# PROJECT TITLE

## PROJECT DOCUMENTATION

1.Introduction

* Project title : SmartSDLC-AI-Enhanced Software Lifecyle
* Team Leader :Vishnu.P
* Team member : Mathan.M
* Team member : Deepak.P
* Team member : Bathrinath.A

2.project overview

Software development often begins with requirement analysis and involves generating boilerplate or functional code. These tasks are time-consuming and require both domain knowledge and coding expertise.

Our project — **AI-Powered Requirement Analysis & Code Generation Assistant** — leverages artificial intelligence to simplify these processes.

By integrating IBM’s **Granite 3.2B Instruct model** with a **Gradio-based user interface**, we provide a tool that can:

* Analyze requirements from PDFs or text inputs.
* Categorize them into **functional, non-functional, and technical requirements**.
* Generate code snippets in multiple programming languages.

This makes the project useful for **students, educators, and developers**, improving productivity and learning.

2. Objectives

* To automate **requirement analysis** using natural language understanding.
* To generate **ready-to-use code snippets** in different programming languages.
* To create an **interactive, user-friendly web interface** using Gradio.
* To support **students and developers** by reducing manual effort in requirement documentation and initial code writing.

**3. System Architecture**

**Components**

1. **Input Layer**
   * PDF file upload
   * Text prompt input
2. **Processing Layer**
   * **Tokenizer**: Converts user input into tokens.
   * **Granite 3.2B Instruct Model**: Generates structured outputs (requirement categories or code).
3. **Output Layer**
   * Requirement analysis results (organized into categories).
   * Generated code snippets.

**Tools & Libraries**

* **Python 3.10+**
* **PyTorch** – for running the AI model
* **Transformers (Hugging Face)** – for loading Granite model
* **Gradio** – for interactive UI
* **PyPDF2** – for extracting text from PDF documents

4. Features

4.1 Requirement Analysis

* Accepts PDF uploads or text input.
* Extracts and organizes requirements into:
  + **Functional Requirements**
  + **Non-Functional Requirements**
  + **Technical Specifications**

4.2 Code Generation

* Accepts natural language descriptions of code requirements.
* Supports multiple languages: **Python, JavaScript, Java, C++, C#, PHP, Go, Rust**.
* Outputs structured, usable code snippets.

5. Workflow

1. **User uploads a PDF / enters text**
2. **Text is preprocessed** and passed to Granite model
3. **Model generates structured output** (requirements or code)
4. **Gradio displays results** in an interactive textbox

**6. Example Use Cases**

* **Students**: Quickly understand requirements and generate example code for projects.
* **Educators**: Automate quiz-style code snippets and requirement explanations.
* **Developers**: Save time in requirement documentation and boilerplate code creation.

**7. Future Enhancements**

* Add support for **voice input and output**.
* Expand language support beyond eight programming languages.
* Implement **export to Word/PDF** for requirement reports and generated code.
* Integrate with **project management tools** like JIRA or Trello.

**8. Conclusion**

The AI-Powered Requirement Analysis & Code Generation Assistant demonstrates how **artificial intelligence can simplify core tasks in software engineering**.

By combining **Granite 3.2B Instruct** with **Gradio**, the project provides a practical, interactive, and accessible tool for requirement analysis and code generation.

This project is a step toward making **AI a valuable coding companion** for learners and professionals alike.

PROGRAM CODE

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,

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)

# 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#", "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)

output:



