

smartSDLC – AI-Enhanced Software Development Lifecycle

Project Documentation

1. Introduction:

- Project title : smartSDLC – AI-Enhanced Software Development Lifecycle
- Team member : Sebitha S
- Team member : Sindhuja I
- Team member : Srika R
- Team member : Sathiya R

2. project overview:

➤ *Purpose :*

The purpose of *SmartSDLC – AI-Enhanced Software Development Lifecycle* is to simplify and accelerate the software development process using Artificial Intelligence. It helps in analyzing software requirements and converting them into a structured format that is easy to understand. The system also supports automatic code generation in multiple programming languages such as Python, Java, C, and C++. This reduces manual effort by providing ready-to-use code snippets, improves accuracy in understanding requirements, and saves valuable development time. Overall, SmartSDLC acts as a smart assistant that bridges the gap between requirement gathering and implementation, making the software development lifecycle faster, smarter, and more efficient.

➤ *Features :*

- **Requirement Analysis**

Users can upload a PDF or type requirements as plain text. The system automatically analyzes and organizes them into:

- ❖ Functional Requirements
- ❖ Non-Functional Requirements
- ❖ Technical Specifications

- **Code Generation**

Based on the given requirements, the system can generate code in multiple programming languages such as Python, Java, C, and C++.

- **Simple User Interface**

A Gradio-based interface with two main tabs:

- ❖ *Code Analysis* – For requirement extraction and classification.
- ❖ *Code Generation* – For generating code in the selected language.

- **Customizable**

Users can provide their own requirements and choose their preferred programming language for generating relevant code.

- **Time-Saving**

Helps reduce manual effort by quickly analyzing requirements and producing usable code snippets, improving productivity for developers and learners.

3. Architecture:

- **Frontend (Gradio)**

The frontend is built using **Gradio**, which provides a clean and interactive web-based interface. It allows users to:

- Upload PDFs or type requirements.
- View categorized requirements (functional, non-functional, technical).

- Enter custom requirements and generate code in different programming languages.

The interface is organized into two main tabs:

- *Code Analysis Tab* – For uploading or typing requirements and extracting categorized specifications.
- *Code Generation Tab* – For generating code in the selected programming language.

- **Backend (Google Colab + Python)**

The backend runs in **Google Colab**, where Python handles model loading, processing, and communication with the AI model. It:

- Extracts text from PDFs.
- Sends user prompts to the AI model.
- Returns structured analysis or generated code to the Gradio UI.

- **LLM Integration (IBM Granite)**

The project integrates the **IBM Granite Large Language Model (LLM)** for natural language understanding and generation. It is responsible for:

- Analyzing software requirements.
- Categorizing them into functional, non-functional, and technical specifications.
- Generating accurate and usable code snippets in multiple languages.

- **PDF Processing (PyPDF2)**

The system uses **PyPDF2** to extract text from uploaded PDF documents. The extracted content is passed to the AI model for further analysis and classification.

4. Set up Instruction:

Prerequisites

- A **Google Account** to access Google Colab.

- Stable **internet connection** to install required libraries and load AI models from the cloud.
- A **Hugging Face account** for accessing the IBM Granite model.
- Basic knowledge of Python and Google Colab environment.

Installation Process

1. Open **Google Colab**.
2. Create a **new notebook**.
3. Change the runtime to **T4 GPU** (for faster execution).
4. In the first cell, install the required libraries:

```
“!pip install transformers torch gradio PyPDF2”
```
5. After installation, put the project code into the next cell.
6. Run the notebook cells in sequence to launch the application.

5. Folder Structure:

Since the project is developed and executed entirely in **Google Colab**, it consists of a single main notebook file.

Structure:

project/

└─ SmartSDLC.ipynb

- **SmartSDLC.ipynb** – The main Google Colab notebook that contains:
 - Code for requirement analysis and categorization.
 - Code for generating program snippets in multiple languages.
 - Gradio interface setup for user interaction.
 - Integration with the IBM Granite model for AI processing.

6. Running the Application:

1. Open the **Google Colab notebook (SmartSDLC.ipynb)**.
2. Run the first cell to install all required libraries.
3. Run the subsequent cells to load the IBM Granite model and set up the Gradio interface.
4. In the final cell, execute:

`"app.launch(share=True)"`

5. Google Colab will display a public Gradio link such as:

`"Running on public URL: https://xxxx.gradio.live"`

Click the link to open the application in a browser tab.

