SmartSDLC – AI-Enhanced Software Development Lifecycle

Project Description:

SmartSDLC is a full-stack, AI-powered platform that redefines the traditional Software Development Lifecycle (SDLC) by automating key stages using advanced Natural Language Processing (NLP) and Generative AI technologies.

It is not just a tool — it's an intelligent ecosystem that allows teams to convert unstructured requirements into code, test cases, and documentation instantly, thereby minimizing manual intervention, enhancing accuracy, and accelerating the delivery pipeline.

Scenarios:

Scenario 1: Requirement Upload and Classification

Requirement Upload and Classification, the platform simplifies the complex task of requirement gathering by allowing users to upload PDF documents containing raw, unstructured text. The backend extracts content using PyMuPDF and leverages IBM Watsonx's Granite-20B AI model to classify each sentence into specific SDLC phases such as Requirements, Design, Development, Testing, or Deployment. These classified inputs are then transformed into structured user stories, enabling clear planning and traceability. The frontend displays this output in an organized, readable format grouped by phase, significantly improving clarity and saving manual effort.

Scenario 2: AI Code Generator

AI Code Generator, addresses the development phase, where developers can input natural language prompts or structured user stories. These prompts are sent to the Watsonx model, which generates contextually relevant, production-ready code. This reduces the time needed for boilerplate or prototype creation and enhances coding efficiency. The code is presented in a clean, syntax-highlighted format on the frontend, ready for use or further enhancement.

Scenario 3: Bug Fixer

Bug Fixer, the platform supports debugging by accepting code snippets in languages such as Python or JavaScript. Upon receiving the buggy code, the Watsonx AI analyzes it for both syntactical and logical errors and returns an optimized version. This not only assists developers in identifying mistakes without extensive manual reviews but also provides immediate, corrected code directly in the frontend for comparison.

Scenario 4: Test Case Generator

Test Case Generator focuses on quality assurance, where users provide functional code or a requirement, and the AI generates suitable test cases. These are structured using familiar testing frameworks like unittest or pytest, enabling seamless validation and automation of software testing. This module eliminates the need for manual test writing, ensuring consistency and completeness in test coverage.

Scenario 5: Code Summarizer

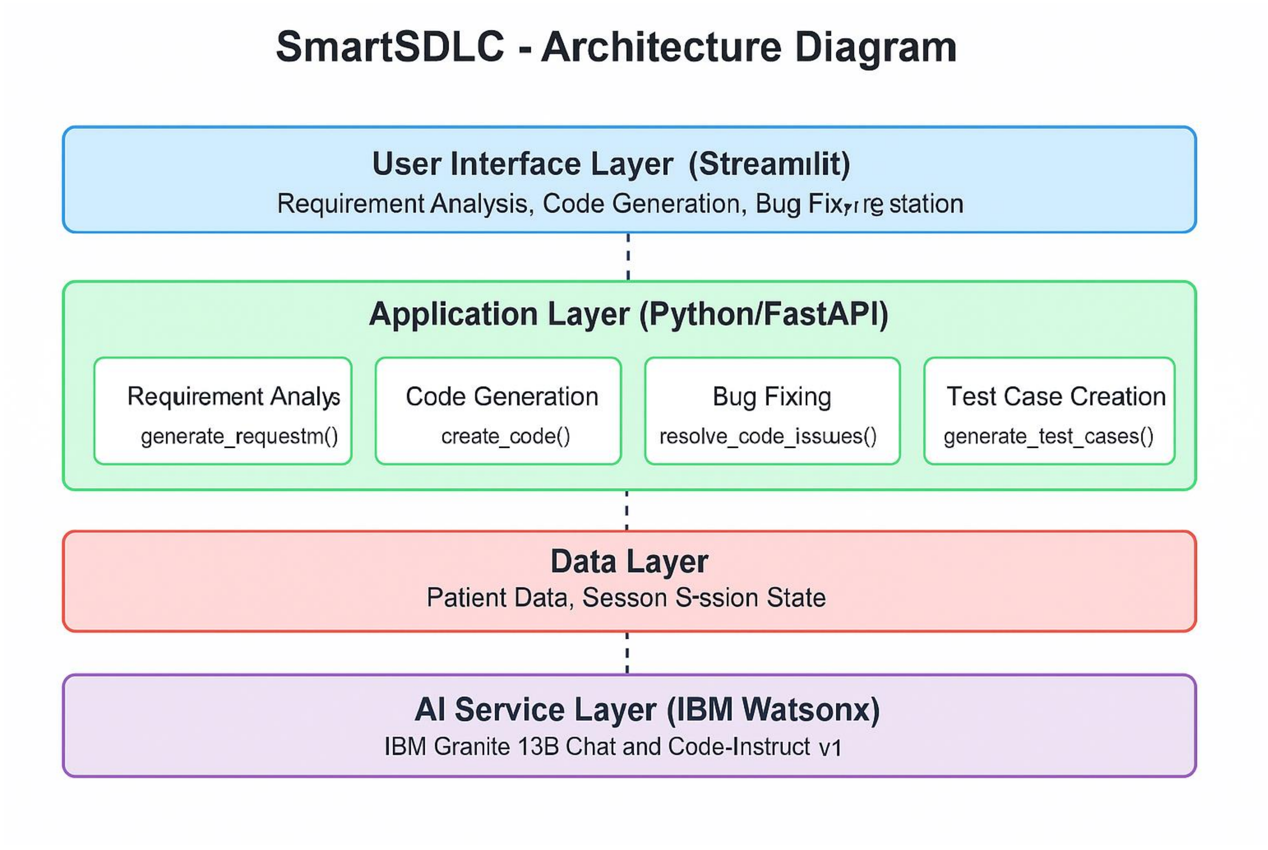
Code Summarizer, the platform aids documentation by accepting any source code snippet or module and generating a human-readable explanation. Watsonx analyzes the logic and purpose of the code, then summarizes its function and use cases. This feature is especially helpful for onboarding new developers or for maintaining long-term documentation, making complex codebases easier to understand.

Scenario 6: Floating AI Chatbot Assistant

Floating AI Chatbot Assistant provides real-time, conversational support throughout the application. Integrated using LangChain, the chatbot responds intelligently to user queries about the SDLC, such as “How do I write a unit test?” or “What is requirement analysis?”. The backend handles prompt routing based on keyword detection, and the frontend presents the response in a styled chat interface, offering an intuitive and interactive help system.

Collectively, these scenarios demonstrate how SmartSDLC intelligently automates core development tasks, enhances team collaboration, and empowers both technical and non-technical users to efficiently engage with the software development process.

Technical Architecture:


SmartSDLC - Architecture Diagram
User Interface Layer (Streamılit)
Requirement Analysis, Code Generation, Bug Fix,ı rę station
Application Layer (Python/FastAPI)
Requirement Analys
generate_requestm()
Code Generation
create_code()
Bug Fixing
resolve_code_issuues()
Test Case Creation
generate_test_cases()
Data Layer
Patient Data, Sesson S-ssion State
AI Service Layer (IBM Watsonx)
IBM Granite 13B Chat and Code-Instruct v1

Pre-requisites:

1. Python 3.10
2. FastAPI
3. Streamlit
4. IBM Watsonx AI & Granite Models
5. LangChain
6. Uvicorn
7. PyMuPDF
8. Git & Github
9. Frontend libraries

Activity 1: Development Environment Setup

- **Activity 1.1:** Set up the development environment by installing necessary libraries and dependencies for FastAPI, Streamlit, LangChain, PyMuPDF, and IBM Watsonx Granite model integration.

Activity 2: Core Functionalities Development

- **Activity 2.1:** Develop the core functionalities: Requirement Classification, Code Generation, Bug Fixing, Test Case Creation, and Code Summarization, powered by Watsonx Granite models.
- **Activity 2.2:** Implement auxiliary utilities like PDF parsing for requirement documents and code validation metrics for generated outputs.

Activity 3: Backend Development

- **Activity 3.1:** Write the main backend logic in `main.py` and relevant routes, establishing API endpoints for each SmartSDLC feature and integrating AI responses using Watsonx models.
- **Activity 3.2:** Create effective prompt engineering strategies for the Watsonx Granite models to ensure high-quality, structured SDLC outputs across multiple programming languages.

Activity 4: Frontend Development

- **Activity 4.1:** Design and develop the user interface using Streamlit components, ensuring an intuitive, clean, and task-oriented layout for SDLC automation workflows.
- **Activity 4.2:** Create dynamic visualizations and outputs (e.g., requirement classification results, generated code blocks, test cases) for clear feedback to the user.

