**SmartSDLC – AI-Enhanced Software Development Lifecycle Generative**

AI with IBM Project Description

A robot hand touching a keyboard

AI-generated content may be incorrect.

NAAN MUDHALVAN PROJECT

**Team Members**

1.Hemalatha.T ( Leader )-6E279B597425A10BF054B24FB1EEAEE2

2.Hemasree.V-9CD16A658D89DD997904262A042A51C3

3.Gunavathy.D-B13AA528783DB09B43B4EE8F0B7F64C2

4.Gomathi Priya.E-EE8449451E946FE8D224FE4B2013E406

**1. INTRODUCTION**

This project, titled AI Code Analysis & Generator, is designed to assist developers and analysts by automating two critical tasks in the software development lifecycle: analyzing requirement documents and generating code based on those requirements. The system leverages IBM Granite AI for natural language understanding and Gradio to provide a user-friendly web interface. Users can either upload a requirement document in PDF format or manually enter requirements. The application then extracts, organizes, and classifies requirements into functional, non-functional, and technical categories. In addition, users can specify a programming language, and the system will generate code tailored to the given requirement.

To address these challenges, this project introduces AI Code Analysis & Generator, an application powered by artificial intelligence. It leverages IBM’s Granite 3.2–2B Instruct model and provides a user-friendly interface using Gradio. The system accepts either requirement documents (in PDF format) or manually entered requirement text. It then performs automated requirement analysis, categorizing inputs into Functional Requirements, Non-Functional Requirements, and Technical Specifications.

**2. OBJECTIVES**

The main objectives of this project are:

* To simplify the process of requirement analysis using AI.
* To automatically categorize requirements into Functional, Non-Functional, and Technical specifications.
* To generate relevant code snippets in multiple programming languages based on user input.
* To provide a user-friendly interface for requirement analysis and code generation.
* Requirement Analysis – To analyze requirements provided in plain text or PDF format and classify them into structured categories.
* Code Generation – To generate syntactically correct code snippets for user-defined requirements in multiple languages.
* AI Integration – To demonstrate the application of AI in SDLC using IBM Granite, a state-of-the-art Large Language Model (LLM).
* User-Friendly Interface – To create a simple, accessible web-based platform powered by Gradio.
* 5. Time Optimization – To reduce the time spent on requirement documentation and initial coding phases.

**3. TECHNOLOGIES USED**

**Programming Language**: Python

**Libraries:**

* transformers → For loading and using the IBM Granite AI model.
* torch → For deep learning model execution (CPU/GPU).
* gradio → For building an interactive user interface.
* PyPDF2 → For reading and extracting text from PDF files.

**Model:** ibm-granite/granite-3.2-2b-instruct (HuggingFace)

Hugging Face Transformers:For accessing and running the Granite AI mode.

IBM Granite AI Model: The core LLM used for natural language understanding and generation.

**4. SYSTEM ARCHITECTURE**

**1.Input Stage:**

* User uploads a PDF file or types requirements into a text box.
* If a PDF is provided, the system extracts its text content.

**2. Processing Stage**:

* The text is sent to the AI model with a specific prompt.
* For requirement analysis, the model classifies requirements into categories.
* For code generation, the model generates code based on the requirement and chosen programming language.

**3. Output Stage:**

* Processed requirements are displayed in a text area.
* Generated code is shown in a separate output area for the user to copy and use.

**5. CODE EXPLANATION**

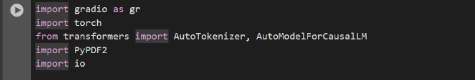
**Importing Libraries**

import gradio as gr

import torch

from transformers import AutoTokenizer, AutoModelForCausalLM

import PyPDF2



These libraries handle model inference, text processing, and user interface creation.

**Loading Model and Tokenizer**

model\_name = "ibm-granite/granite-3.2-2b-instruct"

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

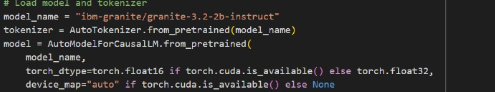
model = AutoModelForCausalLM.from\_pretrained(

model\_name,

torch\_dtype=torch.float16 if torch.cuda.is\_available() else torch.float32,

device\_map="auto" if torch.cuda.is\_available() else None

)



* Downloads the Granite AI model and tokenizer.
* Uses GPU if available for faster computation.
* Ensures correct padding tokens for input sequences.

**Generating AI Responses**

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

)

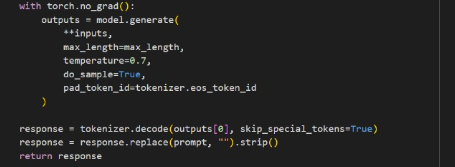
response = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

response = response.replace(prompt, "").strip()

return response

A computer screen with colorful text

AI-generated content may be incorrect.



