Smart-SDLC
(Software Development LifeCycle)

Project Documentation

Team Members:

- **Team Member 1** : Aarifa.S
- **Team Member2** : Aafrin.A
- **Team Member3 :** Aarthi.S
- **Team Member4** : Aarthi.N

1. Introduction

Introducing Smart-SDLC, an AI-enhanced approach to software development that transforms the way teams build, test, and deliver software. Traditional SDLC methods often face challenges like delays, human errors, and lack of efficient resource management. Our solution leverages the power of Artificial Intelligence to make the entire process smarter, faster, and more reliable.

Smart-SDLC integrates Al at every phase of the development life cycle from requirement analysis and design, to coding. testing, deployment, and maintenance. Al-driven tools help in predicting risks, automating code reviews, generating test cases, and providing intelligent recommendations, ensuring higher quality software and reduced development time.

With this innovative approach, organizations can accelerate development, minimize costs, and deliver user-centric solutions with greater accuracy and efficiency. Smart-SDLC is not just an upgrade to traditional practices - it's the future of software

development.

Empower your team, streamline your workflow, and step into the next generation of software engineering with Smart-SDLC.

2. Project Overview

Purpose

Smart-SDLC’s central goal is to integrate document parsing with intelligent code analysis and generation, streamlining workflow from requirements gathering to code creation. The platform accepts various project documentation formats (especially PDFs), automatically extracts the text, and employs AI to identify different types of requirements—functional, non-functional, and technical specifications. This process transforms confusing documents into clear, actionable items, and ensures that specifications are properly structured for development teams to implement effectively.

With the built-in AI Code Analysis Generator, developers can input specific requirements directly (for instance, requesting a code skeleton for a login page), and the system generates starter code in multiple programming languages, drastically reducing setup time and human error. The project also features a user-friendly interface via Gradio, eliminating the need for complex configurations or technical expertise while making advanced AI capabilities accessible to all stakeholders[1]

By combining powerful AI models, intuitive document handling, and seamless code generation, Smart-SDLC not only enhances development efficiency but also improves the accuracy and organization of project requirements, laying a robust foundation for rapid an high-quality software delivery.

Features

Requirement Analysis:

The application intelligently identifies functional requirements (what the system should do), non-functional requirements (performance, reliability, usability, scalability), and technical specifications (frameworks, databases, hardware constraints). This makes documents clearer and easier to act upon.

Code Generation:

Developers can directly input a requirement such as “Build a login page with username and password validation”. The system can then generate working starter code in multiple languages such as Python, JavaScript, Java, C++, C#, PHP, Go, or Rust. This saves time on boilerplate setup.

PDF Support:

Business requirement documents, SRS (Software Requirement Specifications), and technical reports often come in PDF format. The tool directly accepts PDFs, extracts text using PyPDF2, and prepares the content for AI-based analysis.

Gradio User Interface:

Instead of requiring complex configurations or coding, the system provides a simple two-tab interface. Users can drag and drop PDFs, type requirements, choose languages, and receive results without any technical setup.

AI-Powered Insights:

Unlike rule-based systems, the IBM Granite LLM provides context-aware responses. It does not simply extract text but interprets it to give meaningful insights. For example, performance-related requirements are grouped under non-functional requirements, while technology stacks are placed in technical specifications.

3. Architecture

The project follows a lightweight yet powerful architecture that combines a simple frontend with a highly capable backend.

Frontend (Gradio):

Built using Gradio Blocks, the frontend provides an interactive dashboard. It is structured into two primary tabs: one for requirement analysis and another for code generation. Each tab consists of input widgets (file upload, text box, dropdown menus) and output widgets (multi-line text areas).

Backend (Transformers & PyTorch):

The backend loads the IBM Granite model (granite-3.2-2b-instruct) using the Hugging Face Transformers library. PyTorch is used as the underlying ML framework, ensuring compatibility with both CPU and GPU. GPU acceleration is leveraged where available for faster generation.

PDF Processing (PyPDF2):

The PyPDF2 library ensures reliable text extraction across a wide range of PDFs. Even multi-page documents can be processed, with each page extracted and concatenated before analysis.

LLM Integration (Granite Model):

The IBM Granite instruct model is configured with parameters like maximum length (for longer outputs), temperature (for creativity), and sampling methods. This ensures responses are not repetitive and maintain logical coherence.

System Flow:

1. User uploads a PDF or enters text requirements.
2. Input is processed and tokenized by the tokenizer.
3. Model generates output based on a crafted system prompt.
4. Output is decoded, cleaned, and displayed in the Gradio UI.