The application provides two main functionalities:

- **Code Analysis** → Upload a PDF or type requirements → Click *Analyze* → Get categorized requirements (Functional, Non-Functional, Technical).
- **Code Generation** → Enter requirements + select a programming language → Click *Generate Code* → AI generates source code based on the input.

7. API Documentation:

The backend logic is implemented in Python within the Google Colab environment and powered by the IBM Granite model. While the system primarily runs through the Gradio interface, the following conceptual API endpoints represent the core functionality:

- **POST /analyze-requirements**
 - **Description:** Accepts either a PDF file or plain text containing software requirements.

- **Response:** Returns an AI-generated analysis organized into *Functional Requirements*, *Non-Functional Requirements*, and *Technical Specifications*.
- **POST /generate-code**
 - **Description:** Accepts a requirement description and the selected programming language.
 - **Response:** Returns AI-generated source code in the chosen language.
- **POST /extract-pdf-text**
 - **Description:** Accepts a PDF file.
 - **Response:** Extracts and returns plain text for further processing or analysis.

Note: These endpoints are conceptual representations for documentation purposes. In practice, all interactions are handled directly through the Gradio interface within Google Colab.

8. Authentication:

Currently, the project runs in an open environment inside **Google Colab** with a shareable Gradio link. This means:

- Anyone with the link can access the application.
- No login or security mechanism is enforced by default.

Planned Enhancements for Authentication:

- **User Login System** – Add a secure login (username/password).
- **Token-Based Authentication (JWT/API Keys)** – Protect API calls for requirement analysis and code generation.
- **Role-Based Access Control (RBAC)** – Define different access levels for developers, testers, and administrators.

- **OAuth2 Integration** – Use authentication via trusted providers (e.g., Google, GitHub).
- **User Sessions and History Tracking** – Allow users to track their past analyses and generated code.

These features will improve the security and usability of *SmartSDLC* when deployed beyond the Colab environment.

9. User Interface:

The user interface of *SmartSDLC* is designed with **Gradio** to provide a simple and accessible experience for developers and learners.

Key Components:

- **Google Colab Notebook Integration** – The entire project runs inside Colab, where Gradio provides the interactive web interface.
- **Two Main Tabs:**
 1. **Code Analysis Tab** – Users can upload a PDF or enter requirements as text. The AI analyzes and categorizes them into *Functional*, *Non-Functional*, and *Technical Specifications*.
 2. **Code Generation Tab** – Users can enter a requirement and select a programming language. The AI then generates the corresponding source code.
- **Textboxes for Input and Output** – Simple and clear fields for entering requirements and displaying analysis or generated code.
- **Action Buttons** –
 - *Analyze* → Runs requirement analysis.
 - *Generate Code* → Produces code in the selected language.
- **Public/Local Link Access** – Gradio provides a temporary shareable link that allows users to access the interface from any browser or device.

The design prioritizes **ease of use, clarity, and quick interaction**, ensuring even beginners can operate the tool effectively.

10. Testing:

To ensure the reliability and accuracy of *SmartSDLC – AI-Enhanced Software Development Lifecycle*, different levels of testing were performed:

- **Functionality Testing**
Verified that the *Code Analysis* tab correctly extracts and categorizes requirements from both PDF uploads and manually entered text.
- **Code Generation Testing**
Confirmed that the *Code Generation* tab successfully produces code in multiple programming languages (Python, Java, C, C++, JavaScript) based on user requirements.
- **Model Response Validation**
Ensured that the IBM Granite model generates meaningful, structured, and readable output without crashes or irrelevant content.
- **Interface Testing**
Checked that all buttons (*Analyze* and *Generate Code*) work as expected and display results properly in the output textboxes.
- **Cross-Device Access Testing**
Tested the Gradio public link on different browsers and devices (desktop and mobile) to confirm smooth operation and consistent performance.
- **Edge Case Handling**
Validated how the system responds to unexpected inputs, such as incomplete requirements, unsupported programming languages, or empty PDF files.

