

SmartSDLC – AI-Enhanced Software Development Lifecycle

Project Documentation

1.Introduction

- ✚ **Project title** : SmartSDLC – AI-Enhanced Software Development Lifecycle
- ✚ **Team ID** : NM2025TMID02144
- ✚ **Team Leader** : PRAVIN RAJA V
- ✚ **Team member** : SAHAYA GOSPER JOSHIN S
- ✚ **Team member** : NAVEEN NARESH R
- ✚ **Team member** : TAMIL SELVAM Y

2.Project Overview

Purpose:


This project is an **AI Code Analysis & Generator** web application built using **Gradio** and **Hugging Face Transformers**. It leverages the `ibm-granite/granite-3.2-2b-instruct` model to perform two primary functions: analyzing software requirements and generating code snippets. Users can either upload a PDF document or type in their requirements for analysis. The application categorizes these requirements into functional, non-functional, and technical specifications. Additionally, it can generate code in various programming languages based on user prompts, streamlining the development process. The application is designed to be accessible via a user-friendly Gradio interface.

3.Architecture

The application's architecture is a simple monolithic structure with a clear separation of concerns, all running within a single Python script.

- **Frontend (UI):** The user interface is built using the Gradio library. It provides a web-based, interactive dashboard with separate tabs for "Code

Analysis" and "Code Generation". Gradio handles all user input (text boxes, file uploads, dropdowns) and output display.

 **Backend (Core Logic):** The backend is written in Python. It's responsible for:

- **Model Management:** Loading the ibm-granite/granite-3.2-2b-instruct model and its tokenizer from Hugging Face. It uses torch for managing the model and utilizes GPU acceleration (device_map="auto") if available.
 - **Requirement Analysis:** The requirement_analysis function processes text from either a PDF (via PyPDF2) or a text input field. It then crafts a detailed prompt for the AI model to categorize the requirements.
 - **Code Generation:** The code_generation function takes a user prompt and a selected programming language to formulate a specific prompt for the AI model to generate code.
 - **AI Interaction:** The generate_response function is the core of the backend. It prepares inputs for the transformer model, runs the inference, and decodes the model's output into a human-readable response. It manages tokenization, input tensor creation, and GPU acceleration.
-

4.Setup Instructions ✂

To get the application up and running on your local machine, follow these steps.

1. Clone the Repository (if applicable):

Bash

```
git clone <repository_url>  
cd <repository_name>
```

2. Create and Activate a Virtual Environment:

Bash

```
python -m venv venv  
# On Windows  
venv\Scripts\activate  
# On macOS/Linux  
source venv/bin/activate
```

3. **Install Dependencies:** The provided script requires a few key libraries. Install them using pip.

Bash

```
pip install gradio torch transformers PyPDF2
```

Note: If you have a CUDA-enabled GPU, you should also install the appropriate version of torch for GPU acceleration. Refer to the PyTorch website for instructions.

5.Folder Structure

The project is a **single-file application**, meaning all the code is contained within one script.

```
/ai-code-app/  
| README.md  
| app.py      # The main application script with all the code
```

- **app.py:** Contains all the logic for the Gradio interface, model loading, requirement analysis, and code generation. This is the only file you need to run.
-

6.Running the Application

Once you have completed the setup, running the application is a single command.

1. **Navigate to the project directory** if you are not already there.
2. **Run the Python script:**

Bash

```
python app.py
```

3. **Access the web interface:** The script will start a local web server and provide a URL in the console, typically `http://127.0.0.1:7860`. Open this URL in your web browser. The `share=True` parameter in the `app.launch()`

command also provides a public URL for sharing, which will be valid for a limited time.

7.API Documentation 📖

This application does not have a traditional REST API. It is an interactive web application that uses Gradio's backend to handle function calls from the UI. The "API" is effectively the direct invocation of the Python functions `requirement_analysis` and `code_generation` via the Gradio interface.

🚦 **Endpoint for Analysis:** The Gradio UI button `analyze_btn` triggers the `requirement_analysis` function.

- **Inputs:** `pdf_file` (a Gradio `gr.File` object) and `prompt_text` (a `gr.Textbox` string).
- **Outputs:** A string of text containing the analyzed requirements.

