Smart SDLC – AI Enhanced Software Development Lifecycle

Project Documentation

Introduction

Project Title : Smart SDLC

Team member: DEVI R

Team member: KOWSALYA R

Team member: ASWINI B

Team member: PRIYANGA S

Project overview

Purpose

The SMART SDLC Assistant is designed to simplify and accelerate the Software Development Life Cycle (SDLC) using AI-driven automation. It leverages IBM Watsonx Granite LLMs integrated with Google Colab, Gradio, and PyTorch to streamline requirement gathering, code generation, and debugging.

The platform addresses three core phases of SDLC:

1. **Requirement Upload & Classification** – Automatically extracts and classifies requirements into SDLC phases (Requirements, Design, Development, Testing, Deployment).
2. **AI Code Generator** – Converts natural language prompts or structured user

stories into clean, production-ready code.

3. **Bug Fixer** – Identifies and fixes syntactical and logical errors in code snippets, returning optimized versions.

Features

Requirement Upload and Classification

- **Key Point:** Automated requirement analysis
- **Functionality:** Users can upload PDF documents containing raw, unstructured requirements. Using PyMuPDF for text extraction and Watsonx Granite-20B for classification, each sentence is categorized into SDLC phases (Requirements, Design, Development, Testing, Deployment). The output is displayed as structured user stories grouped by phase, improving clarity and traceability.

AI Code Generator

- **Key Point:** Accelerated code development
- **Functionality:** Developers can input natural language prompts or structured user stories. The Watsonx AI model generates clean, contextually relevant, production-ready code. Code is displayed with syntax highlighting for easy review and integration.

Bug Fixer

- **Key Point:** Intelligent debugging support

- **Functionality:** Users can submit buggy code snippets in languages like Python or JavaScript. This accelerates debugging and ensures cleaner, more reliable code.

3. Architecture

- **Frontend(Gradio):**
  Provides a multi-tab interface (Requirement Classification, Code Generation, Bug Fixer). Handles file uploads, text inputs, dropdown selections, and displays AI-generated outputs.
- **Backend (IBM Watsonx Granite via Hugging Face Transformers):** Processes user inputs and generates outputs using large language models for classification, code generation, and debugging.
- **PDF Processing (PyPDF2):**
  Extracts raw text from PDF requirement documents for AI analysis.
- **Runtime(GoogleColab):**
  Ensures libraries (transformers, torch, gradio, PyPDF2) are installed dynamically in the notebook environment for reproducible execution.

4. Setup Instructions

Prerequisites

- Google Colab account
- Python 3.9+ runtime
- Libraries: transformers, torch, gradio, PyPDF2

Installation Process (Colab)

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

Running the Application

1. Open the SMART_SDLC.ipynb in Google Colab.
2. Install dependencies (executed once per session).
3. Run all code cells.
4. Launch Gradio interface (app.launch(share=True)).
5. Interact with the assistant via browser tabs.

5. Folder/Notebook Structure

- **SMART_SDLC.ipynb** – Main Colab notebook.
- **Helper Functions:**
  - generate_response() – LLM response handler.
  - extract_text_from_pdf() – Extracts text from PDF documents.
- **Scenario Functions:**
  - requirement_analysis() – Classifies requirements into SDLC phases.
  - code_generation() – Generates code in selected language.
  - bug_fixer() – Analyzes and fixes buggy code.
- **Gradio Interface:**
  - Tab 1: Requirement Classification
  - Tab 2: AI Code Generator
  - Tab 3: Bug Fixer

6. Running the Application

To start the project:

1. Open the notebook in Google Colab.
2. Install dependencies with !pip install ....
3. Execute the notebook cells.
4. Use the Gradio UI to:
    - Upload requirements as **PDF** or enter text.
    - Generate code for selected requirements.
    - Paste buggy code and receive corrected versions.

7. API/Model Documentation

- **Model Used:** ibm-granite/granite-3.2-2b-instruct (LLM via Hugging Face).
- **Key Functions:**
    - **Requirement Analysis:** Classifies text into SDLC phases.
    - **Code Generation:** Converts NL requirements to code.
    - **Bug Fixer:** Returns optimized, corrected code with explanations.

