SmartSDLC – AI-Enhanced Software Development Lifecycle

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

BACHELOR OF COMPUTER SCIENCE

Submitted By

(Team-Id: NM2025TMID04709)

Name	**Reg. No**
SHYAM D	222305227
SABESH S	222305218
SENTHILKUMAR R	222305226
GOWTHAM A	222305207

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 **“SmartSDLC – AI-Enhanced Software Development Lifecycle”** is the Bonafide record of work done by **(Shyam - Sabesh - Senthilkumar - Gowtham)** 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	**Staff-in-Charge**
Dr. J. KARTHIKEYAN	Mr. K. DINESHKUMAR

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

Internal Examiner	**External Examiner**

SmartSDLC – AI-Enhanced Software Development Lifecycle

Table of Contents

1. Introduction

Project Title: SmartSDLC – AI-Enhanced Software Development Lifecycle

Team Members:

S. No.	Name	Reg. No
1	SHYAM D	222305227
2	SABESH S	222305218
3	SENTHILKUMAR R	222305226
4	GOWTHAM A	222305207

Program Background:

This project was undertaken as part of the **IBM Naan Mudhalvan Smart Internz Program**, which provides students and early professionals with an opportunity to work on cutting-edge technologies, including **IBM Watsonx**, **Granite LLMs**, and **AI-driven automation**.

The *Smart SDLC AI* project is aimed at transforming the traditional Software Development Lifecycle (SDLC) by integrating **Generative AI, forecasting models, anomaly detection, and multimodal interaction capabilities** into a single intelligent assistant. This AI-powered platform assists developers, project managers, and stakeholders in accelerating software development, improving requirement clarity, generating documentation, and monitoring project KPIs in real time.

2. Project Overview

2.1 Purpose of the Project

The **Smart SDLC AI** project has been designed to **intelligently augment and streamline the Software Development Lifecycle (SDLC)** using Artificial Intelligence (AI) and Machine Learning (ML). Traditional SDLC processes often involve manual effort, ambiguity in requirements, delays in project forecasting, and difficulty in monitoring key performance indicators (KPIs).

The primary purpose of Smart SDLC AI is to:

- Enhance **efficiency** by reducing repetitive and time-consuming manual tasks.
- Provide **data-driven decision support** for developers, project managers, and stakeholders.
- Enable **intelligent insights** such as summarization, forecasting, anomaly detection, and KPI monitoring.
- Support **multimodal inputs** (text, PDFs, CSVs) for flexible document and data processing.
- Create a **unified AI-powered workspace** where users can chat with an assistant, receive actionable insights, and generate structured outputs such as reports.

By embedding **IBM Watsonx Granite LLMs** into the workflow, Smart SDLC AI ensures that software projects become more transparent, predictable, and adaptive.

2.2 Key Features

The system integrates a wide range of **AI-driven functionalities**, making it a versatile assistant for the SDLC process:

1. **Conversational AI Chatbot**
    - Provides a natural language interface where users can ask questions, clarify requirements, and receive guidance.
    - Built on IBM Watsonx Granite for contextual and human-like responses.
2. **Document & Policy Summarization**
    - Converts lengthy requirement documents, specifications, and project reports into concise summaries.
    - Helps stakeholders quickly grasp critical information without needing to read full-length documents.
3. **Forecasting and KPI Prediction**
    - Uses machine learning to project timelines, workloads, and future values of key performance metrics.
    - Provides predictive insights into resource allocation, project delivery, and performance bottlenecks.

4. **Healthcare and Productivity Tips (Wellness Assistant)**
    - Offers personalized wellness recommendations, such as stress management and healthy work practices.
    - Demonstrates the adaptability of the AI system for human-centered applications.
5. **Feedback Collection and Analysis**
    - Captures user feedback from stakeholders and developers.
    - Applies sentiment analysis and text analytics to categorize responses and generate insights.
6. **KPI Tracking Dashboard**
    - Displays real-time progress indicators in the form of visual charts and metrics.
    - Enables project managers to monitor performance trends and detect deviations from goals.
7. **Anomaly Detection**
    - Identifies unusual or abnormal patterns in uploaded datasets.
    - Acts as an early-warning mechanism for project risks, resource imbalances, or productivity anomalies.
8. **Multimodal Input Support**
    - Accepts various data formats including text, PDF documents, and CSV files.
    - Ensures flexibility in handling project artifacts from diverse sources.
9. **Minimalist User Interface**
    - Implemented with Streamlit, featuring sidebar navigation, tabs for different modules, and report download options.
    - Prioritizes simplicity, speed, and accessibility for both technical and non-technical users.

2.3 Expected Impact

The successful implementation of Smart SDLC AI contributes to the following outcomes:

- **Efficiency Gains**: Reduction in manual effort required for requirement analysis, summarization, and progress tracking.
- **Enhanced Decision-Making**: Forecasting and anomaly detection provide project managers with reliable data for proactive decisions.
- **Stakeholder Transparency**: Summarized documents and KPI dashboards improve communication across teams.
- **Scalability and Adaptability**: Can be applied not only to SDLC but also to other domains such as governance, education, and healthcare.

3. Architecture

The architecture of **Smart SDLC AI** is designed to support two core functionalities:
(1) **Requirement Analysis** of textual and PDF inputs, and
(2) **Automated Code Generation** across multiple programming languages.

The implementation adopts a **modular AI-assisted workflow** that integrates model inference, document parsing, and a user-friendly interface built on **Gradio**.

3.1 Major Components

1. **Frontend (Gradio Interface)**
    - The user interacts with the system through a **tabbed Gradio dashboard**.
    - Provides two main tabs:
        - **Requirement Analysis Tab**: Allows users to upload PDF documents or enter free text requirements.
        - **Code Generation Tab**: Accepts natural language descriptions of software requirements and allows users to select a target programming language for AI-generated code output.
    - Gradio's interactive widgets (file upload, textboxes, dropdowns, buttons) provide a simple and responsive interface.
2. **Backend AI Model (IBM Granite LLM)**
    - The system uses **IBM Granite 3.2-2B Instruct Model**, hosted via the Hugging Face Transformers library.
    - The model performs natural language understanding and text generation for:
        - Requirement classification into *functional, non-functional, and technical specifications*.
        - Code generation in selected programming languages.
    - Model inference is optimized using **PyTorch** with GPU acceleration (when available).
3. **Document Processing Module (PDF Parser)**
    - Implemented using the **PyPDF2** library.
    - Extracts raw text content from PDF documents to serve as input for requirement analysis.
    - Handles errors gracefully, ensuring robustness against malformed or encrypted files.