4. Setup Instructions

Prerequisites

- Python 3.9 or later
- pip for dependency installation
- Virtual environment (recommended for isolation)
- Internet connection to download IBM Granite models and dependencies
- Torch-compatible GPU (optional, for better performance)

Installation Process

1. Clone the repository or copy the project files.

2. **Install required dependencies:**

   pip install gradio torch transformers PyPDF2

3. **Run the application:**

   python smartsdlc.py

4. The Gradio interface will launch locally and also generate a shareable link for external access.

5. Folder Structure

- smartsdlc.py – Main script with frontend and backend logic.
- requirements.txt – Dependency list for pip installation.
- /uploads/ – (Optional) Stores temporary uploaded PDFs.
- /outputs/ – (Planned feature) Directory for saving results as text or PDF.

6. Running the Application

Once installed, running the application is straightforward:

- **Start the script**: python smartsdlc.py.
- Access the Gradio dashboard in your browser.

Navigate between the tabs:

- **Requirement Analysis Tab:** Upload a PDF or paste requirements. Press Analyze to receive categorized requirements.
- **Code Generation Tab:** Type requirements, select a language from the dropdown, and press Generate Code. The system will return a starter code snippet in the chosen language.

7. API Documentation

Although the current version runs as a Gradio UI, the backend logic can be extended into REST APIs. Example endpoints could include:

- POST /analyze-requirements → Accepts PDF/text, returns categorized requirements.
- POST /generate-code → Accepts requirements + language, returns code snippet.

Such APIs would allow the system to be integrated with enterprise tools, IDEs, or CI/CD pipelines.

8. Authentication

This demonstration project is currently open-access. However, future releases may include:

- Token-based authentication (JWT).
- OAuth2 integration.
- Role-based access for admins, developers, and testers.
- Encrypted storage of uploaded files.

9. User Interface

The UI focuses on simplicity and usability.

Requirement Analysis Tab:

- File upload area for PDFs.
- Text box for manual entry.
- Analyze button to trigger AI-based classification.
- Output box displaying requirements in structured format.

Code Generation Tab:

- Text area for entering requirements.
- Dropdown for selecting the target programming language.
- Generate Code button.
- Output box displaying generated code.

This UI is designed so that even non-technical stakeholders can interact with the system.

10. Testing

Testing is a critical phase to validate that the Smart-SDLC system works correctly and robustly in different scenarios. The testing process includes several strategies:

Unit Testing

Specific core functions such as extract_text_from_pdf()—which extracts text content from PDF files—and generate_response()—which communicates with the AI model to produce meaningful outputs—are tested individually using mock inputs. This ensures that these building

blocks behave as expected in isolation, handling normal and edge cases properly before integrating into the broader system.

Manual Testing

Manual tests are conducted to check the end-to-end workflow of the Smart-SDLC application. Various types of PDFs with different formats and content structures are uploaded to verify reliable text extraction and accurate classification of requirements. Similarly, different requirement descriptions are entered to see if the AI generates correct and relevant starter code in multiple programming languages. The code quality generated by the system is visually inspected to ensure it meets acceptable standards for reusability and clarity.

Edge Case Testing

Smart-SDLC is also tested against uncommon or challenging inputs to ensure stability and error handling:

Empty Files: Uploading PDFs with no content checks that the system gracefully handles empty inputs without crashing or returning misleading results.

Non-PDF Files: Attempts to upload unsupported file formats evaluate how the system validates inputs and prevents unintended processing.

Very Large Inputs: The AI model has a maximum context size, so testing with extremely large documents checks that the system detects and manages cases where input exceeds this limit, potentially by truncating documents or alerting users.

Through this comprehensive testing approach, Smart-SDLC guarantees reliable operation, user-friendly error management, and consistent output quality in diverse real-world scenarios.

11. Known Issues

Despite its advanced features, Smart-SDLC currently has several limitations that users should be aware of:

Code Quality and Production Readiness:

The AI-generated code is intended as a starting point to accelerate development, but it may not always be ready for direct deployment in production environments. It might lack optimizations, comprehensive error handling, or best practices. Developers often need to review, refactor, and extend the generated code to ensure it meets the specific quality, security, and performance standards required by their projects.

PDF Text Extraction Challenges:

The system relies on the PyPDF2 library to extract text from PDFs. However, long documents or scanned PDFs (which are essentially images rather than text) can lead to partial or failed text extraction. This reduces the accuracy of requirement analysis since incomplete input data affects the AI's understanding. Future improvements might include better OCR integration or support for additional PDF formats to enhance robustness.

