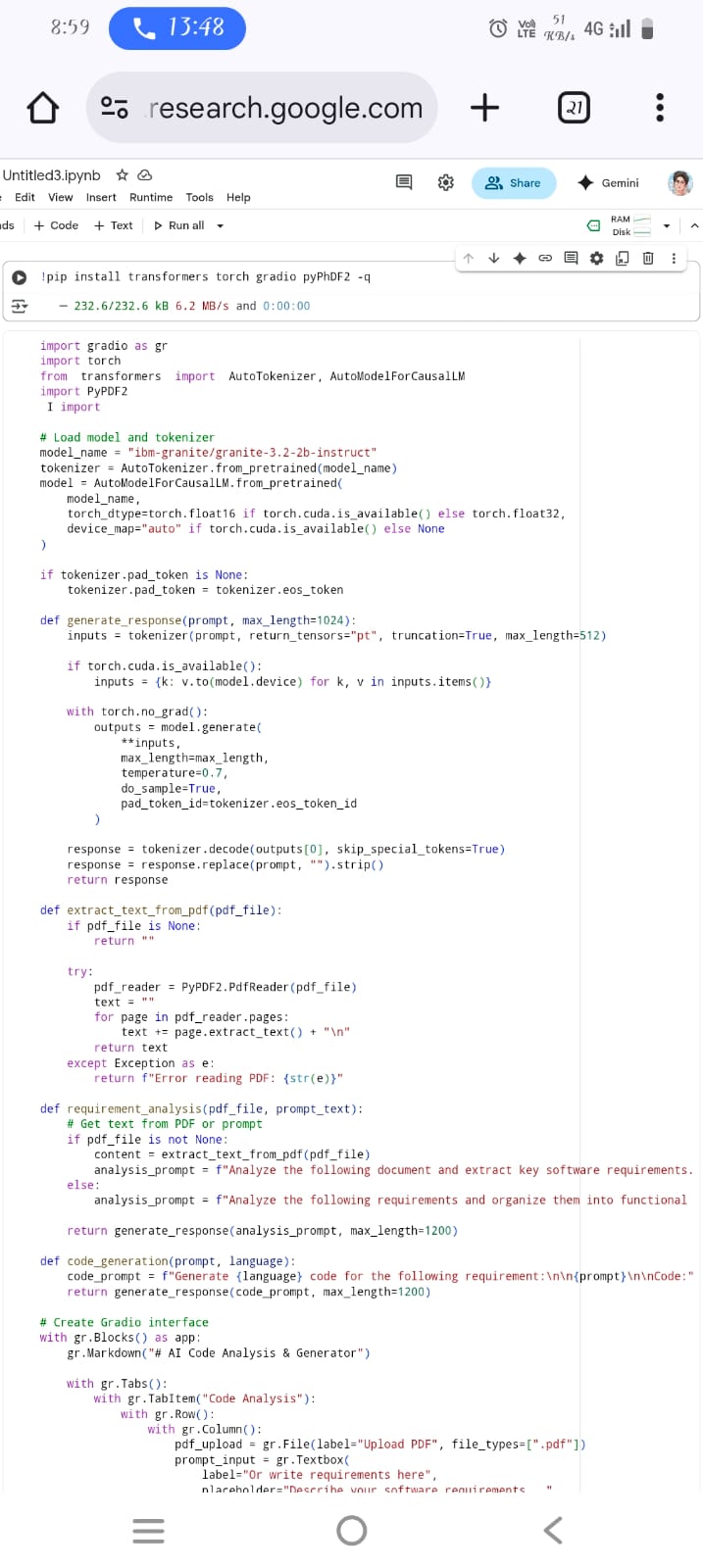
4. code\_generation(prompt, language)