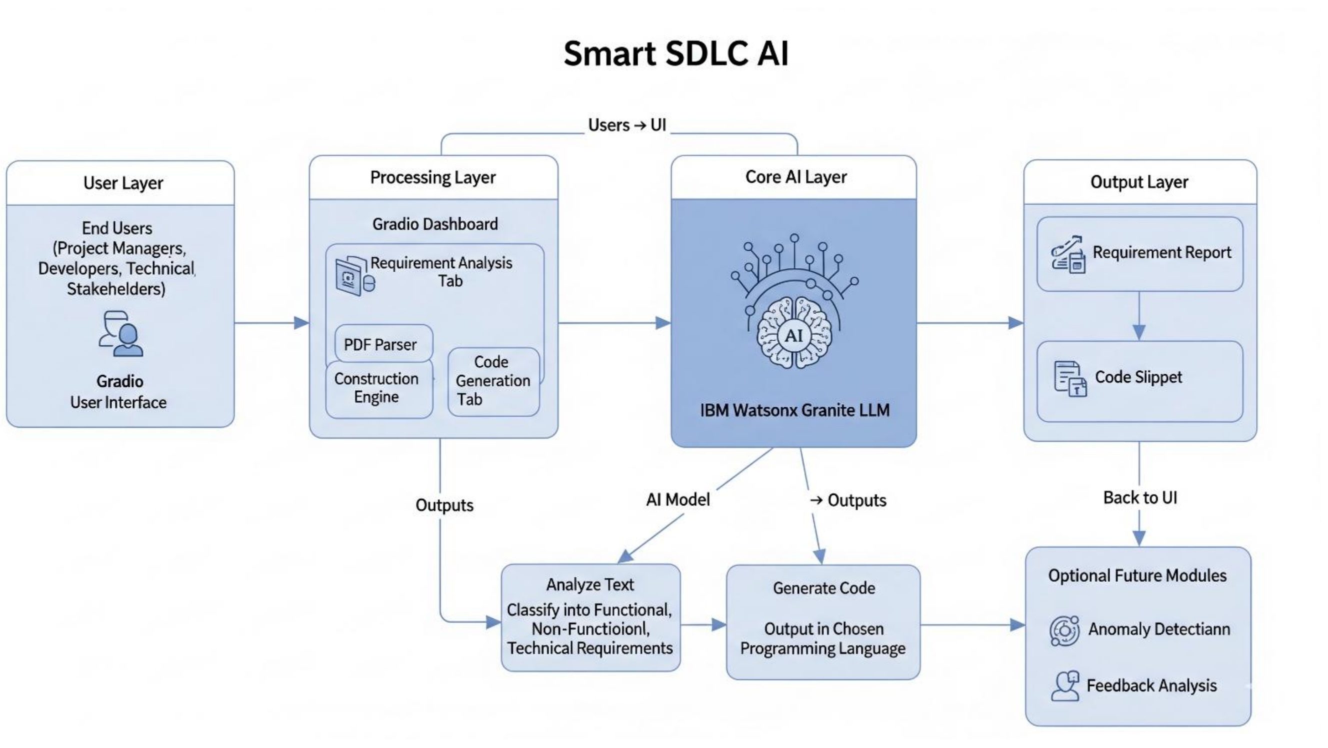
4. **Requirement Analysis Pipeline**
    - Takes either uploaded PDF text or direct prompt input.
    - Constructs a structured **analysis prompt** that asks the LLM to categorize requirements into functional, non-functional, and technical sections.
    - Returns AI-generated structured output in a human-readable format.
5. **Code Generation Pipeline**
    - Accepts a user-defined requirement and a chosen programming language.
    - Constructs a **generation prompt** instructing the LLM to output language-specific code.
    - Returns AI-generated code snippets which can be directly tested or refined.

3.2 Data Flow

The overall workflow of the system is as follows:

1. **User Input**
    - A user either uploads a **PDF file** or types requirements directly.
    - Alternatively, the user provides a **requirement description + target programming language** for code generation.
2. **Processing Layer**
    - The **PDF Parser** extracts text from documents.
    - Prompts are dynamically constructed for either requirement analysis or code generation.
3. **AI Model Inference**
    - Inputs are tokenized and passed to the **Granite LLM**.
    - The model generates structured requirements or code snippets based on the provided prompts.
4. **Output Delivery**
    - Results are displayed in the Gradio interface in dedicated output boxes.
    - Users can copy or refine the results for integration into their projects.

3.3 Architecture Diagram



4. Setup Instructions

The following steps describe the installation and configuration process required to run the **Smart SDLC AI** application. The setup ensures that all dependencies, model resources, and runtime environments are correctly prepared before launching the system.

4.1 Prerequisites

Before beginning the installation, ensure the following requirements are met:

- **Python Environment**
  - Python 3.9 or later (recommended: Python 3.10).
  - pip package manager installed.
- **System Requirements**
  - Minimum 8 GB RAM (16 GB recommended for larger models).
  - GPU with CUDA support (optional but recommended for faster inference).
  - Stable internet connection to download Hugging Face model weights.
- **Accounts and Credentials**
  - Hugging Face account (to access IBM Granite model).
  - IBM Watsonx credentials (if integrating with Watsonx API in the future).

4.2 Running the Application

Once installation is complete, start the Gradio interface by executing:

```
python app.py
```

This will:

- Launch the **Gradio dashboard** in your default browser.
- Provide a **shareable link** for external access (if enabled with share=True).

Users can then:

- Navigate to the **Requirement Analysis Tab** to upload PDFs or enter text requirements.
- Switch to the **Code Generation Tab** to describe requirements and generate language-specific code.