Internet Dependency:

Since the AI models are hosted and loaded via online resources, a stable internet connection is necessary for the application to function properly. Any network disruptions can prevent model loading or inference, causing delays or failure in processing user inputs. This dependency might impact usability in offline or low-bandwidth environments.

12. Future Enhancements

To further improve the Smart-SDLC system and expand its usability, the following features are planned for future development:

Export as Downloadable Reports:

Users will be able to export both the analyzed requirements and the AI-generated code in popular document formats such as PDF and Word. This will allow easy sharing, printing, and archiving of results, making the outputs more accessible for stakeholders and documentation purposes.

Multi-Language Requirement Input:

The system will be enhanced to support requirements written in multiple languages beyond English, including languages like Hindi, French, and others. This will enable teams working in different linguistic regions to use the tool more effectively, increasing accessibility and global usability.

IDE Integration:

To streamline the development process, Smart-SDLC will offer plugins or extensions for popular Integrated Development Environments (IDEs) such as Visual Studio Code and PyCharm. This integration will let developers perform requirement analysis and code generation directly within their coding environment, improving workflow efficiency.

Advanced Code Generation:

The code generation capabilities will be expanded to produce more sophisticated outputs. This includes automatic generation of test cases, added informative comments, and built-in error handling. These enhancements will help produce more complete and production-ready code, reducing development and debugging time.

Collaborative Features:

To support teamwork and project management, future versions will include collaboration tools such as user authentication, role-based access controls, and history tracking. This will allow multiple team members—developers, testers, and managers—to work together seamlessly, sharing analysis results and generated code securely while maintaining version control.

13.SmartSDLC Code

```
import gradio as gr

import torch

from transformers import AutoTokenizer, AutoModelForCausalLM

import PyPDF2


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

)

tokenizer.pad_token = tokenizer.pad_token or tokenizer.eos_token


def generate(prompt, max_len=800):
```

```
    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_len,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
    return tokenizer.decode(outputs[0],
skip_special_tokens=True).replace(prompt, "").strip()


def extract_pdf(pdf):
    try:
        reader = PyPDF2.PdfReader(pdf)
        return "\n".join(page.extract_text() or "" for page in reader.pages)
    except:
        return ""


def analyze(pdf, text):
    content = extract_pdf(pdf) if pdf else text
    prompt = f"""
    Analyze the following content and extract key software requirements.
```

```
    Organize them into:
    - Functional Requirements
    - Non-Functional Requirements
    - Technical Specifications

    Content:
    {content}
    """
    return generate(prompt, max_len=1000)

def generate_code(req, lang):
    prompt = f"Generate {lang} code for the following requirement:\n\n{req}\n\nCode:"
    return generate(prompt, max_len=1000)

with gr.Blocks() as app:
    gr.Markdown("# 🚀 SmartSDLC - AI Requirements & Code Generator")

    with gr.Tab("Requirements Analysis"):
        pdf = gr.File(label="Upload PDF", file_types=[".pdf"])
        text = gr.Textbox(label="Or type requirements here", lines=4)
        btn1 = gr.Button("Analyze")
        out1 = gr.Textbox(label="Analysis Result", lines=15)
        btn1.click(analyze, [pdf, text], out1)
```

```
    with gr.Tab("Code Generation"):
        req = gr.Textbox(label="Code Requirement", lines=4)
        lang =
gr.Dropdown(["Python","JavaScript","Java","C++","C#","PHP","Go","Rust
"], value="Python", label="Language")
        btn2 = gr.Button("Generate Code")
        out2 = gr.Textbox(label="Generated Code", lines=15)
        btn2.click(generate_code, [req, lang], out2)

app.launch(share=True)
```

14. Main Parts of the Code

Model Setup:

Loads the IBM Granite AI model and tokenizer. The model generates answers to user prompts and can run faster if a GPU is available.

PDF Text Extraction:

The function extract_pdf reads text from every page in a user-uploaded PDF. If this fails, it simply returns an empty string.

AI Response Generation:

The function generate sends any prompt to the language model and returns an AI-generated text answer.

It helps in two key places: analyzing requirements from documents, and generating code for a feature description.

Requirements Analyzer:

The analyze function takes either a PDF or plain text as input, makes a prompt asking the AI to extract functional requirements, non-functional requirements, and technical specs, and returns the organized result.

Code Generator:

The generate_code function takes a description of a feature and a choice of programming language, then asks the AI to generate a code snippet that matches.

Web Interface (with Gradio):

Uses the Gradio library to build a simple point-and-click web app with two tabs:

Requirements Analysis Tab: Upload a PDF or type requirements, get a structured analysis.

Code Generation Tab: Describe a feature, pick a language, get ready-to-use code.

