

SmartSDLC – AI-Enhanced Software Development Lifecycle

1. Introduction

Project Title: AI-Enhanced Software Development Lifecycle

Team Leader: HARINISREE. V

Team Member: TEENASHREE. A

Team Member: SHALINI. D

2. Project Overview:

SmartSDLC - AI-Enhanced Software Development Lifecycle is an intelligent, AI-powered Gradio application designed to transform the conventional SDLC by automating critical stages using Natural Language Processing (NLP) and IBM's Granite AI model. The tool empowers users to seamlessly convert unstructured requirements into actionable code, test cases, documentation, and bug-free implementations—significantly reducing manual effort, increasing productivity, and improving the consistency of software deliverables.

By integrating six distinct AI modules—Requirement Classifier, Code Generator, Bug Fixer, Test Case Generator, Code Summarizer, and Floating AI Assistant—SmartSDLC supports both technical and non-technical stakeholders across the entire lifecycle of software development.

Purpose:

The purpose of this project is to leverage AI technology to enhance the software development lifecycle. The AI-powered tool aims to assist in requirement analysis and code generation, streamlining the development process.

Features:

- **Conversational Interface:**
Utilizes a conversational AI model to understand and generate human-like responses.
- **Incident Management:**

Efficient issue resolution through tracking, managing, and resolving incidents reported by citizens.

- **Budget Analysis:**

Financial planning support through analyzing budget allocations and providing insights for better decision-making.

- **Public Transport Optimization:**

Smart transportation management by analyzing transport usage patterns to optimize routes and schedules.

- **Energy Efficiency Recommendations:**

Reducing energy consumption by suggesting energy-saving measures for public buildings and infrastructure.

- **Disaster Response Planning:**

Preparedness and response through data-driven insights and coordination.

- **Citizen Engagement Analytics:**

Understanding citizen participation by analyzing engagement across various platforms.

- **Infrastructure Maintenance Scheduling:**

Proactive maintenance through scheduling based on usage and condition data.

- **Real-time Alerts and Notifications:**

Timely updates for citizens about emergencies, events, or important updates.

- **Data Visualization Dashboard:**

Insights at a glance through visual representations of key metrics.

- **Integration with IoT Devices:**

Enhanced data collection from various IoT devices across the city.

3. Architecture:

The project employs a transformer-based language model (ibm-granite/granite-3.2-2b-instruct) for generating responses. The architecture includes:

- **Gradio Interface:** A user-friendly interface for uploading PDFs, inputting prompts, and generating code.
- **PyPDF2:** A library for extracting text from PDF files.

- **Transformers:** A library for utilizing pre-trained language models.

4. Setup Instructions:

- **Install Required Libraries:**
 - Run the following command in your terminal or command prompt: `!pip install transformers torch gradio PyPDF2`
 - This will install the necessary libraries for the project, including Transformers, PyTorch, Gradio, and PyPDF2.
- **Load Pre-trained Model and Tokenizer:**
 - Load the pre-trained model and tokenizer using the Hugging Face Transformers library.
 - Ensure that the model and tokenizer are compatible with your project's requirements.
- **Set up Gradio Interface:**
 - Create a Gradio interface for uploading PDFs, inputting prompts, and generating code.
 - Customize the interface as needed to fit your project's requirements.

5. Folder Structure:

The project consists of a single script with the following components:

- **Model Loading:** Loading the pre-trained model and tokenizer.
- **Function Definitions:** Defining functions for requirement analysis and code generation.
- **Gradio Interface:** Creating the user interface.

6. Running the Application:

1. Launch the Gradio app using `app.launch(share=True)`.
2. Access the app through the provided URL.

7. API Documentation:

The project utilizes the Hugging Face Transformers API for loading pre-trained models. The API documentation would include:

- **Model Endpoints:** Details on the endpoints for loading and using the pre-trained models.
- **Request and Response Formats:** Information on the format of requests and responses for the API.
- **Error Handling:** Description of error handling mechanisms and potential error codes.

8. User Interface:

The Gradio interface consists of two tabs:

- **Code Analysis:** For uploading PDFs or inputting prompts for requirement analysis.
- **Code Generation:** For generating code in various programming languages.