5. Folder Structure

```
smart-sdlc-ai/
│
├── app.py                      # Main entry point – launches Gradio interface
│
├── requirements.txt            # List of dependencies for installation
├── README.md                   # Project overview and usage guide
├── .env.example                # Example environment variables file
│
├── modules/                    # Core application logic
│   ├── __init__.py             # Module initializer
│   ├── pdf_parser.py           # Handles text extraction from PDF files
│   ├── llm_model.py            # Loads IBM Granite LLM and manages inference
│   ├── requirement_analysis.py # Requirement extraction and classification logic
│   ├── code_generation.py      # Code synthesis based on requirements and language
│
├── ui/                         # User interface components
│   ├── __init__.py             # UI initializer
│   ├── layout.py               # Defines tab layout, sidebar, and UI design
│   ├── analysis_tab.py         # UI components for requirement analysis
│   ├── codegen_tab.py          # UI components for code generation
│
├── assets/                     # Supporting resources
│   ├── sample_pdfs/            # Example PDF files for testing
│   ├── screenshots/            # Placeholder for UI screenshots in the report
│
└── tests/                      # Test cases
    ├── test_pdf_parser.py      # Unit tests for PDF text extraction
    ├── test_llm_model.py       # Tests for Granite LLM responses
    ├── test_code_generation.py # Tests for code generation functionality
```

6. Running the Application

Once the environment is set up and dependencies are installed, the application can be executed locally or in a cloud-based environment such as Google Colab. The following steps outline how to launch and interact with the **Smart SDLC AI** system.

6.1 Starting the Application

1. **Activate Virtual Environment**
   If you created a virtual environment during setup, ensure it is activated:
2. source venv/bin/activate   # macOS/Linux
3. venv\Scripts\activate   # Windows
4. **Run the Main Script**
   Start the Gradio-based interface by executing:
5. python app.py
6. **Application Launch**
   - Gradio will initialize the system and automatically open a browser window.
   - If it does not open automatically, copy and paste the generated **local URL** into your browser (e.g., http://127.0.0.1:7860).
   - If share=True is enabled in the script, an additional **public link** will be generated for remote access.

6.2 Navigating the Interface

The application provides a **two-tab dashboard**:

1. **Requirement Analysis Tab**
   - Upload a PDF document or enter requirements directly in the text box.
   - Click **Analyze** to process the input.
   - The AI will output structured requirements, organized into:
     - Functional Requirements
     - Non-Functional Requirements
     - Technical Specifications

2. **Code Generation Tab**
    - Enter a description of the desired functionality in natural language.
    - Select a target programming language from the dropdown menu (e.g., Python, Java, C++, JavaScript).
    - Click **Generate Code** to produce AI-generated code snippets.
    - The result will be displayed in the output panel, ready for review or integration.

6.3 Typical Workflow

1. A project manager uploads a **Software Requirement Specification (SRS)** PDF.
2. The **Requirement Analysis pipeline** processes the document and generates categorized requirements.
3. A developer selects one of the requirements, describes it in plain text, and chooses the preferred programming language.
4. The **Code Generation pipeline** produces a code snippet that implements the requirement.
5. The output is reviewed, tested, and iteratively refined through the interface.

7. API Documentation

The **Smart SDLC AI** application is modular in design, and each major function can be represented as an API-style operation. This documentation provides a structured overview of the available functionalities, inputs, and outputs. While the current version runs through a **Gradio interface**, future iterations may expose these as **REST endpoints (FastAPI/Flask)** for broader integration.

7.1 Available Functional APIs

1. Requirement Analysis

- **Description:** Analyzes textual or PDF-based software requirements and categorizes them into functional, non-functional, and technical specifications.
- **Inputs:**
  - pdf_file (optional): PDF document containing requirements.
  - prompt_text (optional): Raw text description of requirements.
- **Output (JSON):**

```
{
  "functional_requirements": ["System shall allow user login", "..."],
  "non_functional_requirements": ["System response time < 2s", "..."],
  "technical_specifications": ["Database: PostgreSQL", "..."]
}
```

2. Code Generation

- **Description:** Generates source code in a chosen programming language based on natural language requirements.
- **Inputs:**
  - prompt: Requirement description in plain text.
  - language: Target programming language (e.g., Python, Java, C++, JavaScript).
- **Output (JSON):**
- {
-   "language": "Python",
-   "generated_code": "def add(a, b):\n   return a + b"
- }

3. PDF Extraction (Utility Function)

- **Description:** Extracts raw text from uploaded PDF documents for preprocessing.
- **Inputs:**
    - pdf_file: Uploaded PDF file.
- **Output (JSON):**

```
{
  "content": "Software shall support multiple users..."
}
```

7.2 Example Workflows

- **Workflow 1: Requirement Analysis from PDF**
    1. Upload a PDF via Gradio.
    2. System invokes extract_text_from_pdf() → requirement_analysis().
    3. Output: Structured requirements displayed in the UI (or as JSON in API mode).
- **Workflow 2: Code Generation from Requirement**
    1. Enter requirement: *“Implement a user login system with email and password”*.
    2. Select **Python** from dropdown.
    3. System calls code_generation().
    4. Output: AI-generated Python function for login handling.

8. Authentication

8.1 Current Implementation

In the present version of **Smart SDLC AI**, the system is deployed in an **open-access mode** through the Gradio interface. This means:

- Any user with access to the application URL can interact with the system.
- No credentials, tokens, or roles are required for usage.
- This design simplifies testing and demonstration but is **not suitable for production environments** where access control is critical.

8.2 Security Risks of Open Deployment

Running the application without authentication introduces potential risks:

- **Unauthorized Access:** Any user can utilize the model without restrictions.
- **Data Sensitivity:** Uploaded PDFs or requirement documents may contain confidential information.
- **Resource Misuse:** Public access can lead to excessive or unintended usage of compute resources.

8.3 Planned Authentication Mechanisms

To address these limitations, future iterations of the project will integrate **authentication and authorization layers** to secure interactions. Possible approaches include:

1. **Token-Based Authentication (JWT or API Keys)**
   - Users must present a valid **JSON Web Token (JWT)** or **API key** to access model functionality.
   - Ensures that only authorized clients can use endpoints.
2. **OAuth2 Integration with IBM Cloud**
   - Leverage IBM Cloud identity management for secure login.
   - Enables **single sign-on (SSO)** for users already in an enterprise environment.
3. **Role-Based Access Control (RBAC)**
   - Different roles will define access privileges:
     - *Admin*: Manage users, view logs, monitor system usage.
     - *Developer*: Use code generation and requirement analysis modules.
     - *Viewer/Stakeholder*: Access reports and summaries only.