11. Known Issues:

- **Performance Dependence on Internet Speed**

Since the project runs in Google Colab and relies on cloud-based models, it may run slowly on weak internet connections.

- **Inaccurate or Incomplete Outputs**

At times, the AI model may generate partial, inaccurate, or less-optimized code and requirement analysis.

- **No Authentication**

The current version does not include login or security features. Anyone with the Gradio link can access the application.

- **Limited Programming Language Support**

Code generation is restricted to the languages supported by the AI model (Python, Java, C, C++, JavaScript, etc.).

- **Dependency on Google Colab**

The application requires Google Colab to run and cannot function offline or outside a cloud environment.

12. Future Enhancements:

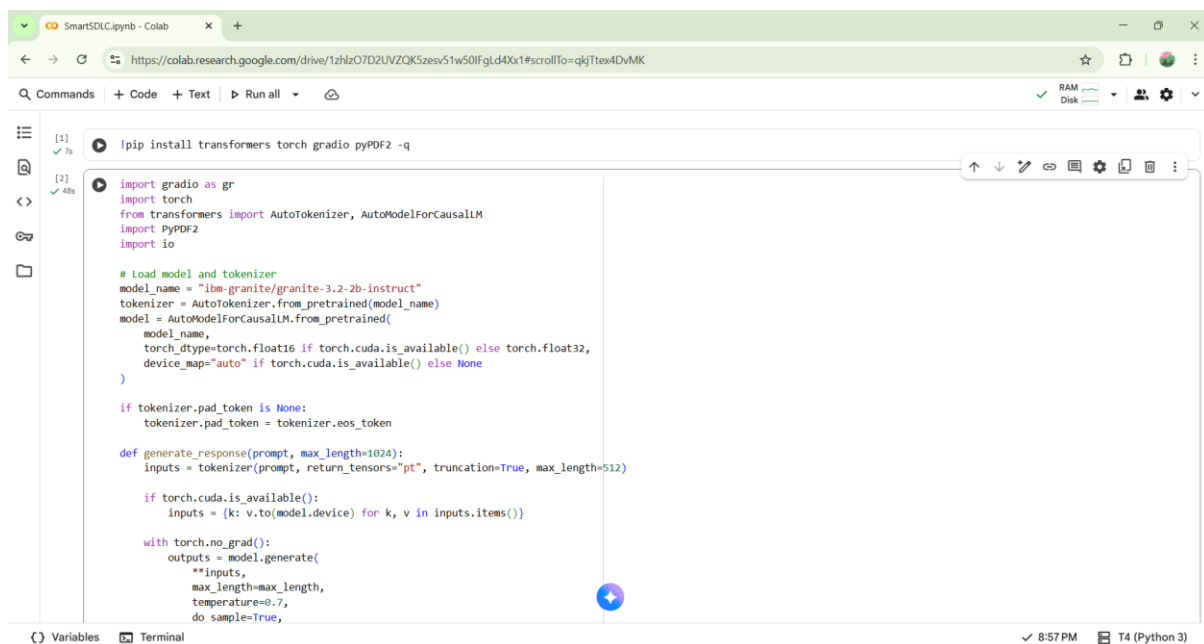
The following improvements can be considered for future versions of the project:

- **User Authentication:** Implement a login system to enhance security and personalize user experience.
- **Download Option:** Allow users to save generated code or analysis directly for offline use.
- **AI Improvements:** Enhance the AI model to provide more accurate, detailed, and context-aware results.
- **Real-Time Collaboration:** Enable multiple users to work together simultaneously on the same project or analysis.
- **Direct Deployment:** Provide options to deploy outputs directly as a website or mobile application for easier access.
- **Enhanced UI/UX:** Improve the interface with better formatting, themes, and user-friendly design.

Additional File Formats: Support more file types, such as DOCX, TXT, and others, for wider usability.