Activity 5: Deployment

- **Activity 5.1:** Prepare the SmartSDLC application for deployment by configuring optimized model loading, environment variables, and memory-efficient API integration for the Granite models.
- **Activity 5.2:** Deploy the full-stack application on a suitable hosting platform, making SmartSDLC accessible to users for streamlined software development lifecycle automation.

Model Selection and Architecture

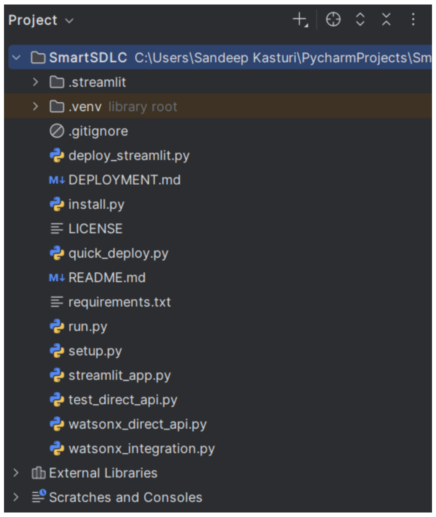
In this milestone, we focus on selecting and integrating the IBM Granite-3.3-2b-instruct model for our language learning needs. This involves configuring the model with appropriate parameters, ensuring optimal performance for educational content generation, and establishing the foundation for multilingual language instruction capabilities.

Activity 1.1: Set up the development environment

1. Install Python and Pip: Ensure Python is installed along with pip for managing dependencies.
2. Create a Virtual Environment: Set up a virtual environment to isolate project dependencies.
3. Install Required Libraries:

   bash
   ```
   pip install streamlit langchain langchain-ibm pymufpdf
   ```

4. Set Up Application Structure: Create the initial directory structure for the Language Guru application.



```
Project
SmartSDLC C:\Users\Sandeep Kasturi\PycharmProjects\Sm
  .streamlit
  .venv library root
  .gitignore
  deploy_streamlit.py
  DEPLOYMENT.md
  install.py
  LICENSE
  quick_deploy.py
  README.md
  requirements.txt
  run.py
  setup.py
  streamlit_app.py
  test_direct_api.py
  watsonx_direct_api.py
  watsonx_integration.py
External Libraries
Scratches and Consoles
```

Core Functionalities Development

Activity 2.1: Develop Core Functionalities

1. **Requirement Classification System**
   - Implement document parsing interface to extract requirements from PDF files
   - Create prompting system for the Watsonx Granite model to classify requirements (e.g., functional, non-functional)
   - Develop validation mechanisms to flag incomplete or ambiguous requirements
2. **Automated Code Generation System**
   - Build automatic language detection for code snippets (e.g., Python, Java, JavaScript)
   - Develop contextual prompt generation for the Granite-20b-code-instruct model to produce accurate, functional code
   - Structure output format to show code blocks with clear annotations
3. **Bug Detection and Fixing Module**
   - Implement bug detection logic by analyzing code input and AI-generated diagnostics
   - Create structured bug reports with identified issues and recommended fixes
   - Integrate automated bug-fix generation using Granite model responses
4. **Test Case and Documentation Generator**
   - Develop intelligent test case generation based on provided requirements or code
   - Create three output types: Unit Tests, Integration Tests, and Edge Case Tests
   - Build automatic documentation generation summarizing code functionality, limitations, and usage examples

---

Activity 2.2: Implement Data Management and Utility Functions

1. **Language and File Type Detection Integration**
   - Integrate file type detection (PDF, TXT, DOCX) and basic language identification for code or text inputs
   - Support for multiple programming languages including Python, Java, JavaScript, and C++
   - Implement confidence scoring for detection accuracy and content validation
2. **Code Quality and Complexity Metrics**
   - Calculate key metrics such as lines of code, cyclomatic complexity, and function count
   - Generate visual reports using Matplotlib or Streamlit charts for project maintainability insights
   - Create radar charts to represent code quality, complexity, and coverage levels
3. **Model Response Processing and Output Structuring**
   - Implement intelligent response parsing for different SDLC output types (requirements, code, tests)
   - Structure outputs with clear formatting, syntax highlighting, and export options

main.py Development

Activity 3.1: Write the Main Application Logic

The `main.py` and backend routes are organized into several key sections:

1. Imports and Setup

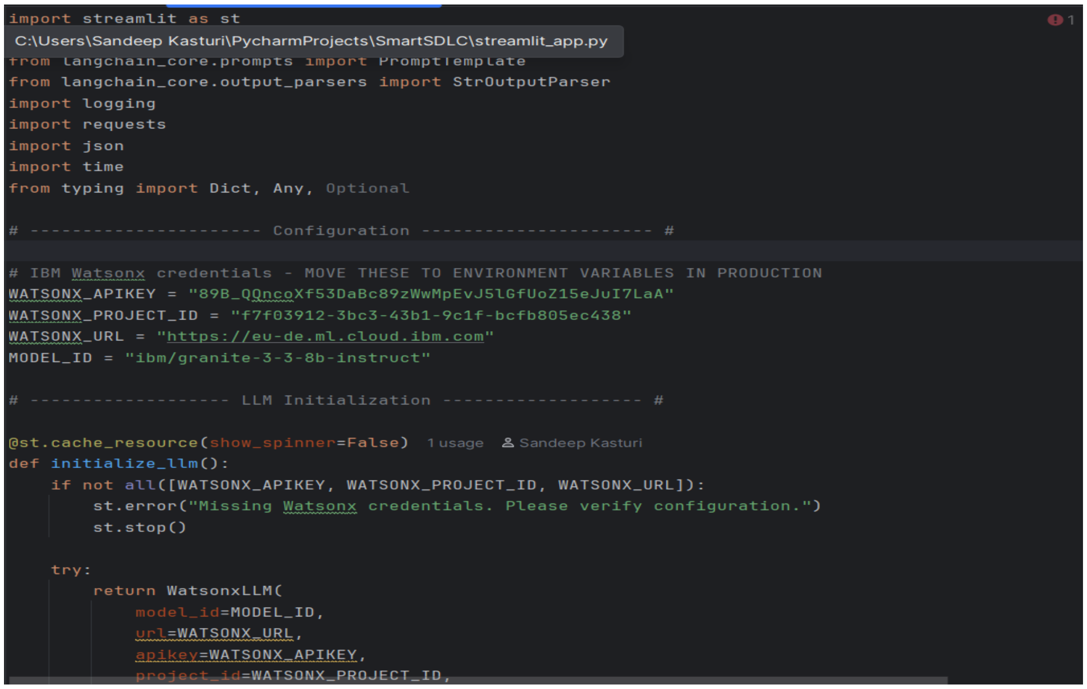
- Import necessary libraries (FastAPI, PyMuPDF, Streamlit, LangChain, IBM Watsonx SDK, Pydantic)
- Load IBM Watsonx Granite-13b-chat-v1 and Granite-20b-code-instruct models with optimal configurations
- Initialize prompt templates, authentication, and model settings

2. Core Functions

- `generate_requirement_classification()`: Handle requirement classification using the Granite model
- `generate_code()`: Automatic, AI-powered code generation based on prompts or requirements
- `fix_bugs()`: Analyze buggy code input and provide AI-suggested fixes
- `generate_test_cases()`: Create relevant, structured test cases for the provided code
- `summarize_code()`: Generate human-readable summaries of complex code blocks

3. Frontend Integration (Streamlit)

- Interactive layout using Streamlit Pages and Components for clean navigation
- PDF upload components, text input forms, and result displays
- AI-generated code blocks, test cases, and summaries rendered dynamically

- 

```
import streamlit as st
C:\Users\Sandeep Kasturi\PycharmProjects\SmartSDLC\streamlit_app.py
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
import logging
import requests
import json
import time
from typing import Dict, Any, Optional

# --------------------- Configuration --------------------- #

# IBM Watsonx credentials - MOVE THESE TO ENVIRONMENT VARIABLES IN PRODUCTION
WATSONX_APIKEY = "89B_QQncoXf53DaBc89zWwMpEvJ5lGfUoZ15eJuI7LaA"
WATSONX_PROJECT_ID = "f7f03912-3bc3-43b1-9c1f-bcfb805ec438"
WATSONX_URL = "https://eu-de.ml.cloud.ibm.com"
MODEL_ID = "ibm/granite-3-3-8b-instruct"

# ------------------ LLM Initialization ------------------ #

@st.cache_resource(show_spinner=False)  1 usage  Sandeep Kasturi
def initialize_llm():
    if not all([WATSONX_APIKEY, WATSONX_PROJECT_ID, WATSONX_URL]):
        st.error("Missing Watsonx credentials. Please verify configuration.")
        st.stop()

    try:
        return WatsonxLLM(
            model_id=MODEL_ID,
            url=WATSONX_URL,
            apikey=WATSONX_APIKEY,
            project_id=WATSONX_PROJECT_ID,
```