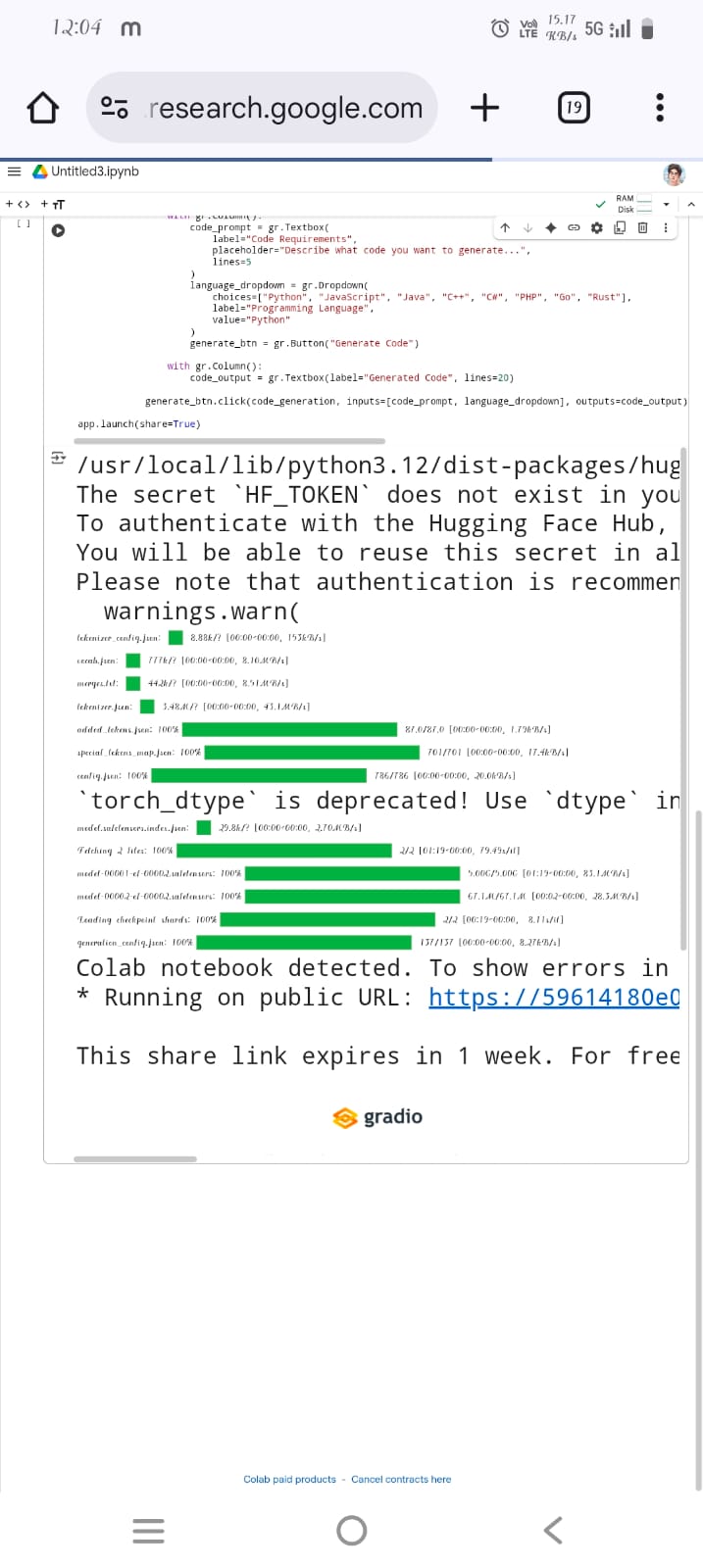
Input: valid prompt + “Python” → should return Python code.

Input: unsupported language → should return an error or fallback.

