­GENERATIVE AI WITH IBM CLOUD

**Project Documentation**

**1.Introduction:**

**Project Title: SmartSDLC – AI- Enhanced Software Development Lifecycle**

**Team Members:**

* **Team Leader:** Viswanadhapalli Hasanthi
* **Team Member:** Tadisetti Tejasri
* **Team Member:** Thatiparthy Sai Teja
* **Team Member:** Vaishnavi Pachipala

**2. Project Overview:**

**Purpose:**

Smart SDLC is a lightweight AI-powered tool designed to support software developers, students, and learners across different stages of the Software Development Life Cycle. It uses IBM Watsonx's Granite model to provide intelligent automation for code generation, bug fixing, test case creation, requirement clarification, and technical doubt-solving. The platform accelerates development workflows, reduces manual effort, and promotes self-learning.

### ****Features:****

* AI-powered code generation and bug fixing
* Automated test case generation from requirements
* Requirement-to-functional spec transformation
* Chatbot for software development–related doubts
* Topic-based code explanation across multiple languages
* Natural language input to development artifact generation

### ****3. Architecture****

#### **Frontend (UI Layer):**

* Built using **Streamlit**, a Python-based UI framework.
* A simple, tabbed interface allows users to choose between functionalities like bug fixing, code generation, requirement transformation, and test case generation.
* Users can input prompts, view AI responses, and interact with a developer chatbot through a clean and minimal interface.

#### **Backend (Application Logic):**

* Entire backend logic is implemented using **FastAPI**.
* Each functionality (e.g., bug fixing, code generation) is handled in separate Python modules for modularity.
* Integrates IBM Watsonx **Granite 3.3-2B-Instruct** model via the ModelInference method using credentials and project ID (no IAM Authenticator used).

#### **Storage & Sessions:**

* No persistent database is used.
* Temporary user interactions and session tracking are managed using **Python dictionaries** and Streamlit session state to support user flow during a session.

### ****4. Setup Instructions****

#### **Prerequisites:**

* Python **3.10+**
* **Streamlit** for frontend
* **FastAPI** for backend
* IBM Watsonx account with API key and project ID with Watsonx Machine Learning Service Associated to project
* Basic command-line knowledge
* VSCode

#### **Installation Steps:**

* **Clone the repository:**
* git clone https://github.com/your-username/SmartSDLC.git
  + cd SmartSDLC
* **Create a virtual environment and activate it:**
  + python -m venv venv
  + venv\Scripts\activate # For Windows
  + source venv/bin/activate # For macOS/Linux
* **Install dependencies:**
  + pip install -r requirements.txt
* **Set your IBM Watsonx credentials:**
  + Create a .streamlit/secrets.toml file in the frontend directory:

API\_KEY = "your-ibm-api-key"

PROJECT\_ID = "your-project-id"

* **Run the backend:**
  + cd backend
  + uvicorn main:app –reload
* **Run the frontend: open new terminal in vscode**
  + Cd frontend
  + streamlit run app.py

#### **5. Folder Structure:**

The project is structured into two key folders: the backend (FastAPI) and the frontend (Streamlit). Each has its own virtual environment and utilities.

Below is the hierarchy:

SmartSDLC/

├── fastapi\_app/

│ ├── templates/

│ │ └── index.html

│ ├── venv/

│ ├── main.py

│ ├── requirements.txt

│ └── watsonx\_utils.py

│

├── streamlit\_app/

├── venv/

├── app.py

├── requirements.txt

└── watsonx\_utils.py

main.py: Contains FastAPI routes (e.g., /generate\_code, /fix\_bug, etc.).

app.py: Contains the Streamlit UI logic.

watsonx\_utils.py: Shared model logic (one per side, can be similar).

index.html: Optional FastAPI HTML template if not using pure API-only.

Separate venv/ folders allow isolated environments for frontend/backend.