4. Feature Implementation Examples

- `display_requirement_analysis()`: Interface for uploading requirement documents and viewing classifications
- `display_code_generation()`: User-friendly input prompts for code generation with language selection
- `display_bug_fixing()`: Upload buggy code, receive AI-generated fixes and reports
- `display_test_case_generation()`: Automated test case suggestions with explanation

Activity 3.2: Create Prompting Strategies

- **Requirement Analysis Prompting:**
  Craft prompts for Granite-13b-chat-v1 to accurately classify uploaded requirement documents (functional, non-functional, ambiguous).
- **Code Generation Prompting:**
  Structure prompts for Granite-20b-code-instruct to generate complete, optimized, and language-specific code snippets based on requirements or tasks.
- **Bug Detection and Fixing Prompting:**
  Develop clear prompts for bug identification, error descriptions, and AI-generated fixes for common code issues.
- **Test Case Generation Prompting:**
  Design prompts to instruct the AI to produce comprehensive unit, integration, and edge case tests for submitted code.

Milestone 4: Frontend Development

Activity 4.1: Design and Develop the User Interface

1. Main Application Layout

- Configure Streamlit Pages with **SmartSDLC** branding
- Implement sidebar navigation for core modules: Requirement Analysis, Code Generation, Bug Fixing, Test Case Generation, Code Summarization
- Develop intuitive forms with clear validation and error handling

2. Feature-Specific Interfaces

- **Requirement Classification:** PDF upload, result display with color-coded requirement types
- **Code Generation:** Text prompts, language selection dropdown, generated code with syntax highlighting
- **Bug Fixing:** Code editor input for buggy code, AI-suggested fixes output
- **Test Case Generation:** Code upload or prompt input, generated test cases with explanations

---

Activity 4.2: Create Dynamic Visualizations

1. SDLC Progress and Metrics Visuals

- Requirement classification breakdown charts (e.g., pie charts of requirement types)
- Code complexity metrics (lines of code, function count) displayed with bar or radar charts
- Test coverage and bug statistics visualized for clear project health insights

2. Real-Time Feedback Metrics

- Requirement document completeness score
- Code quality indicators (maintainability, complexity)
- AI confidence levels for requirement classification and code generation

Milestone 5: Deployment

Activity 5.1: Prepare for Deployment

1. Model Configuration

- Configure Watsonx Granite models for optimized loading and performance
- Implement memory-efficient processing for AI responses
- Set up API keys and `.env` files for secure configuration

2. Dependency Management

- Create `requirements.txt` with all necessary packages:
  `fastapi`, `streamlit`, `pymupdf`, `requests`, `langchain`, `ibm-watsonx-ai`, `uvicorn`