13. Screenshots:

➤ Program:



```
[1] ✓ 7s | pip install transformers torch gradio pyPDF2 -q

[2] ✓ 48s | import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2
import io

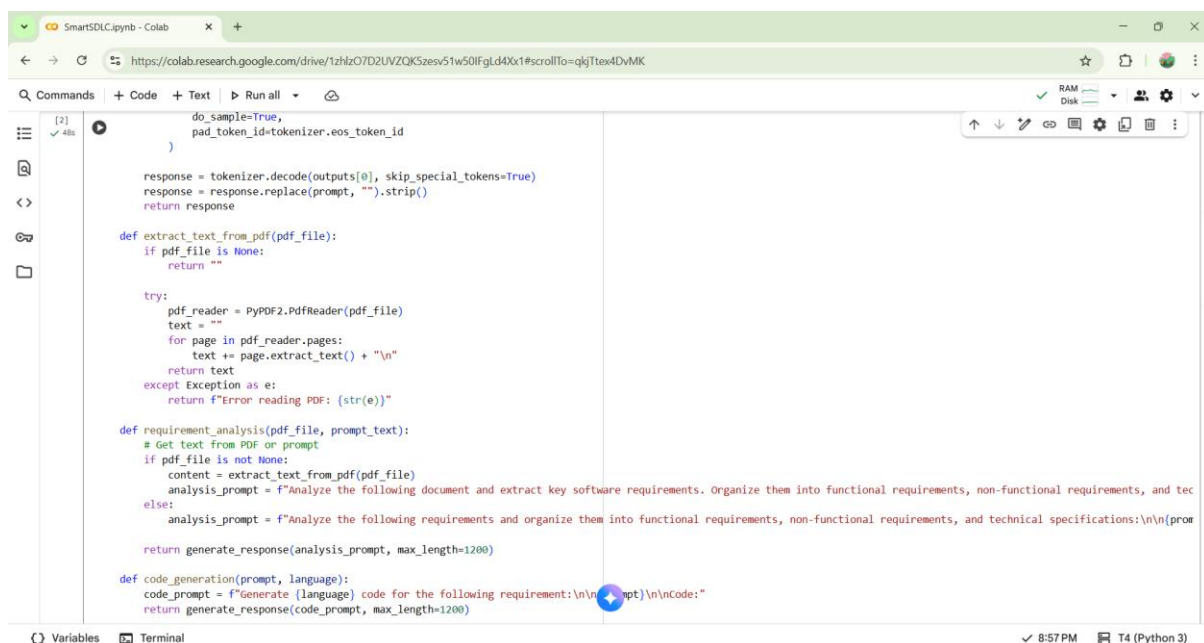
# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

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,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
```



```
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 extract_text_from_pdf(pdf_file):
    if pdf_file is None:
        return ""

    try:
        pdf_reader = PyPDF2.PdfReader(pdf_file)
        text = ""
        for page in pdf_reader.pages:
            text += page.extract_text() + "\n"
        return text
    except Exception as e:
        return f"Error reading PDF: {str(e)}"

def requirement_analysis(pdf_file, prompt_text):
    # Get text from PDF or prompt
    if pdf_file is not None:
        content = extract_text_from_pdf(pdf_file)
        analysis_prompt = f"Analyze the following document and extract key software requirements. Organize them into functional requirements, non-functional requirements, and technical specifications:\n\n{content}"
    else:
        analysis_prompt = f"Analyze the following requirements and organize them into functional requirements, non-functional requirements, and technical specifications:\n\n{prompt_text}"

    return generate_response(analysis_prompt, max_length=1200)

def code_generation(prompt, language):
    code_prompt = f"Generate {language} code for the following requirement:\n\n{prompt}\n\nCode:"
    return generate_response(code_prompt, max_length=1200)
```

```
return generate_response(code_prompt, max_length=1200)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# AI Code Analysis & Generator")

    with gr.Tabs():
        with gr.Tabitem("Code Analysis"):
            with gr.Row():
                with gr.Column():
                    pdf_upload = gr.File(label="Upload PDF", file_types=[".pdf"])
                    prompt_input = gr.Textbox(
                        label="Or write requirements here",
                        placeholder="Describe your software requirements...",
                        lines=5
                    )
                    analyze_btn = gr.Button("Analyze")

                with gr.Column():
                    analysis_output = gr.Textbox(label="Requirements Analysis", lines=20)

            analyze_btn.click(requirement_analysis, inputs=[pdf_upload, prompt_input], outputs=analysis_output)

        with gr.Tabitem("Code Generation"):
            with gr.Row():
                with gr.Column():
                    code_prompt = gr.Textbox(
                        label="Code Requirements",
                        placeholder="Describe what code you want to generate...",
                        lines=5
                    )
                    language_dropdown = gr.Dropdown(
                        choices=["Python", "JavaScript", "Java", "C++", "C#", "Go", "Rust"],
                        label="Programming Language",

