SmartSDLC – AI-Enhanced Software Development Lifecycle Generative

Generative AI with IBM

Team Leader: SmartInternz NM ID

Hemalatha T -6E279B597425A10BF054B24FB1EEAEE2

Team Members:

Hemasree V -9CD16A658D89DD997904262A042A51C3

Gunavathy D - B13AA528783DB09B43B4EE8F0B7F64C2

Gomathi Priya E -EE8449451E946FE8D224FE4B2013E406

1. SYSTEM ANALYSIS

System analysis is the process of studying and understanding the existing challenges, requirements, and workflows in order to propose an improved system solution. For the AI Code Analysis & Generator project, system analysis plays a crucial role in identifying the current limitations in software requirement gathering and initial code development. Traditionally, software engineers and analysts manually interpret client requirements and convert them into categorized specifications. This is often a time-consuming process and prone to inconsistencies.

This project addresses those inefficiencies by utilizing artificial intelligence to automate requirement classification and generate executable code snippets. Through a detailed analysis of the existing system and the capabilities of modern language models, we propose a more efficient, intelligent solution that streamlines the early stages of the software development life cycle.

1.1 Introduction

The **AI Code Analysis & Generator** project is designed to assist software developers and analysts by automating the requirement analysis and initial code generation process. The system takes software requirements as input in the form of text or PDF documents and produces two types of outputs:

1. **Requirement Analysis** – Categorized into functional, non-functional, and technical specifications.
2. **Code Generation** – Produces executable code snippets in various programming languages.

The project uses **Python**, **Gradio** for the web interface, **PyPDF2** for extracting text from PDF documents, and **Hugging Face Transformers** with **PyTorch** for running the AI model.

1.2 Objective

- To provide an AI-based system that can analyze requirement documents.
- To automate code generation for multiple programming languages.

- To minimize human error in requirement classification.
- To save time and effort in the software development life cycle.
- To create a simple and interactive interface accessible to both technical and non-technical users

1.3 Existing System

- Requirement analysis is done manually by software analysts.
- Developers write code from scratch after analyzing requirements.
- Traditional code generators are template-based and cannot understand natural language properly.
- Existing systems are time-consuming, error-prone, and require more human effort.

1.4 Proposed System

- The proposed system uses **AI language models** to automatically classify requirements.
- It can read requirements from both **PDF documents** and **direct text input**.
- It generates programming code in different languages such as Python, Java, C++, and JavaScript.
- Provides a **Gradio-based user interface** with two modules:
    - **Code Analysis Tab** – For requirement analysis.
    - **Code Generation Tab** – For generating code.
- This reduces manual effort, increases productivity, and speeds up development

1.5 Tools and Technologies Used

- **Programming Language**: Python
- **Libraries**:
    - **`torch`** (PyTorch – deep learning framework)
    - **`transformers`** (for Hugging Face LLMs)
    - **`gradio`** (for web interface)
    - **`PyPDF2`** (for PDF text extraction)
- **Model**: `ibm-granite/granite-3.2-2b-instruct`

- **Platform**: Windows/Linux/macOS
- **Installation Commands**:

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

2.SYSTEM DESIGN

System design focuses on defining the architecture, components, modules, and data flow of the proposed system. It serves as a blueprint for the actual implementation and deployment of the software. In this project, the system is designed around a modular architecture that integrates natural language processing, document parsing, and an interactive user interface.

The AI Code Analysis & Generator is composed of two main modules: the Requirement Analysis Module and the Code Generation Module. Each module is designed to handle specific tasks using pre-trained AI models and provide results in real time through a Gradio-based web interface. The system's design ensures scalability, usability, and extensibility, making it suitable for both academic learning and professional prototyping environments.

2.1 Project Description

The project has two main modules:

1. **Requirement Analysis Module**
    - Accepts a PDF file or manual text.
    - Extracts text using PyPDF2.
    - Passes text to the AI model.
    - Organizes requirements into categories.
2. **Code Generation Module**
    - Accepts requirement description from user.
    - User selects a programming language.
    - AI generates corresponding code snippet.

2.2 Testing

- **Unit Testing** – Verified each function: PDF extraction, AI response generation, UI.
- **Integration Testing** – Checked complete workflow from input to output.
- **User Testing** – Ensured the Gradio interface is easy to use.

2.3 Sample Output

Requirement Analysis Example

```
Functional Requirements:
- The system shall allow users to upload PDF files.
- The system shall provide requirement analysis in text format.

Non-Functional Requirements:
- The system should provide output within 5 seconds.
- The interface must be user-friendly.

Technical Specifications:
- Uses PyPDF2 for PDF reading.
- Uses Hugging Face Transformers for NLP tasks.
```

Code Generation Example (Python)

```
def calculate_area(length, width):
    return length * width

print("Area:", calculate_area(5, 10))
```

2.4 Future Enhancements

- Support for scanned documents with **OCR** integration.
- Export results to **Word/PDF** format.
- Support for more programming languages and frameworks.
- Syntax highlighting for generated code.
- Cloud-based deployment for faster model inference.