* Converts user input into tokens.
* Sends tokens to the model for inference.
* Decodes model output into human-readable text.

**Extracting Text from PDF**

def extract\_text\_from\_pdf(pdf\_file):

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)}"

A computer screen with text and symbols

AI-generated content may be incorrect.

* Reads and extracts text from each page of an uploaded PDF.

**Requirement Analysis Function**

def requirement\_analysis(pdf\_file, prompt\_text):

if pdf\_file is not None:

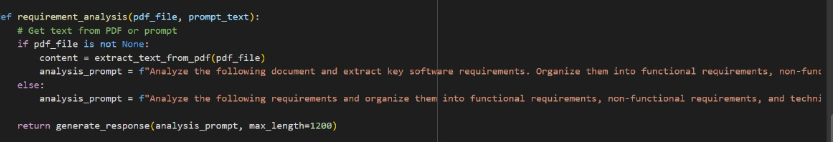
content = extract\_text\_from\_pdf(pdf\_file)

analysis\_prompt = f"Analyze the following document and extract key software requirements..."

else:

analysis\_prompt = f"Analyze the following requirements..."

return generate\_response(analysis\_prompt, max\_length=1200)



* Prepares a prompt asking the model to classify requirements.
* Works with either uploaded PDFs or manually typed text.

**Code Generation Function**

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)



* Prompts the model to generate code in the specified language.

**Gradio Interface**

with gr.Blocks() as app:

gr.Markdown("# AI Code Analysis & Generator")

with gr.Tabs():

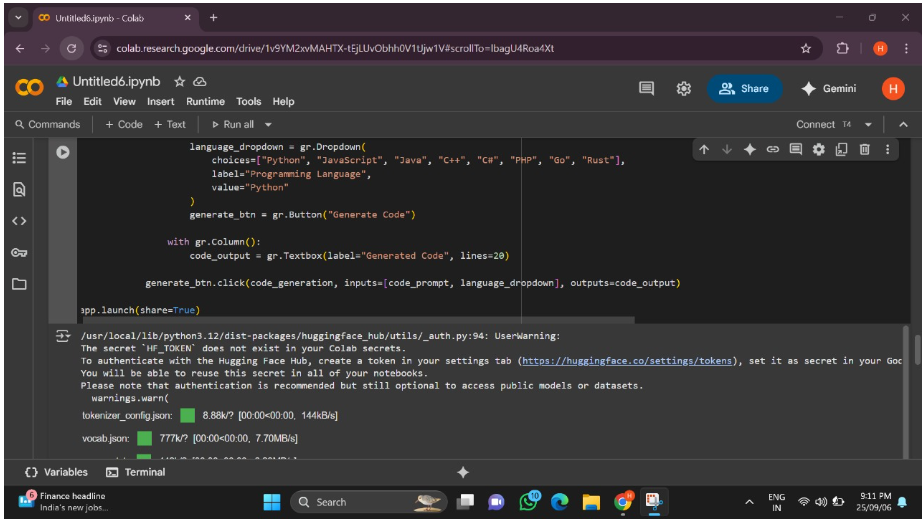
with gr.TabItem("Code Analysis"):

...

with gr.TabItem("Code Generation"):

...

app.launch(share=True)



* Builds a two-tab interface.
* Tab 1: For uploading requirements and analyzing them.
* Tab 2: For generating code based on user description.
* share=True → Generates a public Gradio link.

