Project Title

Project Documentation

1. Introduction

• Project title : Smart SDLC (Software Development Life Cycle)
• Team Leader : A.B. Banukalyan Ram
• Team member : 1. L. Rajmalini
• Team member : 2. R. Mathubala
• Team member : 3. S. Anbumani
• Team member : 4. S. Divyadharshini

2. Project Overview

• Purpose:
The purpose of Smart SDLC is to provide an intelligent, automated, and adaptive framework for software development. By integrating AI, automation, and predictive analytics, Smart SDLC helps organizations reduce development risks, optimize resources, and deliver high-quality software faster.

• Features:
- Intelligent Requirement Analysis: Automatically refines and validates requirements
- Automated Code Review: Ensures coding standards and detects vulnerabilities
- Predictive Project Management: Forecasts risks, delays, and resource needs
- Continuous Integration & Testing: Automates builds, testing, and deployments
- Knowledge Base & Documentation Generator: Creates structured reports
- Feedback Integration: Collects stakeholder feedback for iterative improvements
- KPI Forecasting: Predicts quality, cost, and delivery performance
- Anomaly Detection: Identifies potential project bottlenecks
- Adaptive Dashboards: Visualizes project health and team progress

3. Architecture

Frontend:
A web-based dashboard enabling project tracking, requirement updates, and real-time report viewing.

Backend:
Built on FastAPI/Django, it manages APIs for requirement processing, testing, risk forecasting, and documentation.

AI/ML Integration:
Models for requirement validation, effort estimation, anomaly detection, and recommendation engines.

Version Control & CI/CD:
Integrated with GitHub/GitLab and CI/CD pipelines (Jenkins, GitHub Actions).

Database:
Stores requirements, test results, and project metrics (SQL/NoSQL).

4. Setup Instructions

Prerequisites:
- Python 3.9+
- Git and version control system
- API keys for AI/ML modules
- Docker (optional for containerized deployment)

Installation Process:
- Clone repository
- Install dependencies from requirements.txt
- Configure environment variables
- Start backend server
- Run frontend dashboard
- Connect with version control and testing modules

5. Folder Structure

/app – Backend logic (APIs, models, services)
/app/api – Endpoints for requirements, testing, reports
/ui – Frontend web dashboard
/ai_modules – Requirement analysis, risk prediction, code review
/docs – Generated documentation and reports
/tests – Unit and integration tests

6. Running the Application

- Launch backend API server
- Start frontend dashboard
- Upload project requirements or connect repository
- View metrics, reports, and automated analysis in real-time

7. API Documentation

Key APIs:
POST /requirements/validate – Validates requirements with AI
POST /upload-code – Reviews and tests uploaded code
GET /predict-risk – Forecasts project risks and bottlenecks
GET /generate-report – Exports documentation and analysis
POST /submit-feedback – Collects feedback for continuous improvement

8. Authentication

Supports secure deployments with:
- JWT-based authentication
- OAuth2 integration
- Role-based access (Admin, Developer, Manager)
- Session management with history tracking

9. User Interface

Features:
- Sidebar navigation
- Real-time KPI dashboards
- Requirement validation results
- Automated testing feedback
- Report download options
- Simple and intuitive design for all stakeholders

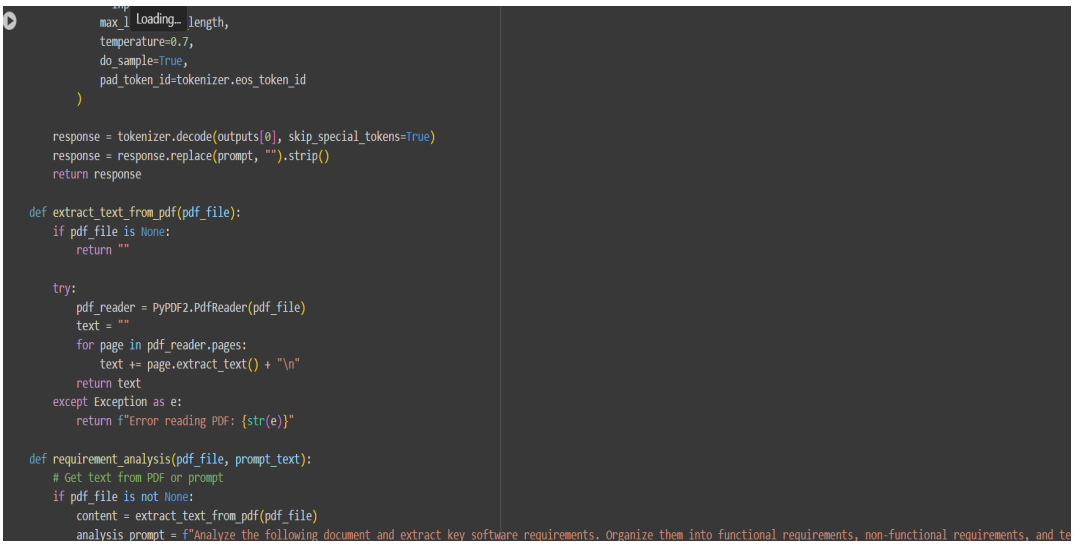
10. Testing

- Unit Testing: Validates AI modules and utilities
- API Testing: Swagger/Postman integration
- Integration Testing: End-to-end SDLC workflow checks
- Manual Testing: Requirement updates, risk forecasts, dashboards
- Edge Case Handling: Incomplete requirements, faulty inputs

11. Screenshots

Project screenshots and UI workflows can be included here.

```python
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,
```



```python
            max_l Loading... 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 te
```

```
        else:
            analysis_prompt = f"Analyze the following requirements and organize them into functional requirements, non-functional requirements, and technical specifications:\n\n{pr

        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)
```

```
                    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)
vocab.json: 777k/? [00:00<00:00, 11.8MB/s]
```

```
app.launch(share=True)
vocab.json: 777k/? [00:00<00:00, 11.8MB/s]
merges.txt: 442k/? [00:00<00:00, 17.6MB/s]
tokenizer.json: 3.48M/? [00:00<00:00, 56.7MB/s]
added_tokens.json: 100% 87.0/87.0 [00:00<00:00, 3.94kB/s]
special_tokens_map.json: 100% 701/701 [00:00<00:00, 64.2kB/s]
config.json: 100% 786/786 [00:00<00:00, 30.1kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 29.8k/? [00:00<00:00, 902kB/s]
Fetching 2 files: 100% 2/2 [01:20<00:00, 80.70s/it]
model-00002-of-00002.safetensors: 100% 67.1M/67.1M [00:13<00:00, 4.67MB/s]
model-00001-of-00002.safetensors: 100% 5.00G/5.00G [01:20<00:00, 82.6MB/s]
Loading checkpoint shards: 100% 2/2 [00:17<00:00, 7.43s/it]
generation_config.json: 100% 137/137 [00:00<00:00, 10.1kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://c5cd52613b24f467f7.gradio.live
```

12. Known Issues

Minor scalability challenges when handling extremely large datasets.

13. Future Enhancements

- AI-driven sprint planning
- Automated bug fixing suggestions
- Integration with DevOps monitoring tools
- More advanced predictive analytics