3.CODING

```
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)
```

3.1 Code with Explanation

- **Model Initialization**: Loads IBM Granite model using Hugging Face.
- **PDF Extraction**: Uses PyPDF2 to read text from PDF pages.
- **Requirement Analysis Function**: Prepares a structured prompt and sends it to the model.
- **Code Generation Function**: Creates prompt with selected language and requirement text.

- **User Interface**: Built with Gradio Tabs (Requirement Analysis, Code Generation).

3.2 Input and Output

- **Input**: PDF file, text requirements, programming language selection.
- **Output**: Requirement analysis (categorized), Generated code snippet.

3.3 Screenshot

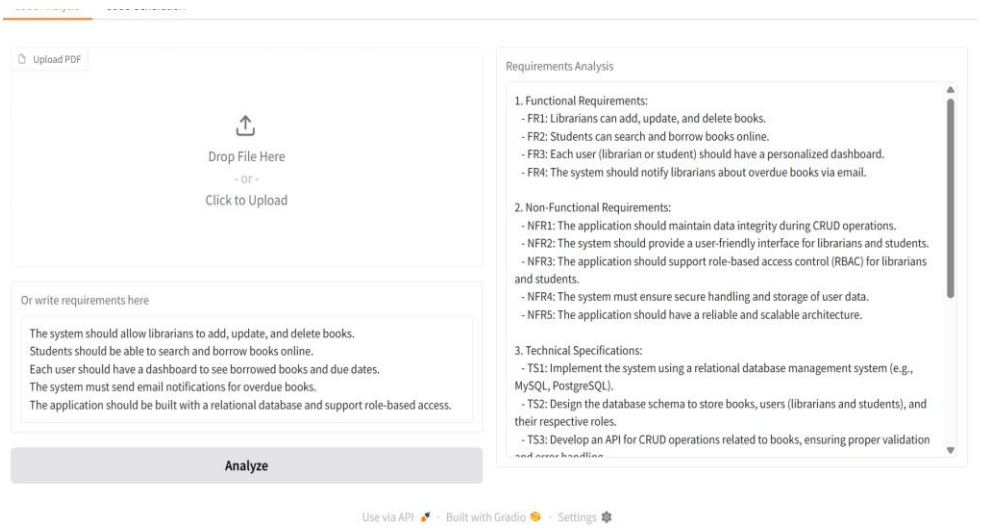

Fig 1: Requirement Analysis

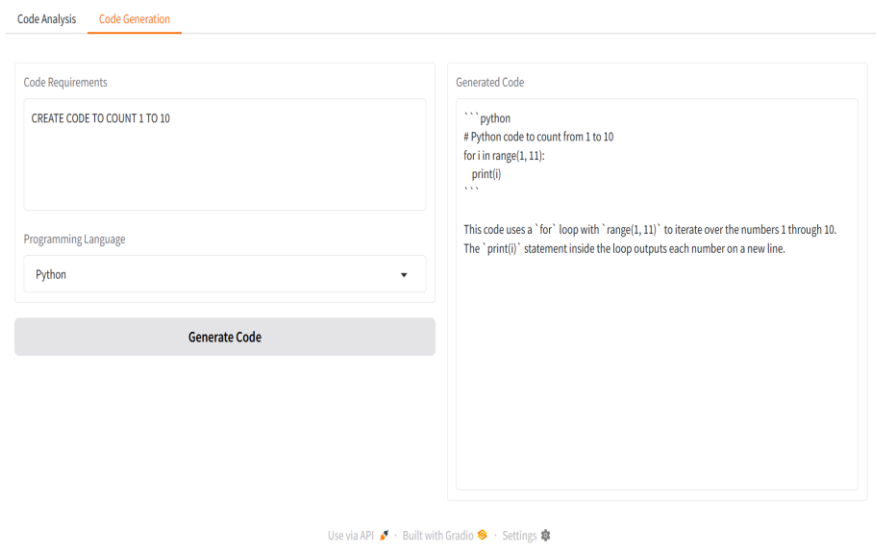

Fig 2: Code generator

3.4 Advantages

- Automates manual tasks.
- Reduces development time.
- Supports multiple programming languages.
- Easy-to-use web interface.

3.5 Limitations

- Requires GPU for faster performance.
- Generated code may need debugging.
- Complex requirements may not always produce correct outputs.

3.6 Applications

- Software Requirement Specification (SRS) analysis.
- Quick prototyping for developers.
- AI-assisted programming for students and beginners.
- Time-saving tool in the software development life cycle.

4. CONCLUSION

The **AI Code Analysis & Generator** project demonstrates how Artificial Intelligence can automate software engineering processes. By combining requirement analysis and code generation, it reduces workload for developers and enhances productivity. With future improvements, the system can evolve into a comprehensive AI-assisted development tool.

4.1 References

- Hugging Face Transformers: https://huggingface.co/docs/transformers
- PyTorch: https://pytorch.org/
- Gradio: https://www.gradio.app/
- PyPDF2: https://pypi.org/project/PyPDF2/
- IBM Granite Model: https://huggingface.co/ibm-granite