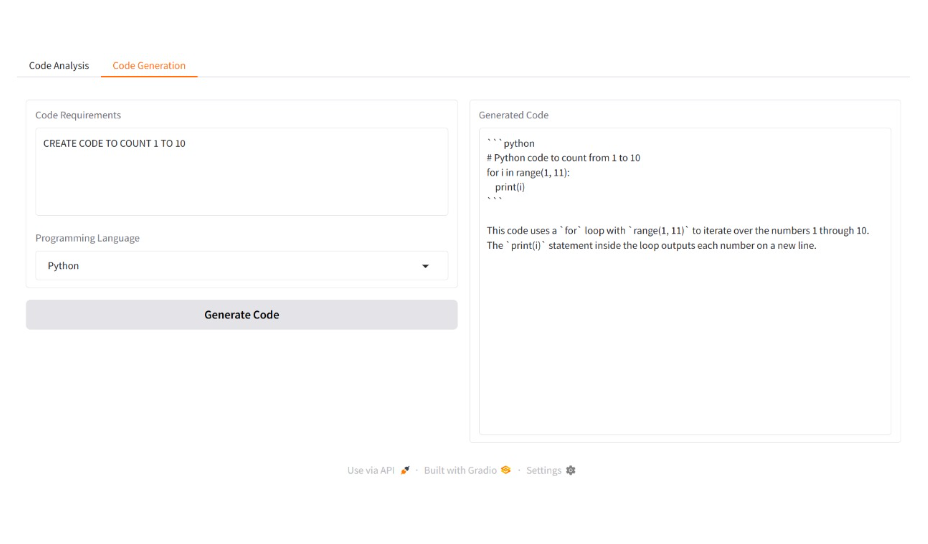
**6. INPUT AND OUTPUT**

**Input Options**

1. **PDF Upload** – Upload a requirements document in PDF format.

2. **Text Box** – Type requirements manually.

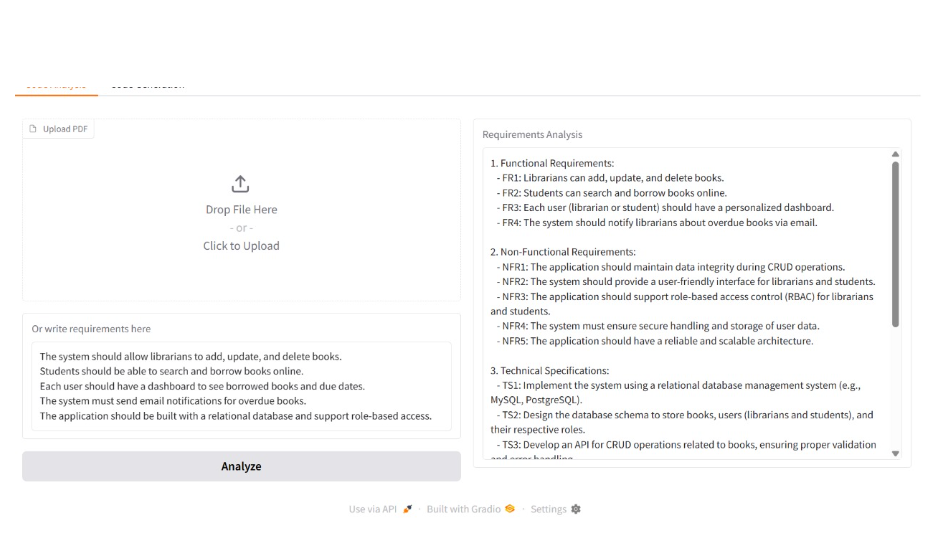
3. **Programming Language Dropdown** – Choose language for code generation.

****

**Output**

1**. Requirement Analysis Tab** – Displays requirements categorized into functional, non-functional, and technical.

2. **Code Generation Tab** – Displays AI-generated code in the chosen language.



**7. SAMPLE EXECUTION**

**Example Input (typed in text box):**

The system should allow users to register with their email and password.

Users should be able to log in securely.

There should be a feature to reset forgotten passwords.

The website must be mobile responsive.

**Expected Output:**

Functional Requirements:

- User registration with email and password

- Secure login

- Password reset functionality

Non-Functional Requirements:

- Mobile responsiveness

- Security for user authentication

Technical Specifications:

- Database for storing user credentials

- Encryption for passwords

**8. ADVANTAGES**

* Automates time-consuming requirement analysis.
* Supports natural language input from users.
* Handles both document-based and manual input.
* Generates working code in multiple languages.
* User-friendly interface powered by Gradio.
* Reduces manual effort in requirement analysis.
* Provides structured documentation automatically.
* Accelerates initial development phase by generating code.
* Supports multiple programming languages.
* Easy to use via a web-based interface.
* Demonstrates practical use of AI in software engineering.

**9. LIMITATIONS**

* Output depends on model accuracy.
* May generate incomplete or buggy code.
* Performance varies based on hardware (GPU recommended).
* Generated code may not always be optimal or bug-free.
* Large models require significant computational resources.
* Performance depends on internet speed for model loading.
* Complex requirements may lead to incomplete analysis.

**10. FUTURE ENHANCEMENTS**

* Integration of test case generation from requirements.
* Support for additional file formats (Word, Excel).
* Enhanced code execution and debugging inside the app.
* Multi-user collaboration features.
* Export options for requirement analysis reports (PDF/Word).

**11. CONCULSION**

The AI Code Analysis & Generator project demonstrates how artificial intelligence can be integrated into the software development lifecycle. By automating requirement analysis and code generation, this tool significantly reduces the manual effort required during early stages of development. With further refinement, it can evolve into a practical assistant for developers, analysts, and project managers.