8. Authentication

- Current version runs in **open demo mode** on Colab with Hugging Face model.
- Future enhancements:
    - IBM Watsonx API key integration
    - JWT-based session security
    - Role-based user management (analyst, developer, tester)

9. User Interface

The Gradio UI includes:

- **Sidebar Tabs:** For Requirement Classification, Code Generator, and Bug Fixer.
- **File Upload:** PDF requirement upload.
- **Dropdowns:** Language selection for code generation and debugging.
- **Outputs:** Textboxes displaying AI results (classified requirements, generated code, fixed code).

10. Testing

- **Unit Testing:** Prompt handling, PDF extraction, model output validation.
- **Manual Testing:** File upload, NL input, code generation, debugging outputs.
- **Edge Cases:** Large PDF inputs, malformed requirements, invalid code snippets.

11. Screenshots

Input

```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,
            temperature=0.7,
            do_sample=True,
```

SMART_SDLC.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▷ Run all

[2]

```
    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(
```

SMART_SDLC.ipynb

File Edit View Insert Runtime Tools Help

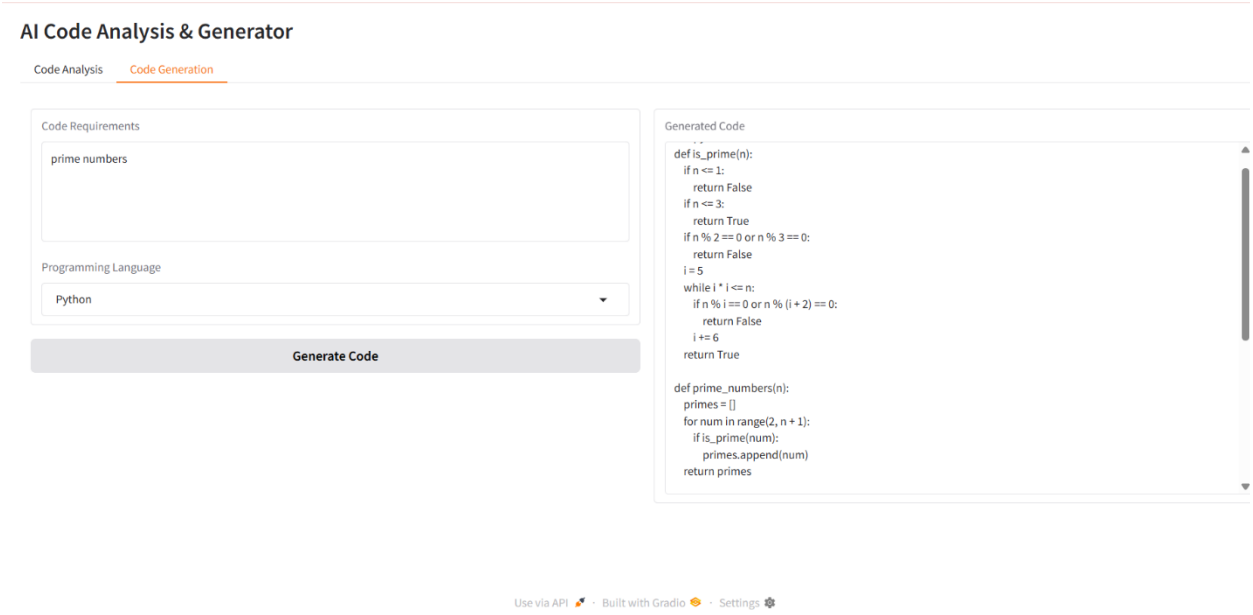
Commands + Code + Text ▷ Run all

[2]

```
                    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)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in y
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 8.88k/? [00:00<00:00, 273kB/s]
vocab.json: 777k/? [00:00<00:00, 17.7MB/s]
merges.txt: 442k/? [00:00<00:00, 10.8MB/s]
```

Output



12. Known Issues

- Output formatting may vary based on AI response.
- Large PDFs may take longer to process.
- Code correctness depends on prompt clarity.
- Requires internet connection to access Hugging Face model.

13. Future Enhancements

- Enhanced UI with collapsible panels for structured output.

- Export to **Word/PDF** reports for requirement analysis.
- Multi-language support (for requirements, not just code).
- Integration with Jira/GitHub for end-to-end SDLC automation.
- Advanced bug-fixing with test case generation.