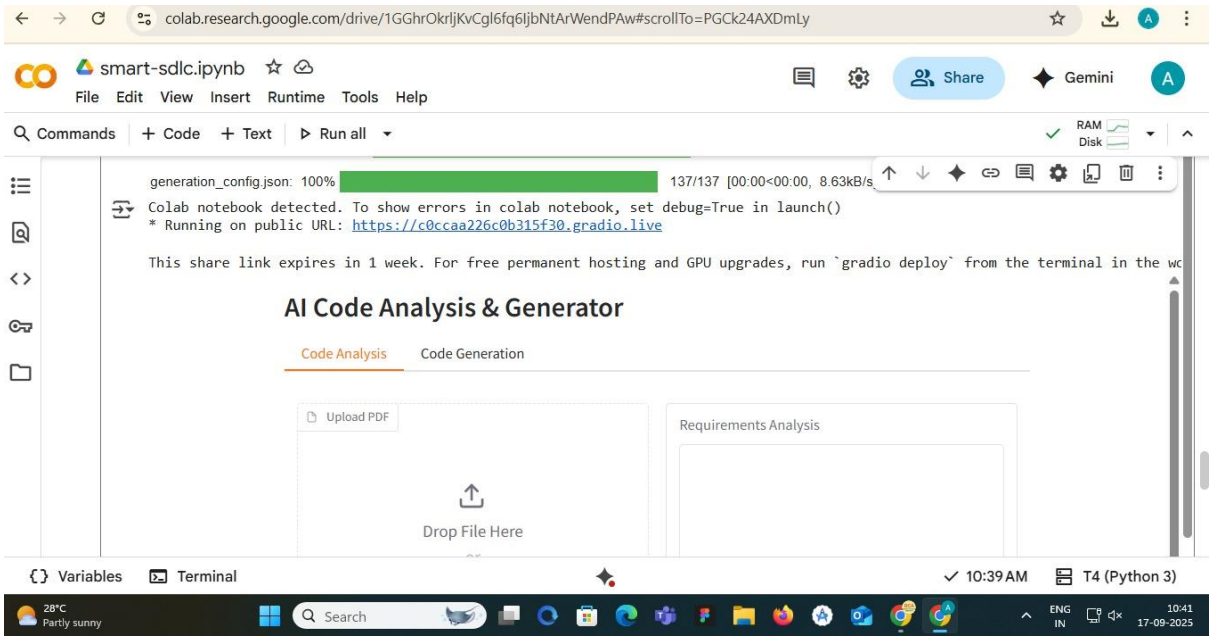
One-Click App Launch:

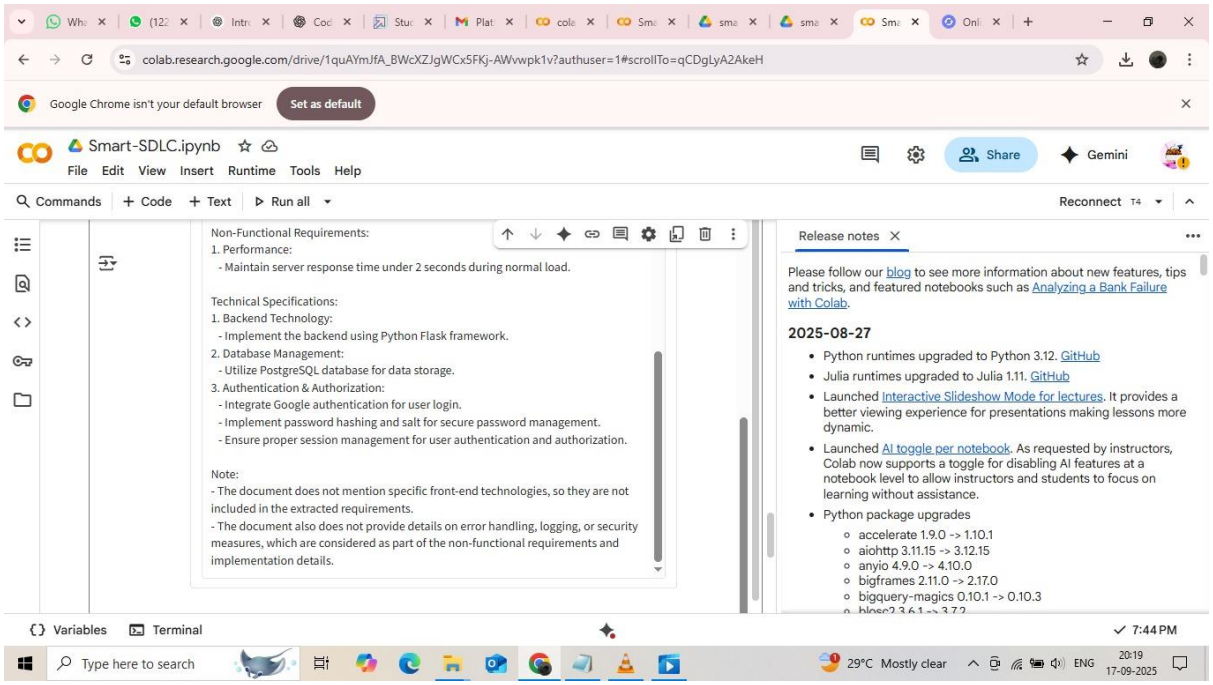
At the end, app.launch(share=True) starts the web app and creates a link for easy sharing.