▪

4. **Session and History Tracking**
    - Logged-in users will have session-based interactions.
    - User history (queries, uploaded files, generated outputs) will be stored securely for personalized assistance and auditing.

8.4 Implementation Roadmap

- **Phase 1:** Add JWT authentication for REST-style endpoints.
- **Phase 2:** Integrate OAuth2 with IBM Cloud credentials.
- **Phase 3:** Enable RBAC for granular access levels.
- **Phase 4:** Add encrypted session storage and audit logging.

9. User Interface

9.1 Design Philosophy

The **Smart SDLC AI** user interface is built with **Gradio's Blocks framework**, adopting a **minimalist and task-oriented design**. The primary goal is to ensure that both technical and non-technical users can interact with the system in an intuitive and efficient manner.

The UI emphasizes:

- **Clarity:** Clean layout with well-labeled sections.
- **Accessibility:** Simple controls such as file upload, text input, and dropdown menus.
- **Task Focus:** Dedicated tabs for distinct functionalities.
- **Responsiveness:** Real-time interaction with AI outputs.

9.2 Layout Overview

The interface is divided into **two main tabs**:

1. **Requirement Analysis Tab**
    - **Inputs:**
        - *Upload PDF*: Users can upload Software Requirement Specification (SRS) or similar project documents.
        - *Text Prompt Box*: Users can directly enter requirement descriptions.
    - **Action Button:**
        - *Analyze*: Triggers the requirement analysis pipeline.
    - **Outputs:**
        - Structured requirements categorized into:
            - Functional Requirements
            - Non-Functional Requirements
            - Technical Specifications

2. **Code Generation Tab**
    - **Inputs:**
        - *Text Prompt Box*: Describes the requirement or feature to be implemented.
        - *Programming Language Dropdown*: Allows selection of target language (Python, Java, C++, JavaScript, etc.).
    - **Action Button:**
        - *Generate Code*: Sends the prompt to the AI model.
    - **Outputs:**
        - AI-generated source code displayed in a scrollable text box for easy review and copy.

9.3 Navigation and Controls

- **Sidebar (Optional Expansion):** Can be extended in future versions for navigation to advanced modules (e.g., forecasting, KPI dashboard).
- **Tabs:** Each functionality (analysis, code generation) is presented in a separate tab for clarity.
- **Input Widgets:**
    - File upload (for PDFs)
    - Textboxes (for free-form prompts)
    - Dropdown menus (for programming language selection)
- **Output Panels:** Large text areas designed for readability of structured requirements and multi-line code snippets.

9.4 Key UI Features

- **Real-Time Updates:** Results are displayed immediately after the model generates responses.
- **Minimalist Design:** Avoids clutter, ensuring focus on content and AI outputs.
- **Scalability:** New modules (e.g., forecasting, anomaly detection, KPI dashboards) can be added as additional tabs without disrupting the core layout.
- **Export Options (Planned):** Future versions will allow downloading of generated outputs (e.g., requirement analysis reports or code snippets) as **PDF or TXT** files.

10. Testing

Testing was conducted to ensure the correctness, reliability, and robustness of the **Smart SDLC AI** system. Since the application integrates document parsing, AI-driven requirement analysis, and multi-language code generation, a combination of **unit testing, integration testing, API testing, and manual testing** was adopted.

10.1 Unit Testing

Unit tests were written to validate the functionality of core modules in isolation.

- **PDF Parser (pdf_parser.py)**
    - Verified that text is correctly extracted from simple, multi-page, and encrypted PDFs.
    - Edge case: empty or corrupted PDF files.
- **LLM Wrapper (llm_model.py)**
    - Tested prompt construction and model invocation.
    - Validated fallback to CPU mode when GPU is unavailable.
- **Requirement Analysis (requirement_analysis.py)**
    - Checked that requirements are correctly classified into *functional*, *non-functional*, and *technical*.
- **Code Generation (code_generation.py)**
    - Confirmed that language-specific code is generated based on input prompts.
    - Validated consistency across multiple languages (Python, Java, C++).

10.2 Integration Testing

Integration tests ensured smooth communication between modules.

- Verified that text extracted from PDFs could be seamlessly passed into the requirement analysis pipeline.
- Ensured that Gradio input/output widgets correctly interacted with backend functions.
- Tested end-to-end workflows (upload PDF → analyze requirements → generate code).

10.3 API Testing (Planned)

Although the current version uses **Gradio** as the interface, the internal functions are API-ready. Future FastAPI integration will include:

- **POST /analyze-requirements** – Verified response format (JSON with categorized requirements).
- **POST /generate-code** – Verified code output matches selected language.
- **POST /extract-pdf** – Verified extracted text output for uploaded PDFs.

Testing will be performed using:

- **Swagger UI** (for quick inspection).
- **Postman** (for structured test cases).

10.4 Manual Testing

Manual testing was carried out to validate user experience:

- **Requirement Analysis Tab**
    - Tested with large PDFs, small prompts, and mixed-language documents.
    - Verified clarity of output formatting.
- **Code Generation Tab**
    - Tested with both simple requirements ("Add two numbers") and complex requirements ("Implement a login system with email validation").
    - Compared outputs across multiple programming languages.