Activity 5.2: Deploy the Application

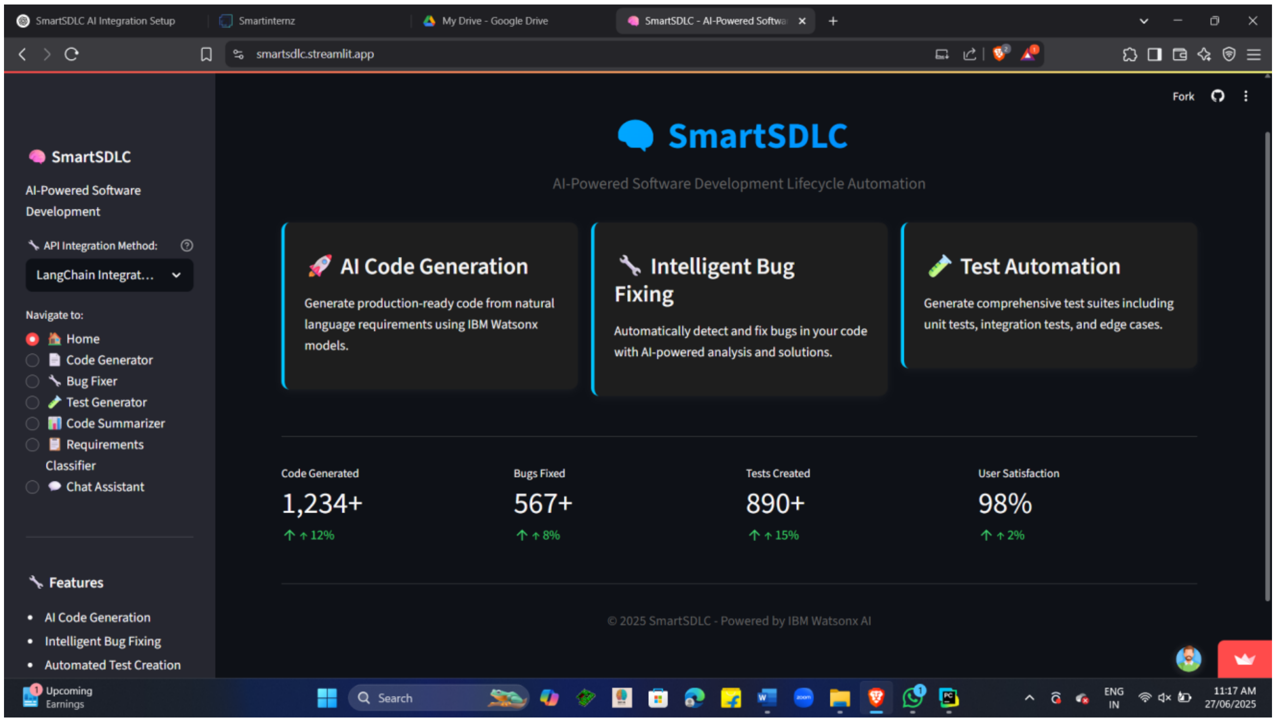
1. Local Deployment Testing

- Run the backend with `uvicorn main:app --reload`
- Launch Streamlit frontend and test all modules for functionality
- Verify AI model integration and output generation

2. Cloud Deployment Options

- Deploy on cloud platforms like IBM Cloud, AWS, or Streamlit Cloud
- Configure model access and GPU resources for optimal AI performance
- Set up monitoring for API usage, model response times, and system health

Exploring Application Features:



SmartSDLC

AI-Powered Software Development Lifecycle Automation

AI Code Generation

Generate production-ready code from natural language requirements using IBM Watsonx models.

Intelligent Bug Fixing

Automatically detect and fix bugs in your code with AI-powered analysis and solutions.

Test Automation

Generate comprehensive test suites including unit tests, integration tests, and edge cases.

Code Generated	Bugs Fixed	Tests Created	User Satisfaction
1,234+	567+	890+	98%
↑ 12%	↑ 8%	↑ 15%	↑ 2%

© 2025 SmartSDLC - Powered by IBM Watsonx AI

Requirement Classification Page

Description: Upload requirement documents (PDF) for AI-driven classification into functional, non-functional, and ambiguous categories, enhancing clarity for the development team.

Code Generation Page

Description: Users input prompts or requirements; the system uses Watsonx Granite models to generate accurate, language-specific code with clear annotations for rapid development.

Bug Detection and Fixing Page

Description: Upload buggy code; the AI identifies errors and suggests clear, structured fixes, improving code quality with minimal manual debugging effort.

Test Case Generation Page

Description: AI automatically generates comprehensive test cases (unit, integration, edge) based on provided code, accelerating the testing process and boosting reliability.

Code Summarization Page

Description: Upload complex code files to receive AI-generated, human-readable summaries explaining functionality, structure, and key logic for better understanding.

Conclusion

SmartSDLC, powered by IBM Watsonx AI and a structured FastAPI + Streamlit architecture, streamlines the software development lifecycle by automating requirement analysis, code generation, bug fixing, testing, and documentation. Through strategic AI integration and a clean, interactive interface, SmartSDLC enhances efficiency, reduces errors, and accelerates software delivery.