```

```
label="Programming Language",
value="Python"
)
generate_btn = gr.Button("Generate Code")

with gr.Column():
    code_output = gr.Textbox(label="Generated Code", lines=20)

generate_btn.click(code_generation, inputs=[code_prompt, language_dropdown], outputs=code_output)

app.launch(share=True)
```

/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret 'HF_TOKEN' does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret in your Google Colab and restart your ses
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
`torch_dtype` is deprecated! Use `dtype` instead!
Loading checkpoint shards: 100% 2/2 [00:15<00:00, 6.52s/it]
Colab notebook detected. To show errors in Colab notebook, set debug=True in launch()
* Running on public URL: <https://dc73d8e9cfbe9526ab.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 Hugging Face Spaces (<https://huggingface.co/spaces>)

AI Code Analysis & Generator

Code Analysis

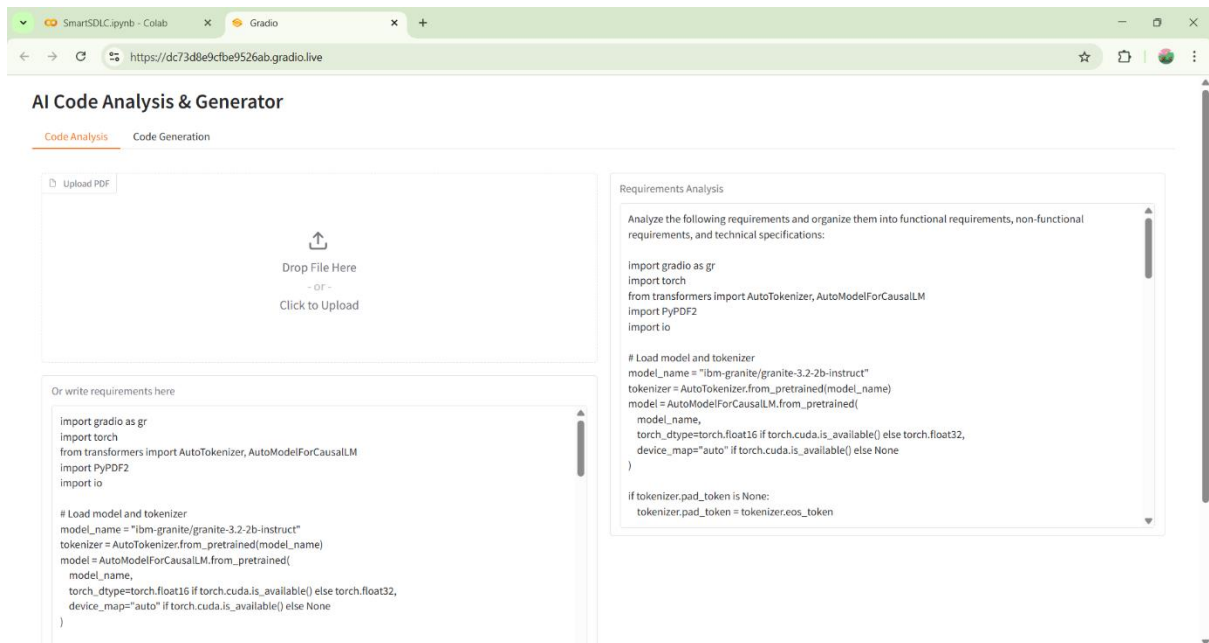
Code Generation

Upload PDF

Requirements Analysis

14. Output:

➤ Code Analysis:



The screenshot shows the 'AI Code Analysis & Generator' web application with the 'Code Analysis' tab selected. The interface includes a file upload section with a 'Drop File Here' instruction and a 'Click to Upload' button. Below this is a text area for 'Or write requirements here' containing a Python code snippet for loading a model and tokenizer. To the right, the 'Requirements Analysis' section displays the same code snippet, along with a 'Requirements Analysis' header and a brief instruction to analyze requirements.