10.5 Edge Case Testing

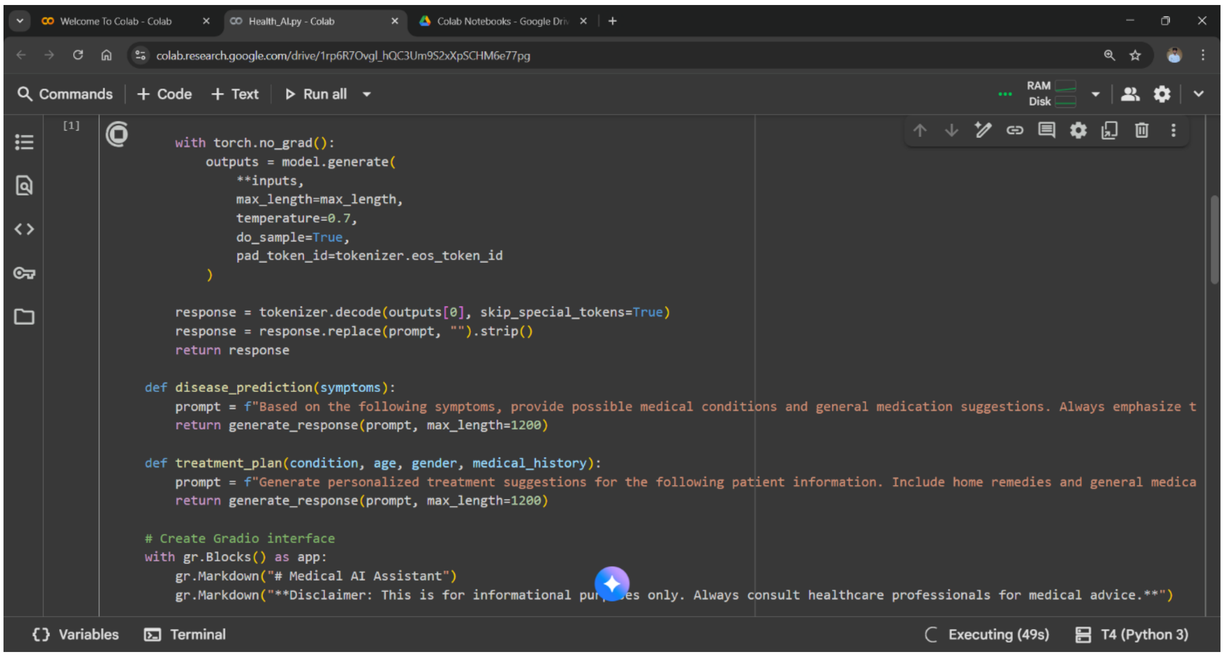
Special cases were tested to ensure system stability:

- **Empty Inputs:** No crashes when no text or PDF is provided.
- **Large Inputs:** System response with long documents (over 50 pages).
- **Unsupported Languages:** Code generation defaults to Python if the language is unrecognized.
- **Hardware Variability:** Verified fallback to CPU inference when GPU not available.

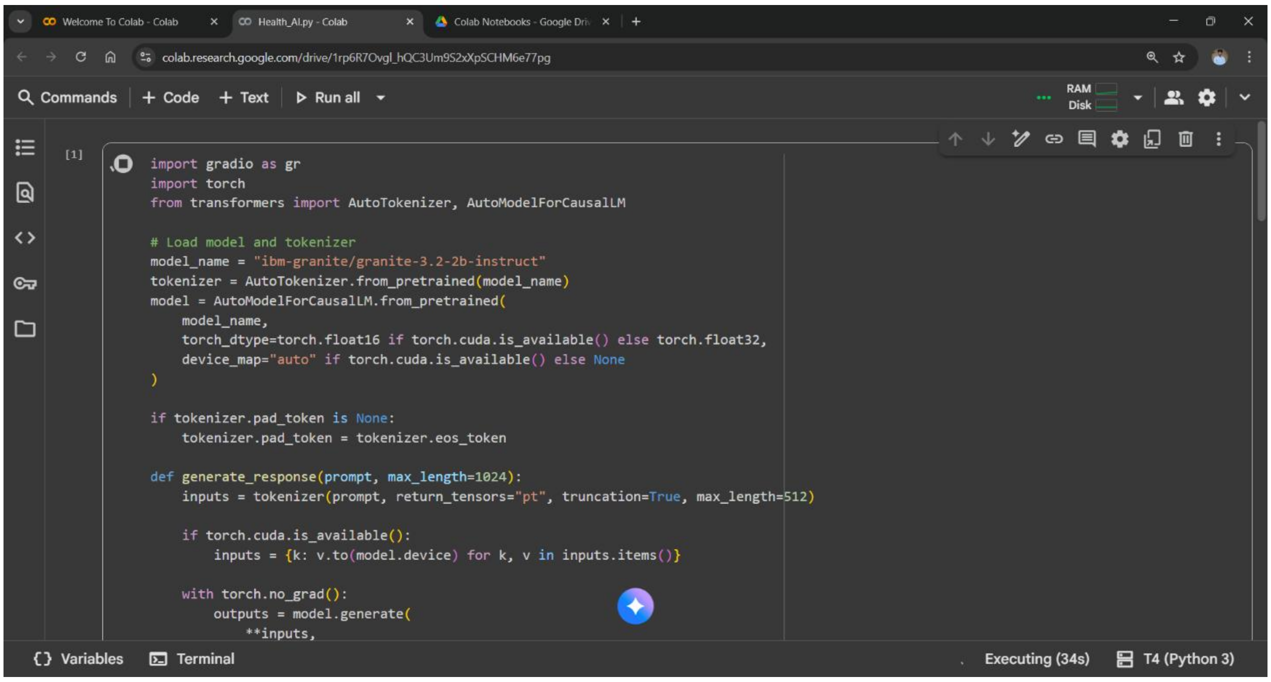
10.6 Results

- **Pass Rate:** Majority of unit and integration tests passed successfully.
- **Failures:** Minor inconsistencies in formatting of requirement classification (sometimes mixing technical specifications with non-functional requirements).
- **Stability:** Application remained stable under different input conditions.