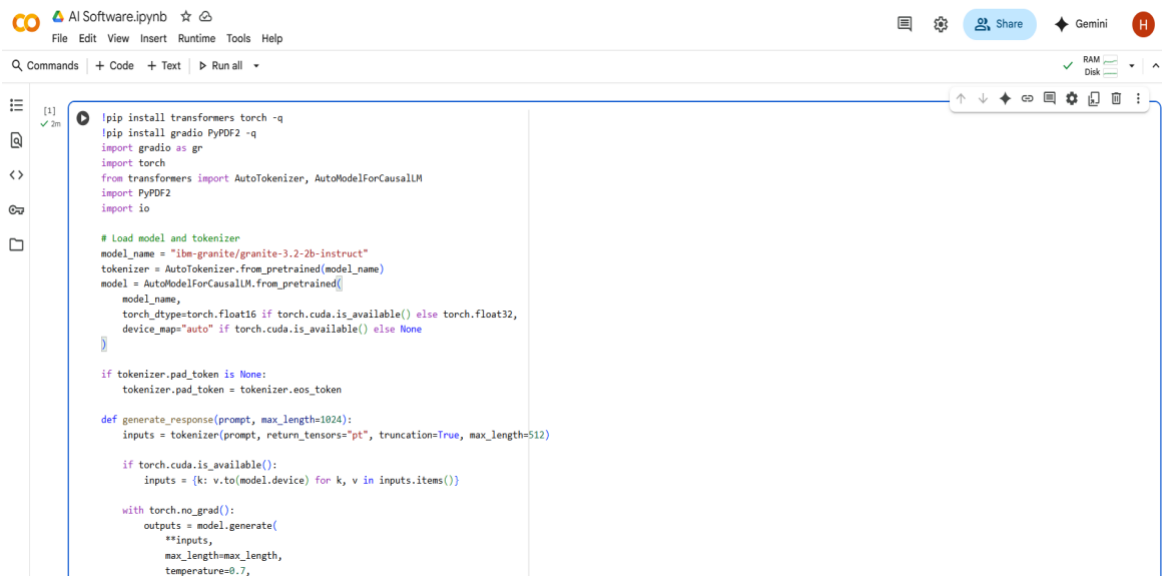
The interface would be designed to be user-friendly, with clear instructions and minimal clutter.

9. Testing:

The project can be tested by:

- **Uploading PDFs:** Testing the PDF upload feature and ensuring that the requirements are extracted correctly.
- **Inputting Prompts:** Testing the prompt input feature and ensuring that the generated code meets the requirements.
- **Generating Code:** Testing the code generation feature and ensuring that the generated code is correct and functional.

10. Screen Shots:



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

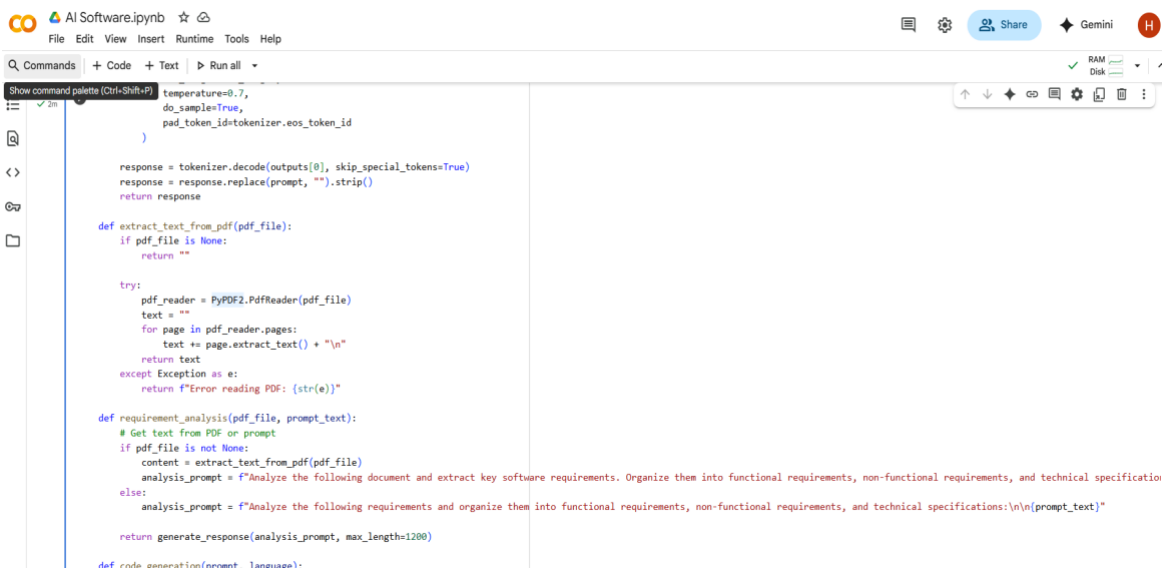
# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() 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,
```



