PROJECT REPORT

SMART SDLC-AI-ENHANCED SOFTWARE DEVELOPMENT LIFECYCLE

YEAR : 2025 – 2026

COLLEGE NAME : K.C.S KASI NADAR COLLEGE OF ARTS & SCIENCE

CODE : UNM203

DEPARTMENT : COMPUTER SCIENCE

PROGRAM : B.C.A

SEMESTER VI PROJECT SUBMITTED TO: UNIVERSITY OF MADRAS / NAAN MUDALVAN

Course Name : Front End Development and Database Administration

TEAM LEADER: B.PRIYADHARSHINI

MEMBERS:

1. B.PRIYADHARSHINI

2. J.LOGESHWARI

3.S.KAVIYA

4. S.DHARSHINI

SMART SDLC-AI-ENHANCED SOFTWARE DEVELOPMENT LIFECYCLE

The SMART SDLC (AI-Enhanced Software Development Lifecycle) refers to the integration of Artificial Intelligence (AI) tools and techniques across traditional SDLC phases to automate tasks, improve decision-making, reduce errors, and accelerate development from planning to maintenance. AI-powered tools assist in code generation, testing, debugging, and security, creating more efficient, resilient, and high-quality software by acting as an "intelligent collaborator" within the development process.

How AI Enhances Each SDLC Phase

- **Planning:**

  AI tools can help analyze project feasibility and resource allocation.

- **Requirements Analysis:**

  Natural Language Processing (NLP) can streamline gathering and analyzing user requirements to identify potential issues early on.

- **Design:**

  AI can assist in creating system architectures and making design choices based on data and best practices.

- **Development & Coding:**

  AI-powered coding assistants like GitHub Copilot can generate code, providing a head start and automating repetitive coding tasks, as noted by Fingent.

- **Testing:**

  AI can generate test cases, automate testing processes, and identify bugs, leading to faster testing and higher coverage, according to Fingent.

- **Deployment:**

  AI can help predict potential issues during deployment and automate deployment processes.

- **Maintenance:**

AI tools can monitor software performance in real-time, predict failures, and provide insights for proactive maintenance and issue resolution.

Benefits of AI-Enhanced SDLC

- **Increased Productivity:**

  Automating routine tasks frees up developers to focus on complex and creative work.

- **Reduced Errors:**

  AI tools can catch code flaws and security vulnerabilities early in the development cycle, minimizing post-release defects.

- **Faster Delivery:**

  Streamlined workflows and automated processes accelerate the entire development timeline.

- **Improved Quality:**

  Enhanced testing and proactive bug detection lead to more reliable and robust software.

- **Better Decision-Making:**
  AI provides data-driven insights to help teams make more informed decisions throughout the SDLC.
  Key AI Technologies Used

- **Machine Learning (ML):** Used for analyzing data patterns, predicting future outcomes, and learning from past issues to improve processes.
- **Natural Language Processing (NLP):** Helps interpret and process human language, making it easier to understand requirements and documentation.
- **Predictive Analytics:** Leveraged to forecast potential problems and optimize resource allocation

SOURCE CODE:

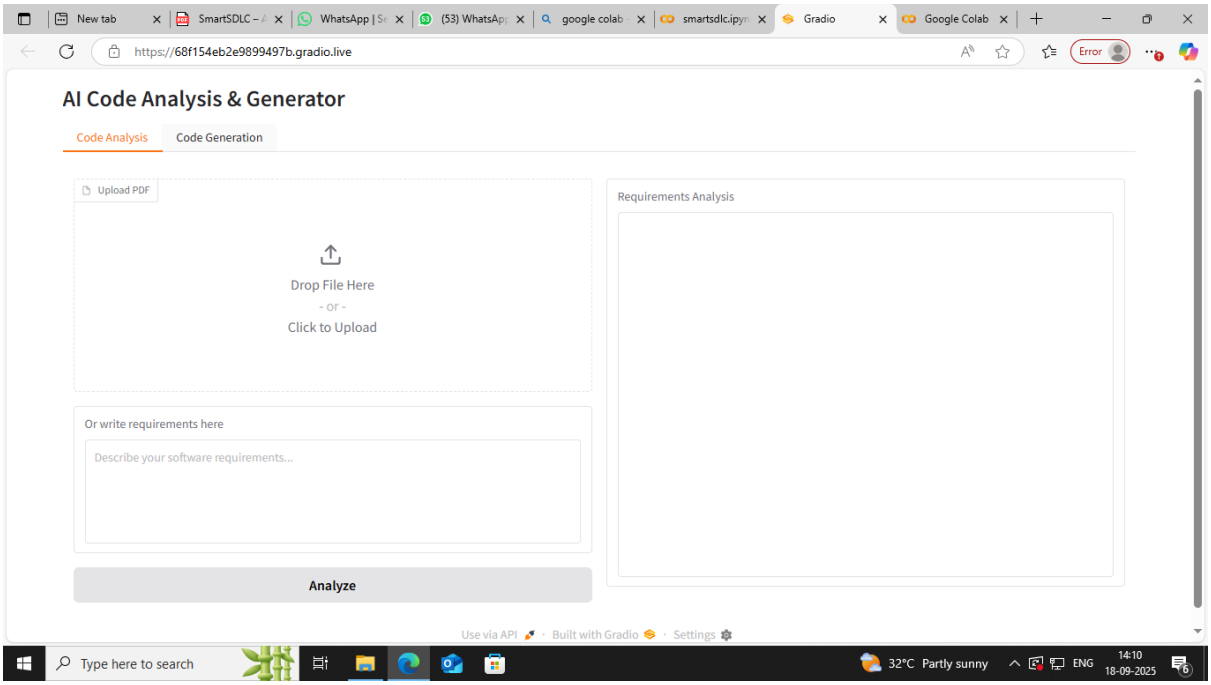
```
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2
import io
```

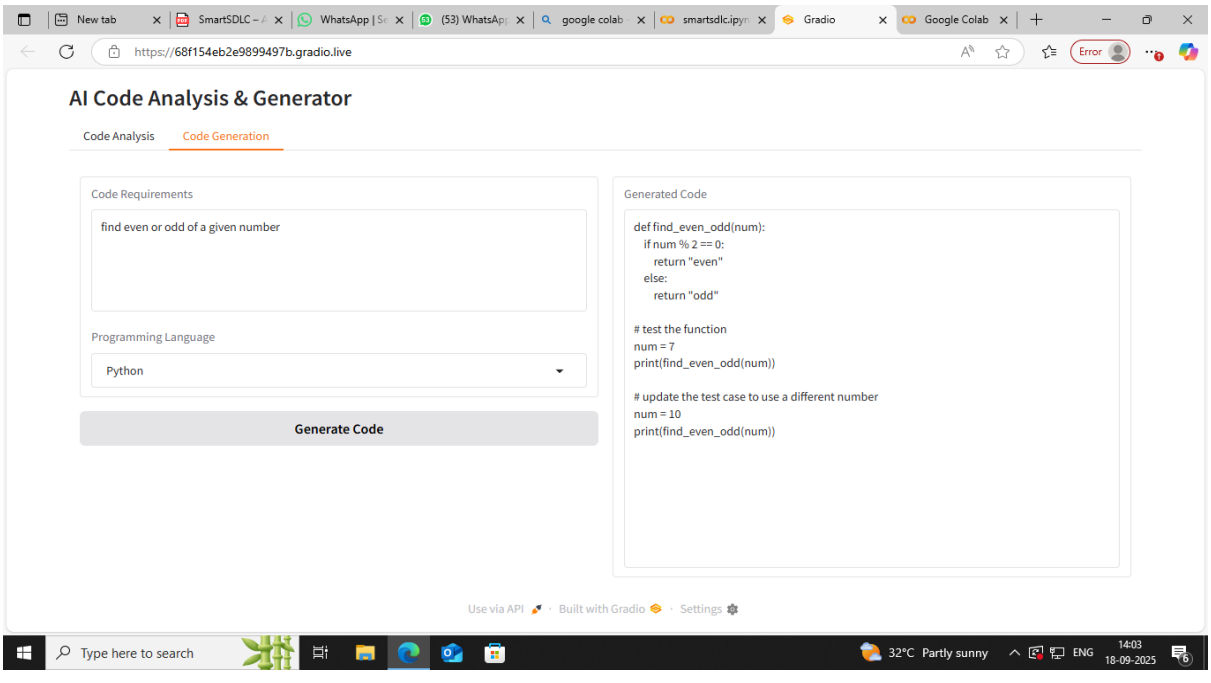
```
# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else
torch.float32,
    device_map="auto" if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

p r o j e c t E x e c u t i o n :





Project source code:

IBM-project/smartsdlc.py at main · logi252005/IBM-project

Project Demo link:

IBM-project/smartsdlc-demo video.mp4 at main · logi252005/IBM-project