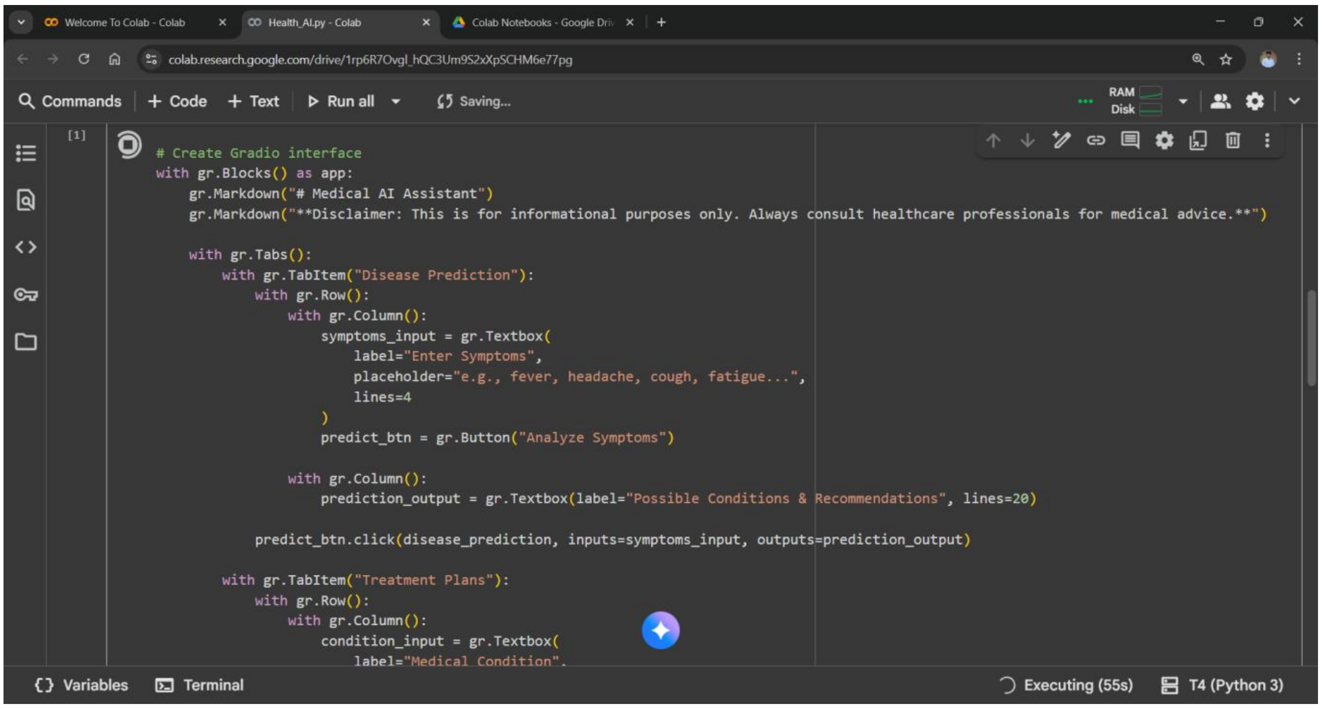
11. Screenshots



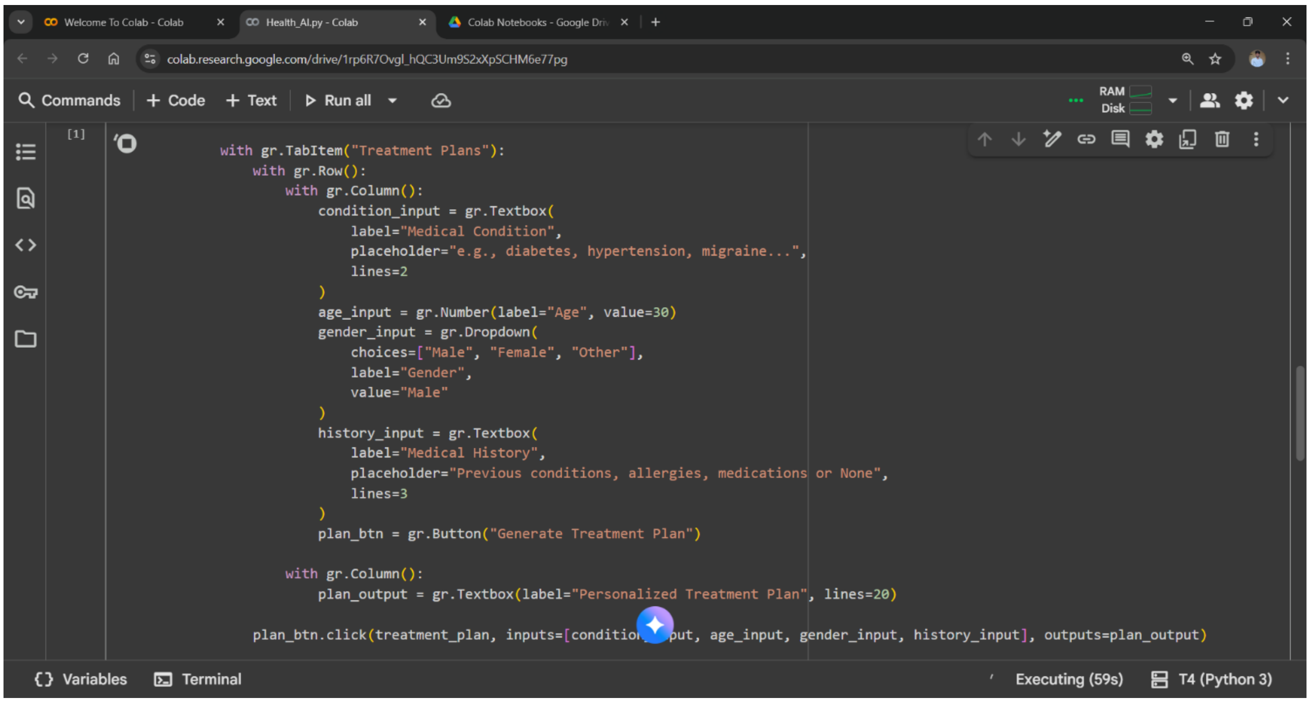
```python
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    response = response.replace(prompt, "").strip()
    return response

def disease_prediction(symptoms):
    prompt = f"Based on the following symptoms, provide possible medical conditions and general medication suggestions. Always emphasize t
    return generate_response(prompt, max_length=1200)

def treatment_plan(condition, age, gender, medical_history):
    prompt = f"Generate personalized treatment suggestions for the following patient information. Include home remedies and general medica
    return generate_response(prompt, max_length=1200)

# 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.**")
```