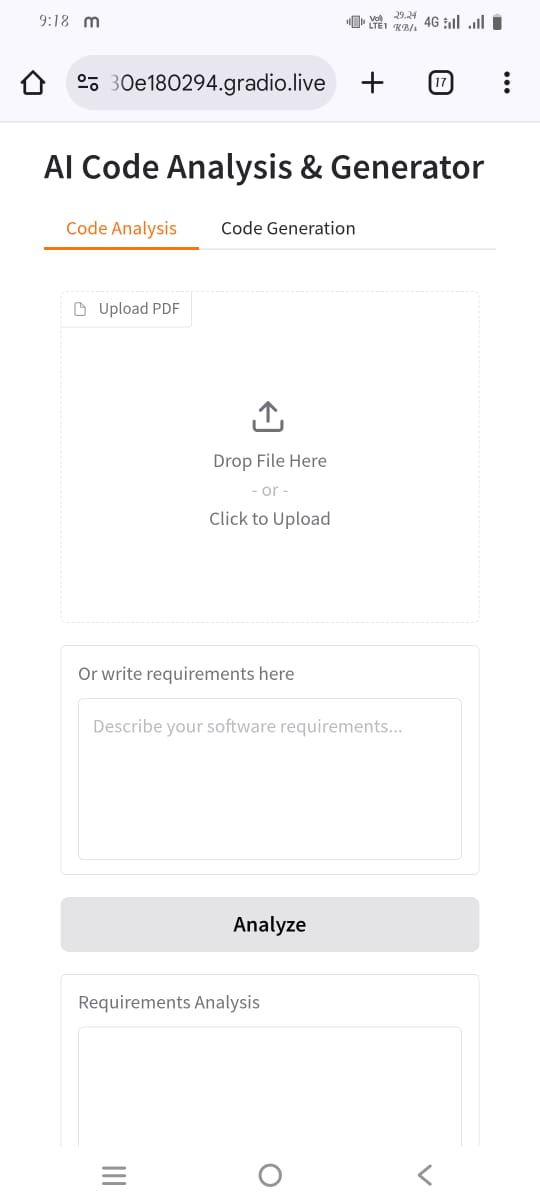
1. **screen shots**

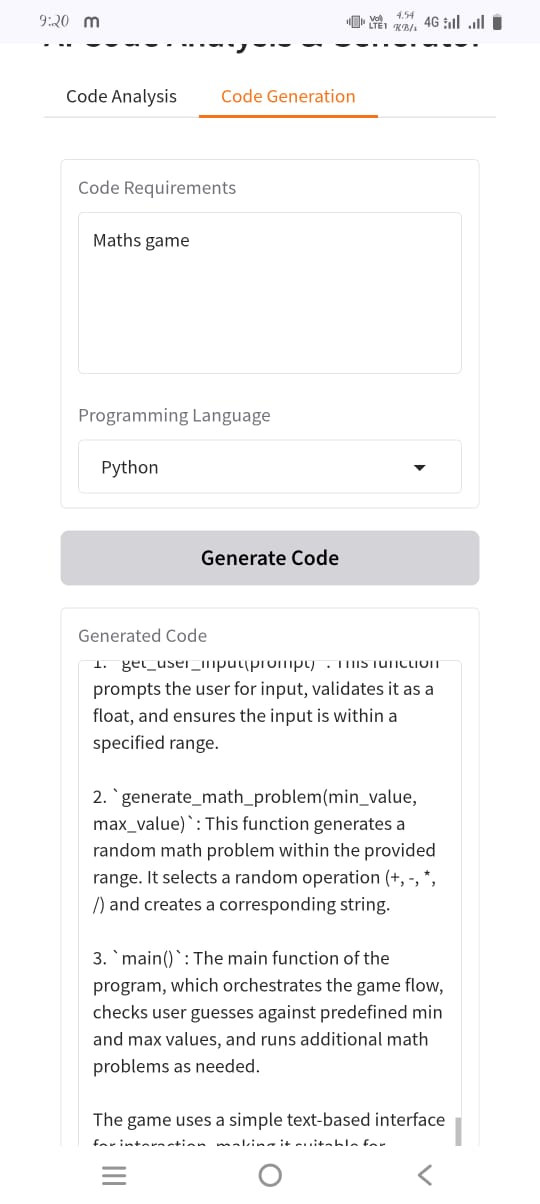
**Input**

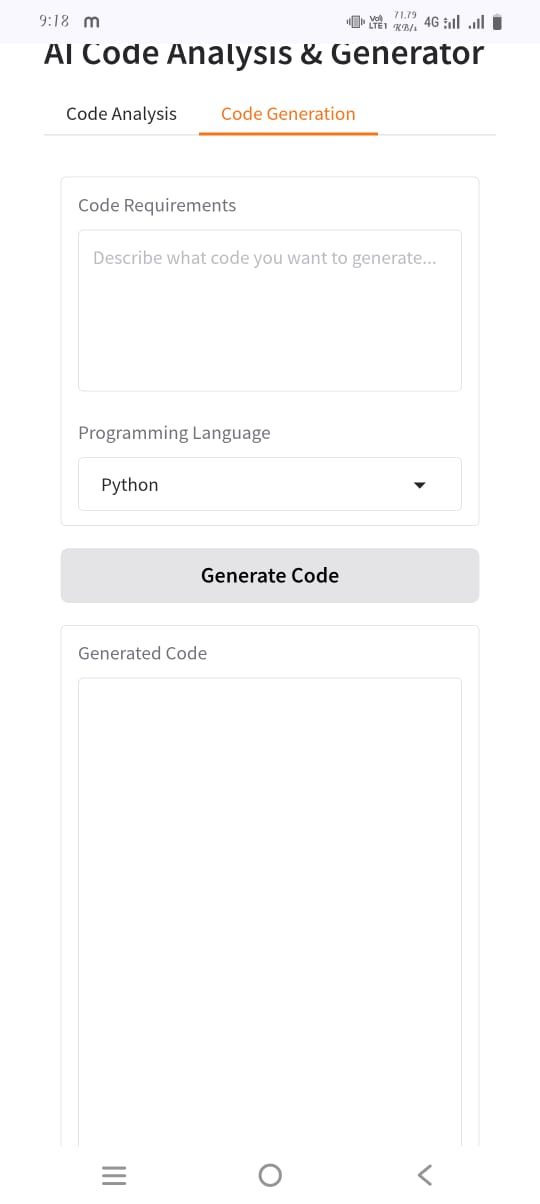
****

****

**Output**

****

****

****

1. **Known Issues**

* PDF Extraction Limitations
* AI Model Hallucinations
* Performance Constraints
* Limited Error Feedback
* Authentication Weakness
* UI Limitations
* No Persistent Storage
* Language & Domain Bias

1. **Future enhancement**

* Advanced PDF Support
* Improved Code Generation
* Requirement Traceability
* Enhanced User Interface
* Collaboration Features
* Authentication & Security
* Model Enhancements
* Deployment Improvements
* Performance Optimization
* Integration with Development Tools