🚦 **Endpoint for Code Generation:** The `generate_btn` button triggers the `code_generation` function.

- **Inputs:** `code_prompt` (a `gr.Textbox` string) and `language_dropdown` (a `gr.Dropdown` string).
 - **Outputs:** A string of generated code.
-

8.Authentication 🔑

The current application **does not implement any authentication or authorization**. It is designed as a standalone, public-facing tool for demonstration and personal use. Any user who can access the URL can use its full functionality. For a production environment, this would need to be integrated with an authentication system like OAuth, JWTs, or a simple username/password model.

9.User Interface 🎨

The user interface is built with **Gradio**, which automatically creates a clean and responsive design.

- ✚ **Tabs:** The interface is organized into two distinct tabs:
 - **Code Analysis Tab:** Contains a file upload component for PDFs, a text area for manual input, and a button to trigger the analysis. The output is displayed in a large, scrollable text box.
 - **Code Generation Tab:** Includes a text area for describing the desired code, a dropdown menu to select the programming language, and a button to generate the code. The generated code is shown in a separate text box.
 - ✚ **Aesthetics:** Gradio's default theme is used, providing a modern, minimalist design that is intuitive to navigate.
-

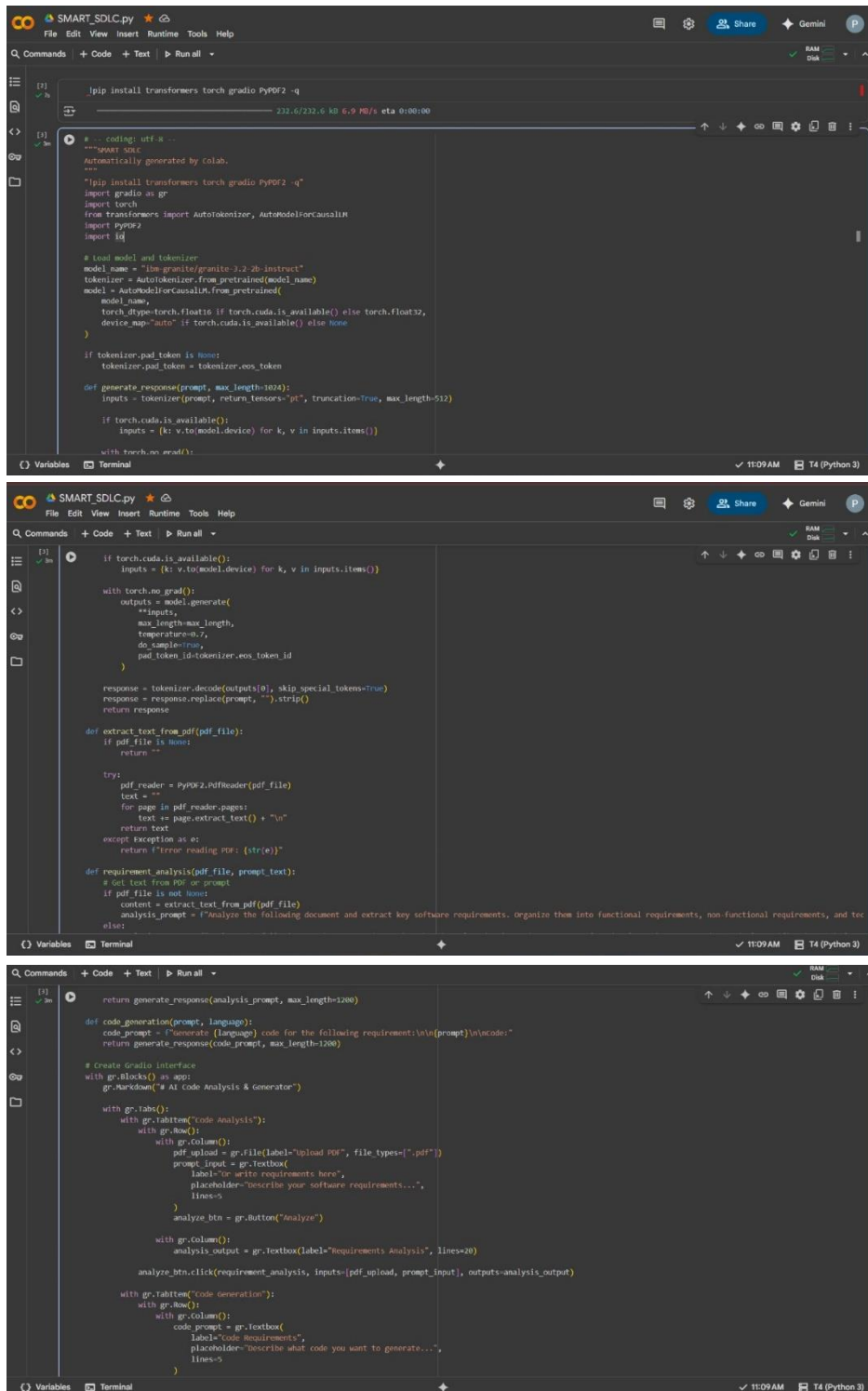
10. Testing

Given the nature of this project, testing can be approached from several angles.

- **Functional Testing:** Manually test the two main functions of the application:
 1. **Analysis:** Upload a PDF with requirements and see if the output is correctly categorized. Type requirements directly into the text box and check if the analysis is accurate.
 2. **Generation:** Provide a clear prompt for a code snippet (e.g., "Write a Python function to reverse a string") and verify that the generated code is correct and in the selected language.
- **Integration Testing:** Ensure that the Gradio UI correctly communicates with the backend Python functions. Verify that button clicks trigger the right functions and that the outputs are displayed correctly on the web page.
- **Performance Testing:** Monitor the application's response time, especially when generating long responses or processing large PDF files. Check if the GPU acceleration is being utilized as expected.
- **Error Handling:** Test the application's behavior with invalid inputs, such as uploading a non-PDF file, leaving the input fields blank, or providing ambiguous prompts. The current implementation includes a try-except block for PDF reading, which is a good starting point.

11. Screen shots

1.Input



```
!pip install transformers torch gradio PyPDF2 -q

# ... coding: utf-8 ...
"""
SMART SDLC
Automatically generated by Colab.
"""

!pip install transformers torch gradio PyPDF2 -q
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2
import id

# Load model and tokenizer
model_name = "lms-granite/granite-7.2-3b-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,
            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 test requirements."
    else:
        analysis_prompt = 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)

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

    with gr.Tab("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.Tab("Code Generation"):
        with gr.Row():
            code_prompt = gr.Textbox(
                label="Code Requirements",
                placeholder="Describe what code you want to generate...",
                lines=5
            )
            generate_btn = gr.Button("Generate")

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

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