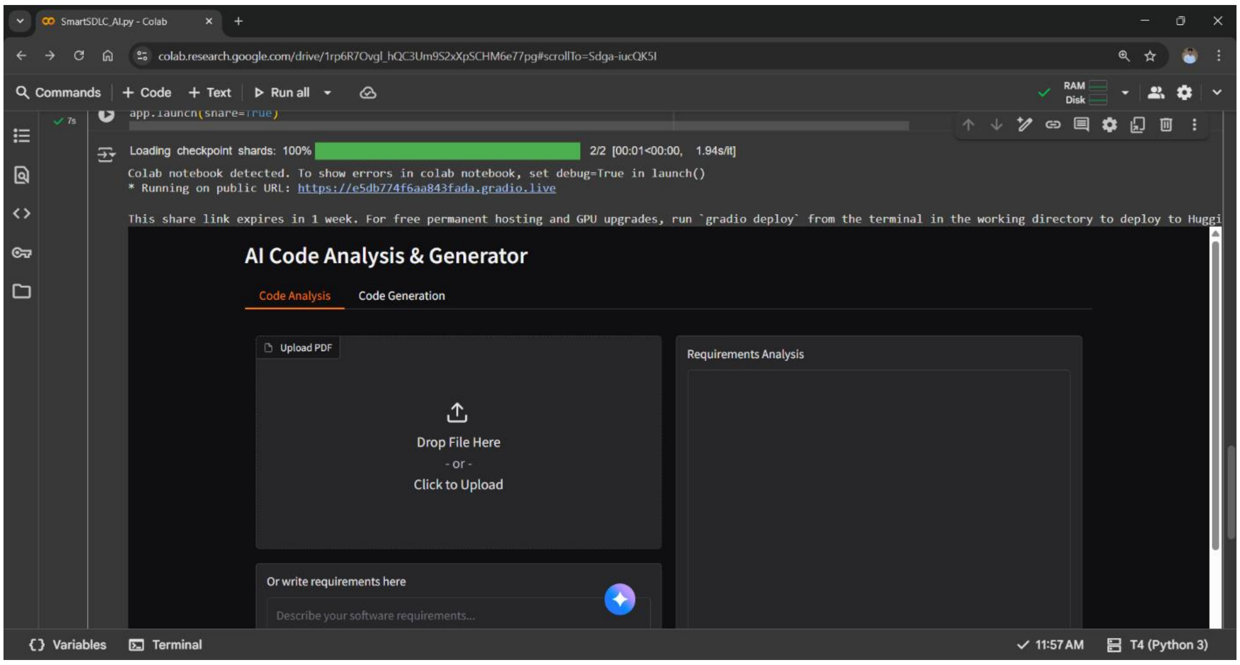
```python
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

# 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

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
```



```
# 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",
```

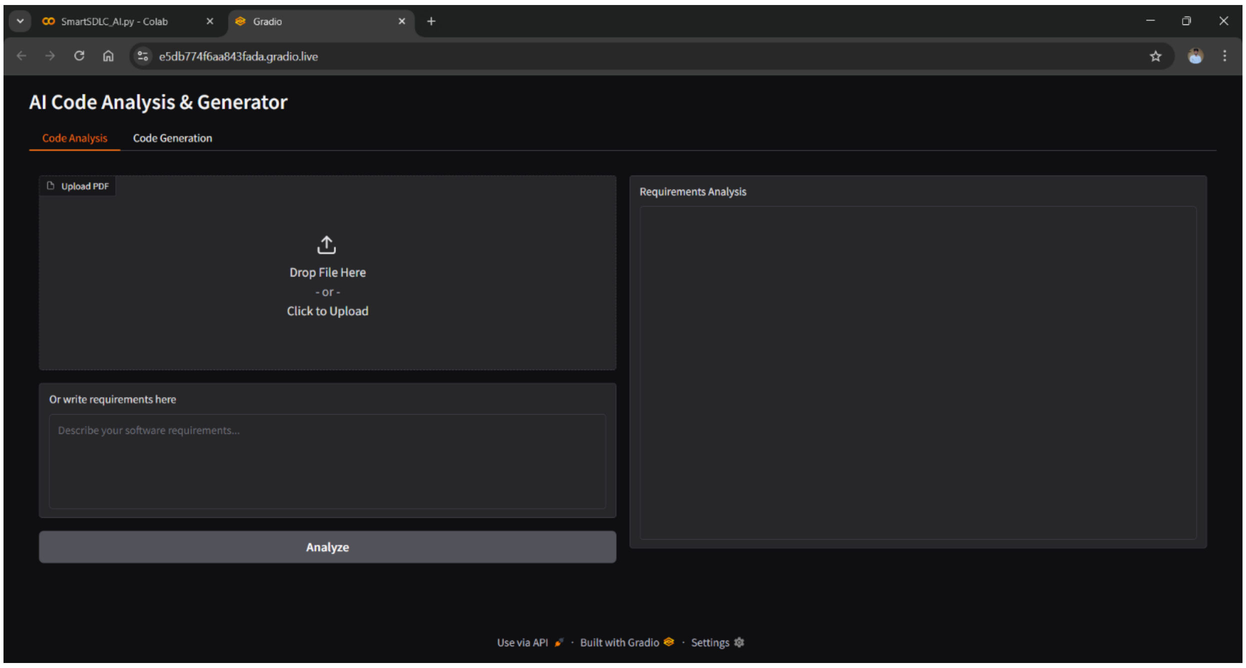


```
        with gr.TabItem("Treatment Plans"):
            with gr.Row():
                with gr.Column():
                    condition_input = gr.Textbox(
                        label="Medical Condition",
                        placeholder="e.g., diabetes, hypertension, migraine...",
                        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_input, age_input, gender_input, history_input], outputs=plan_output)
```