**6. Running the Application**

#### **Local Machine (FastAPI + Streamlit Setup):**

##### **Step 1: Backend (FastAPI)**

1. Open terminal inside the fastapi\_app/ directory.
2. Activate virtual environment (if created):

venv\Scripts\activate # Windows

source venv/bin/activate # Linux/macOS

1. Run the backend:

uvicorn main:app --reload

1. FastAPI will start at: http://127.0.0.1:8000

##### **Step 2: Frontend (Streamlit)**

1. Open another terminal inside the streamlit\_app/ directory.
2. Activate frontend environment:

venv\Scripts\activate # Windows

source venv/bin/activate # Linux/macOS

1. Run the frontend:

streamlit run app.py

1. Streamlit will launch in your browser at:  
   <http://localhost:8501>

### ****7. API Documentation – Smart SDLC****

### **Smart SDLC – Developer Assistant** project is structured into a FastAPI backend and Streamlit frontend. While there are no public REST APIs, internal function calls are made through the Streamlit interface to interact with the AI functionalities.

#### **Function Descriptions (Backend –** fastapi\_app**)**

| **Function Name** | **File** | **Description** |
| --- | --- | --- |
| generate\_code(prompt) | main.py | Converts natural language instructions into code using Watsonx AI. |
| fix\_bugs(prompt) | main.py | Detects and resolves bugs in user-provided code. |
| transform\_requirements(prompt) | main.py | Converts informal requirements into structured SDLC-ready text. |
| generate\_testcases(prompt) | main.py | Creates relevant test cases for given requirements or code. |
| summarize\_code(prompt) | main.py | Summarizes the logic or behavior of a code snippet. |
| chatbot\_reply(prompt) | main.py | Answers technical SDLC-related questions via conversational AI. |

#### **Support Functions (Backend –** watsonx\_utils.py**)**

| **Function Name** | **File** | **Description** |
| --- | --- | --- |
| query\_watsonx() | watsonx\_utils.py | Handles the actual interaction with IBM Watsonx using ModelInference |
| format\_prompt() | watsonx\_utils.py | Formats the user input to align with model expectations (if needed). |

**8. Authentication:**

* The **Login** interface is handled through the **Streamlit app**, where users enter a **username and password** before accessing the Smart SDLC features.
* This is a **basic authentication mechanism** for demonstration purposes and does **not include encryption or secure storage**.
* Ideal for **local demos or academic projects**, but not suitable for production without adding proper security protocols (like hashing, JWT, OAuth, etc.).

### ****9. User Interface – Smart SDLC****

* **Built using Streamlit**, offering a clean and interactive user experience.
* **Sections include**:
  + **Login** and **Register**: For basic authentication before accessing features.
  + **Dashboard**: Where users select various Smart SDLC tools.
* **Input Components**:
  + **Text input areas** for entering software requirements, bug descriptions, code snippets, or prompts.
  + **Dropdowns or radio buttons** to select programming language or task type.
  + **File upload** support for uploading code files or documents.
* **Output Components**:
  + **Text areas** displaying AI-generated outputs (e.g., bug fixes, code, test cases).
  + **Code blocks** for syntax-highlighted source code.

### ****10. Testing – Smart SDLC****

* **Manual functional testing of**:
  + Requirement classification and understanding
  + Code generation from user prompts
  + Bug detection and fixing functionality
  + Test case generation accuracy
  + Code/topic summarization and explanation
  + Login and registration flow validation
* **Output validation**:
  + Results were compared against standard coding practices and verified using sample inputs
  + Bug fixes and generated code were tested in basic IDE environments for correctness
  + Summaries and test cases were reviewed manually for logical accuracy

# **11. Screenshots or Demo:**

Demo Link: <https://drive.google.com/file/d/1_Qm1yXGGMfJJSAUVX6HtB-jAj93YpO_S/view?usp=drive_link>

### ****12. Known Issues****

* **No persistent user sessions** - all input and progress is lost upon app restart.
* **Bug fix accuracy may vary** depending on code structure and prompt clarity.
* **Test cases are not automatically executed or validated** against source code.
* **Code summaries may be generic** if the input lack structure or clarity.
* **Limited error handling** - the system assumes well-formed input from users.

### ****13. Future Enhancements****

* Add evaluation and scoring for generated test cases.
* Integrate a database (e.g., Firebase, MongoDB) to persist user sessions and activity logs.
* Implement code performance metrics and analytics dashboards.
* Enable integration with existing developer tools and LMS platforms like GitHub Classroom or Moodle.
* Incorporate voice-based prompts using Whisper or Speech-to-Text (STT) for hands-free developer support.