```
choices=["python", "Javascript", "Java", "C++", "C#", "PHP", "Go", "Rust"],
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 session.
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(
tokenizer_config.json 8.88k/? [00:00<00:00, 220kB/s]
vocab.json 777k/? [00:00<00:00, 5.48MB/s]
merges.txt 442k/? [00:00<00:00, 7.01MB/s]
tokenizer.json 3.48M/? [00:00<00:00, 39.4MB/s]
added_tokens.json 100% [00:00<00:00, 87.0k/s] [00:00<00:00, 4.07MB/s]
special_tokens_map.json 100% [00:00<00:00, 701/701] [00:00<00:00, 35.4kB/s]
config.json 100% [00:00<00:00, 786/786] [00:00<00:00, 38.4kB/s]
'torch_dtype' is deprecated! Use 'dtype' instead!
model.safetensors.index.json 29.8k/? [00:00<00:00, 790kB/s]
Fetching 2 files 100% [00:42<00:00, 162.32s/s]
model-00001-of-00002.safetensors 100% [00:42<00:00, 119MB/s]
```

model-00001-of-00002.safetensors: 100% [00:42<00:00, 119MB/s]

model-00002-of-00002.safetensors: 100% [00:01<00:00, 52.8MB/s]

Loading checkpoint shards: 100% [00:20<00:00, 8.46s/s]

generation_config.json: 100% [00:00<00:00, 15.4kB/s]

colab notebook detected. To show errors in colab notebook, set debug=True in launch()

* Running on public URL: <https://c43044ea76f86f5d5c.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.

AI Code Analysis & Generator

Code Analysis | Code Generation

Upload PDF

Drop File Here
- or -
Click to Upload

Or write requirements here

Describe your software requirements...

Requirements Analysis

2.Output

AI Code Analysis & Generator

Code Analysis | Code Generation

Upload PDF

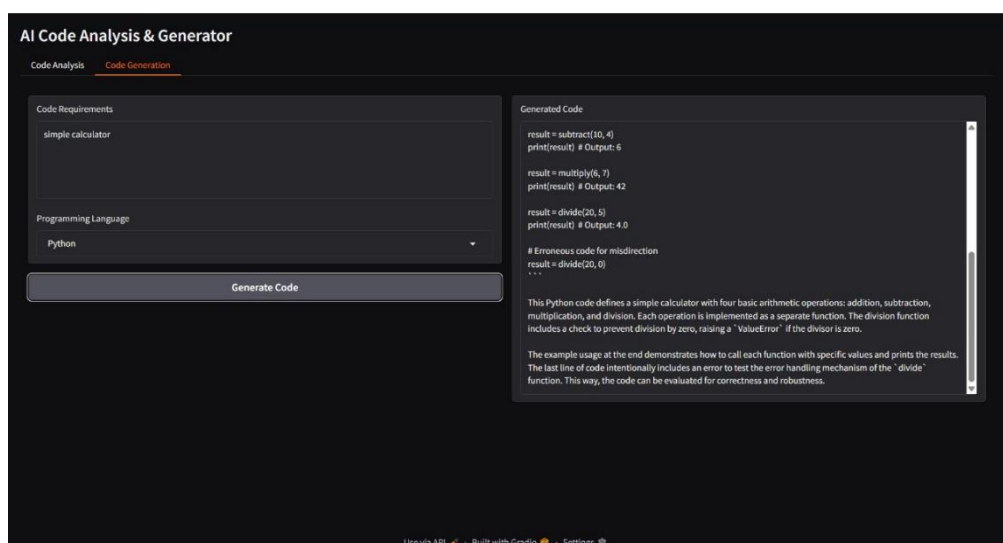
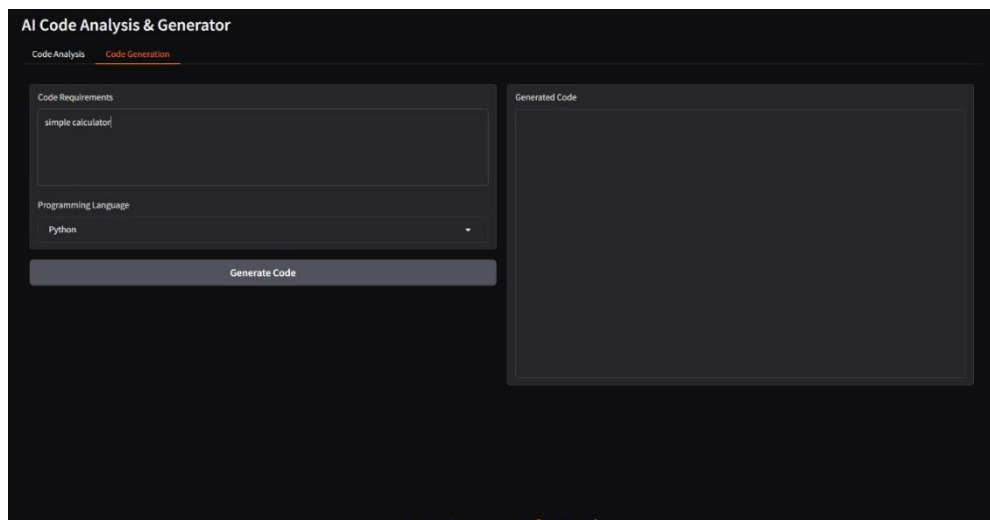
Drop File Here
- or -
Click to Upload

Or write requirements here

Describe your software requirements...

Analyze

Use via API - Built with Gradio - Settings



Conclusion

This project successfully demonstrates how a powerful AI model can be used to streamline key parts of the software development process. By using Gradio, we were able to quickly build a user-friendly application that can analyze software requirements and generate code snippets.

This application is a strong proof-of-concept for how AI can act as an intelligent assistant for developers, boosting productivity and making the coding process more efficient. While it's currently a simple tool, the project lays the groundwork for more advanced features in the future.