Upload PDF

Drop File Here
~ or ~
Click to Upload

Or write requirements here

```
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
)
```

Requirements Analysis

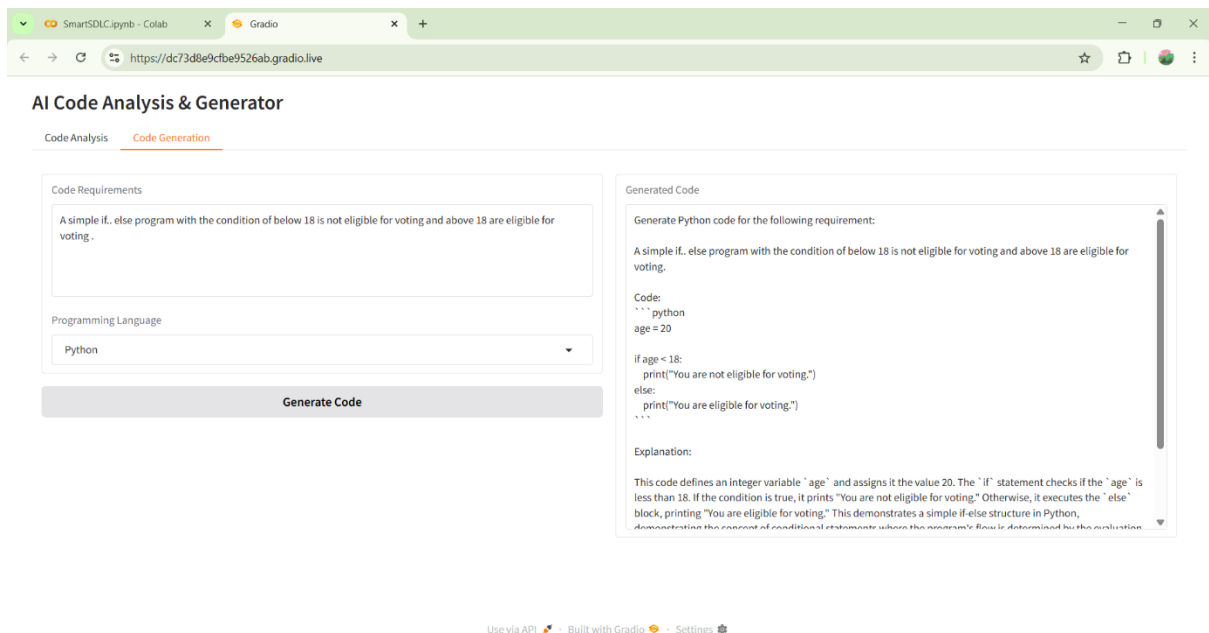
Analyze the following requirements and organize them into functional requirements, non-functional requirements, and technical specifications:

```
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
```

➤ Code Generator:



The screenshot shows the 'AI Code Analysis & Generator' web application with the 'Code Generation' tab selected. The interface includes a 'Code Requirements' section with a text area containing a requirement: 'A simple if..else program with the condition of below 18 is not eligible for voting and above 18 are eligible for voting.'. Below this is a 'Programming Language' dropdown menu set to 'Python'. A 'Generate Code' button is located below the dropdown. To the right, the 'Generated Code' section displays the generated Python code, which implements the requirement using an if-else statement. Below the code is an 'Explanation' section that describes the code's logic.

Code Requirements

A simple if..else program with the condition of below 18 is not eligible for voting and above 18 are eligible for voting.

Programming Language

Python

Generate Code

Generated Code

Generate Python code for the following requirement:

A simple if..else program with the condition of below 18 is not eligible for voting and above 18 are eligible for voting.

Code:

```
python
age = 20

if age < 18:
    print("You are not eligible for voting.")
else:
    print("You are eligible for voting.")
...
```

Explanation:

This code defines an Integer variable 'age' and assigns it the value 20. The 'if' statement checks if the 'age' is less than 18. If the condition is true, it prints "You are not eligible for voting." Otherwise, it executes the "else" block, printing "You are eligible for voting." This demonstrates a simple if-else structure in Python, demonstrating the concept of conditional statements where the program's flow is determined by the evaluation.

Use via API Built with Gradio Settings