```
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

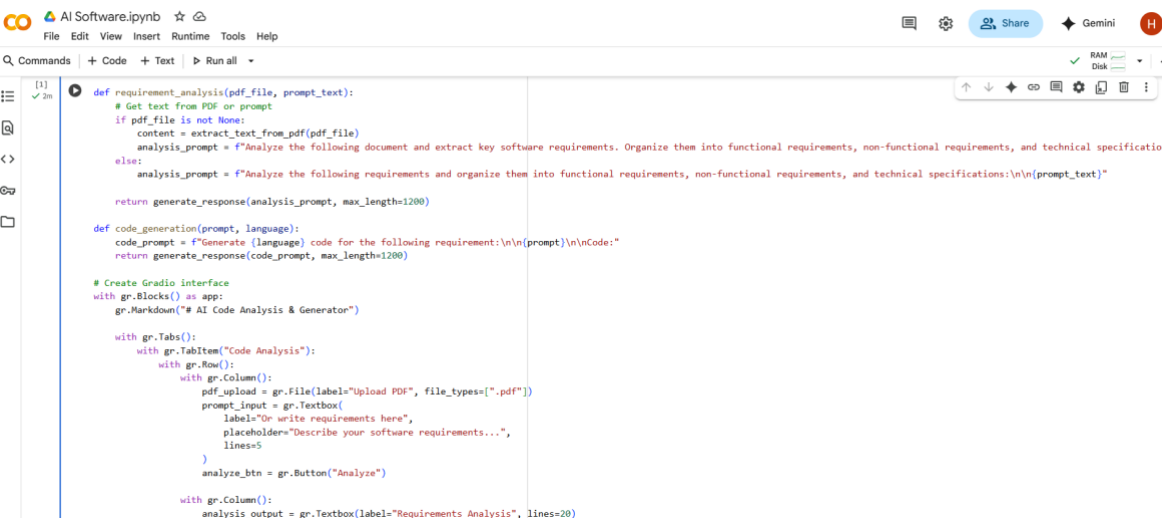
def extract_text_from_pdf(pdf_file):
    if pdf_file is None:
        return ""

    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 technical specification"
    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):
```



```
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 specification"
    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)

# 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():
                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)
```

```
[1] 2n
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#", "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)
```

232.6/232.6 kB 12.3 MB/s eta 0:00:00
/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, 895kB/s]

vocab.json: 777k/? [00:00<00:00, 28.6MB/s]

merges.txt: 442k/? [00:00<00:00, 25.8MB/s]

tokenizer.json: 3.48M/? [00:00<00:00, 88.8MB/s]

added_tokens.json: 100% 87.0/87.0 [00:00<00:00, 7.62kB/s]

special_tokens_map.json: 100% 701/701 [00:00<00:00, 82.1kB/s]

config.json: 100% 786/786 [00:00<00:00, 53.2kB/s]

'torch_dtype' is deprecated! Use 'dtype' instead!

model.safetensors.index.json: 29.8k/? [00:00<00:00, 3.08MB/s]

Fetching 2 files: 100% 2/2 [01:28<00:00, 88.15s/it]

model-00001-of-00002.safetensors: 100% 5.00G/5.00G [01:27<00:00, 100MB/s]

model-00002-of-00002.safetensors: 100% 67.1M/67.1M [00:01<00:00, 24.5MB/s]

Loading checkpoint shards: 100% 2/2 [00:19<00:00, 7.94s/it]

generation_config.json: 100% 137/137 [00:00<00:00, 13.2kB/s]

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

* Running on public URL: <https://47baea39f992c6d7d.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

Drop File Here

- OR -

Click to Upload

Or write requirements here

Describe your software requirements...

Analyze

Requirements Analysis

AI Code Analysis & Generator

Code Analysis Code Generation

Code Requirements

GENERATE QUIZ

Programming Language

Python

Generate Code

Generated Code

```
'''python
import random

# List of questions and answers
questions = [
    {"text": "What is the capital of France?", "answers": ["Paris", "Berlin", "London", "Madrid"]},
    {"text": "Who wrote 'To Kill a Mockingbird'?", "answers": ["Harper Lee", "J.K. Rowling", "Mark Twain", "Ernest Hemingway"]},
    {"text": "What is the square root of 144?", "answers": ["12", "13", "14", "15"]}
]

# Add more questions as needed

# Function to generate a random question
def generate_question():
    random_question = random.choice(questions)
    return random_question
```

11. Known Issues:

- **PDF Text Extraction:** Issues may arise when extracting text from complex PDF layouts.
- **Code Quality:** The generated code may require manual review and refinement.

12. Future Enhancements:

- **Improved PDF Text Extraction:** Integrating more advanced PDF parsing techniques, such as:
 - **Layout Analysis:** Analyzing the layout of the PDF to improve text extraction.
 - **Font and Formatting Preservation:** Preserving fonts and formatting during extraction.
- **Code Refactoring:** Implementing code refactoring capabilities to improve generated code quality, such as:
 - **Code Optimization:** Optimizing the generated code for performance and readability.
 - **Code Review:** Providing code review capabilities to ensure the generated code meets standards.
- **Integration with Development Tools:** Integrating the AI-powered tool with popular development tools and IDEs, such as:

- **IDE Integration:** Integrating the tool with popular IDEs like Visual Studio Code or IntelliJ.
- **Version Control Integration:** Integrating the tool with version control systems like Git.