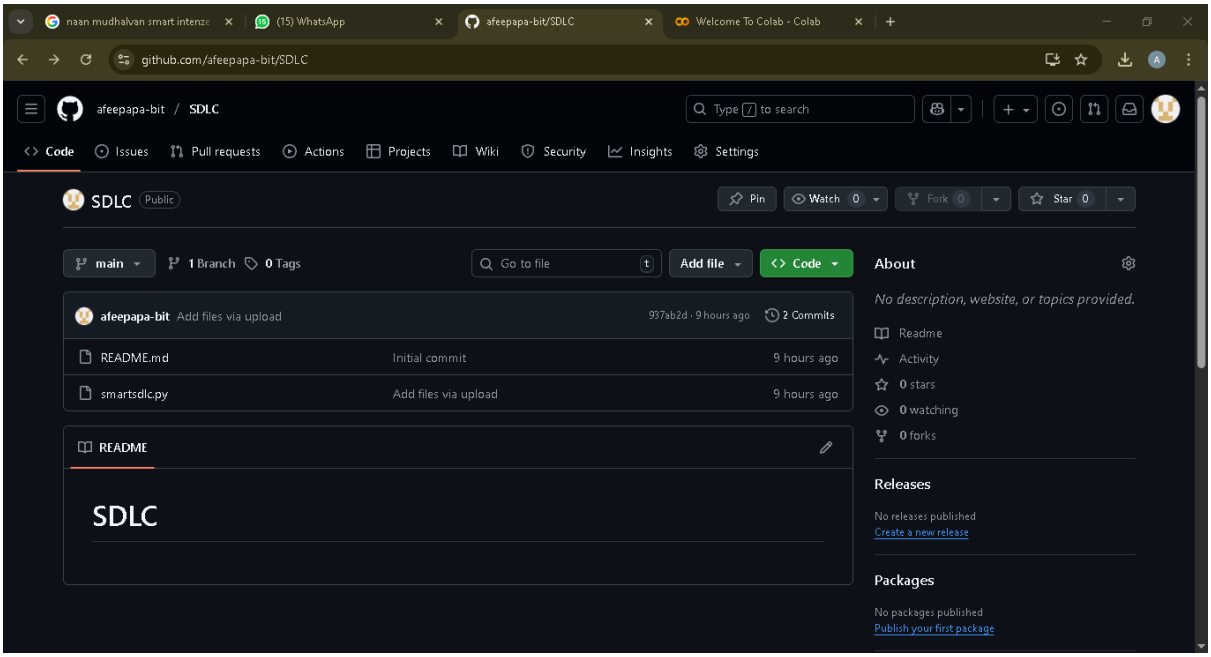
15. Links

1. **Naan Mudhalvan - link**
2. **GitHub - link**
3. **GoogleColab - link**
4. **Program - link**
5. **Hugging Face - link**

SCREENSHOT OF OUTPUT:







Conclusion

Smart-SDLC represents a significant advancement in software development by integrating Artificial Intelligence into the entire Software Development Life Cycle. It addresses common challenges such as manual requirement analysis, repetitive coding, and potential human errors by automating key tasks through AI-driven tools. With capabilities like intelligent requirement classification, automated code generation in multiple programming languages, and a simple user-friendly interface, Smart-SDLC enhances efficiency, accuracy, and collaboration within development teams.

Although still evolving, with limitations like occasional incomplete text extraction and the need for internet connectivity, the platform lays a solid foundation for future enhancements. Planned features such as downloadable reports, multi-language support, IDE integration, advanced code generation, and collaborative tools will further empower teams to deliver high-quality software faster and more reliably.

Overall, Smart-SDLC paves the way toward the future of software engineering by combining the power of AI with practical development workflows, making software creation smarter, faster, and accessible to both technical and non-technical stakeholders.