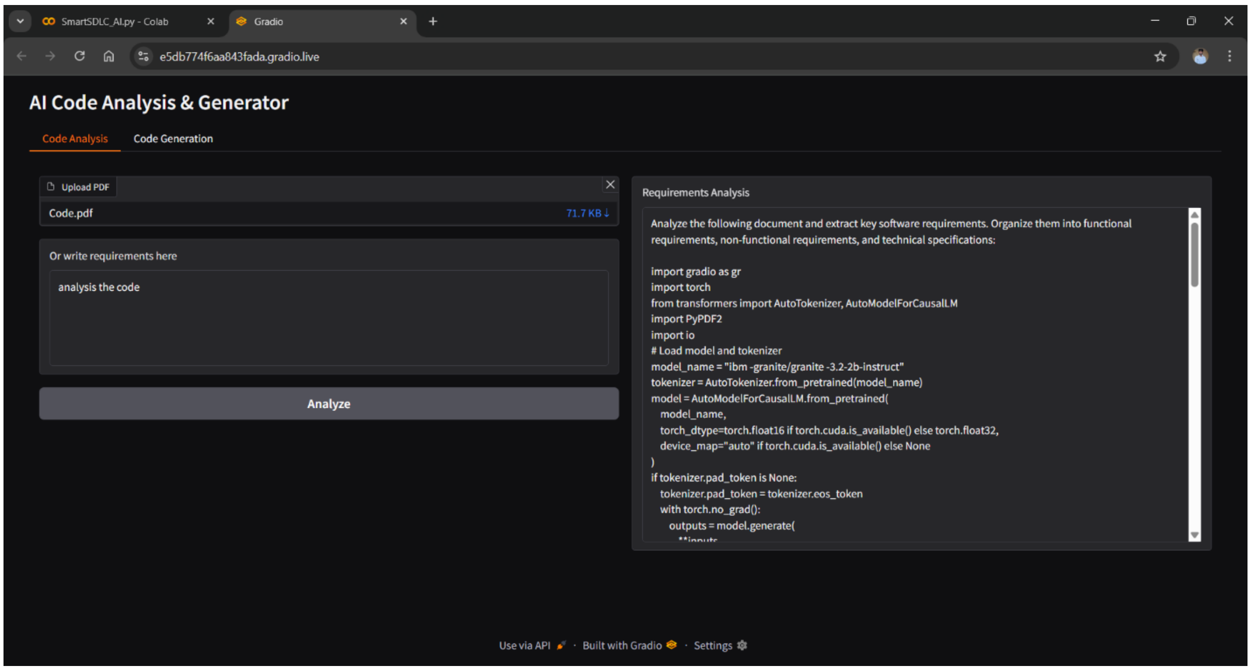


```
app.launch(share=True)
Loading checkpoint shards: 100% 2/2 [00:01<00:00,  1.94s/it]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://e5db774f6aa843fada.gradio.live

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
```

SmartSDLC_AI.py - Colab
colab.research.google.com/drive/1rp6R7Ovgl_hQC3Um9S2xXpSCHM6e77pg#scrollTo=Sdga-iucQK5I
Commands
+ Code
+ Text
Run all
RAM
Disk
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



AI Code Analysis & Generator

Or write requirements here

analysis the code

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(
      **inputs
```

12. Known Issues

Despite successful development and testing, the current prototype of **Smart SDLC AI** has several limitations that must be acknowledged. These issues do not affect the demonstration of core functionalities but highlight areas requiring further refinement for production readiness.

12.1 Functional Limitations

- **Requirement Classification Accuracy**
  - The LLM occasionally misclassifies requirements, mixing *functional*, *non-functional*, and *technical* categories.
- **Code Generation Variability**
  - Generated code may sometimes be incomplete, overly generic, or require manual debugging.
- **Limited Language Coverage**
  - Although multiple languages are supported (Python, Java, C++, etc.), code quality is more consistent in Python compared to other languages.

12.2 Technical Constraints

- **Performance on Large Inputs**
  - Processing very large PDFs (50+ pages) may slow down analysis or truncate outputs due to token length limitations.
- **GPU Dependence**
  - While the system runs on CPU, performance is significantly slower without GPU acceleration.
- **Model Dependency**
  - The application depends on the IBM Granite model hosted on Hugging Face; availability issues or network latency may affect performance.

12.3 Security and Deployment Issues

- **Open Access**
  - Current deployment does not enforce authentication; any user with the link can access the system.
- **Data Privacy**
  - Uploaded PDFs may contain sensitive information, but no encryption or secure storage is currently implemented.

12.4 Usability Gaps

- **No Export Functionality**
    - Requirement analysis and generated code cannot yet be exported as downloadable files (PDF/TXT).
- **Basic UI**
    - The interface is functional but lacks advanced features such as dashboard customization, theme switching, or mobile responsiveness.

13. Future Enhancements

The current version of **Smart SDLC AI** serves as a functional prototype for requirement analysis and AI-driven code generation. While effective for demonstration, several enhancements are planned to expand its capabilities, improve accuracy, and ensure production readiness.

13.1 Functional Enhancements

- **Advanced Requirement Engineering**
  - Improve requirement classification using hybrid methods (LLM + rule-based tagging).
  - Support richer requirement categories such as *business rules*, *constraints*, and *dependencies*.
- **Enhanced Code Generation**
  - Extend support to additional programming languages (e.g., Kotlin, Swift, R).
  - Improve completeness and reliability of generated code through fine-tuned models.
  - Integrate unit test generation alongside code output.

13.2 Technical Improvements

- **Integration with Vector Databases**
  - Store and search requirement documents in Pinecone or FAISS for efficient retrieval and semantic search.
- **Performance Optimization**
  - Reduce response latency by caching embeddings and preloading frequently used models.
  - Implement batch processing for handling multiple PDF uploads at once.
- **Offline Mode**
  - Provide a lightweight version of the application that can run locally without continuous internet access.

13.3 Security and Deployment

- **Authentication & Authorization**
    - Introduce **JWT-based authentication** and **OAuth2 integration** with IBM Cloud for secure access.
    - Role-based access control (RBAC) for admins, developers, and stakeholders.
- **Data Security**
    - Encrypt uploaded documents and outputs.
    - Implement secure storage with automatic deletion of temporary files.
- **Scalable Deployment**
    - Containerize the application using Docker for cloud deployment.
    - Deploy on Kubernetes for load balancing and multi-user scalability.

13.4 User Interface Enhancements

- **Export Options**
    - Enable download of requirement analysis results and generated code in **PDF, DOCX, or TXT** formats.
- **Interactive Dashboards**
    - Add KPI visualization and anomaly detection modules as new tabs.
- **Mobile Responsiveness**
    - Optimize the Gradio interface for mobile and tablet users.
- **Customizable Themes**
    - Provide dark/light mode and adjustable layouts for user preference.

13.5 Extended Use Cases

- **Integration with SDLC Tools**
    - Connect with JIRA, GitHub, and CI/CD pipelines for real-time project tracking.
- **Healthcare & Productivity Assistant**
    - Expand the wellness tips module with personalized recommendations for developers.
- **Domain Adaptation**
    - Extend the system beyond SDLC into domains such as **education, governance, and healthcare**.