PROJECT REPORT

SMART SDLC

TEAM ID : LTVIP2025TMID37665

1. INTRODUCTION

1.1 Project Overview

SmartSDLC AI is an intelligent, AI-powered software solution designed to streamline, automate, and enhance the Software Development Life Cycle (SDLC). The application integrates a locally hosted Large Language Model (LLM), specifically the granite-3.3-2b-instruct-Q4_K_M.gguf model, to assist developers at various stages of software engineering including requirements analysis, code generation, test case creation, bug fixing, and documentation.

This tool was developed with the idea of reducing the cognitive load and manual effort required for common software engineering tasks. The project includes both frontend and backend components. The backend, built using FastAPI and Python, interfaces with the local LLM to process instructions and return intelligent responses. The frontend, built with HTML, CSS, and JavaScript, provides an intuitive interface for users to interact with the AI system seamlessly.

1.2 Purpose

The purpose of this project is to simplify the software development workflow by automating traditionally manual and repetitive tasks using generative AI. By leveraging the power of language models, SmartSDLC AI aims to:

- Assist in the early phases of software development like requirement gathering and analysis
- Automatically generate boilerplate or specific-purpose code based on textual descriptions
- Produce unit and functional test cases for given code blocks
- Debug and suggest fixes for faulty code
- Auto-generate developer-level documentation

This platform helps both novice and experienced developers by reducing time to delivery, improving consistency, and minimizing human error.

2. IDEATION PHASE

2.1 Problem Statement

Software development today remains a labor-intensive process, especially during the early and middle phases of the SDLC. Developers spend a significant amount of time:

- Writing and re-writing boilerplate code
- Manually documenting modules
- Debugging code without smart insights
- Creating test cases from scratch

Problem Statement 1:

Modern software development teams struggle with managing multiple SDLC phases using fragmented tools.
This leads to inefficiencies, redundant efforts, and slower delivery cycles.
There is a need for a unified AI-powered platform to streamline the entire development lifecycle.

Problem Statement 2:

Developers often face challenges in understanding requirements, writing test cases, debugging, and maintaining documentation.
Manual handling of these tasks consumes valuable time and introduces errors.
An AI-enhanced system can automate these phases, improving productivity and code quality.

Despite advancements in IDEs and DevOps pipelines, there remains a need for an intelligent assistant that can speed up development tasks using natural language interaction. The opportunity lies in creating a tool that leverages recent advances in generative AI to address these gaps.

Project Title:

Smart SDLC
AI-Enhanced Software Development Lifecycle

Problem Statement:

Build an AI-enhanced platform that automates SDLC phases using a local LLM (Granite model) to reduce manual effort and improve developer productivity.

Team Members:

o Palavalsa Sai Tarun (Team Leader)
o Eswar Khandavali
o Greeshma Gudla
o Dharmana Gowrav Munindra

Smart SDLC Project

Collaboration Environment

- **Communication: WhatsApp, Google Meet**
- **Version Control: Git & GitHub**
- **Docs & Notes: Google Docs**
- **Brainstorming Tools: Mural, Canva**

Problem Statement

Developers lose valuable time doing repetitive and manual tasks in software development like requirement writing, test generation, bug fixing, and documentation. This project aims to solve that using AI.

Project Goal

To build a smart, AI-driven tool that automates the core SDLC phases using a local Granite LLM (via llama-cpp-python) to improve developer productivity and project delivery speed.

Scope of Brainstorming

Focus on identifying features, tools, and priorities for automating the 5 SDLC phases:

1. Requirement Analysis
2. Code Generation
3. Test Case Creation
4. Bug Fixing
5. Documentation

Target Users

Who will benefit from Smart SDLC?

- Software Developers
- Interns working on SDLC tasks
- Software Testers
- Project Managers (in future versions)
- These users need fast, reliable, and AI-supported tools to reduce manual work.

Tools & Technologies Used

Tech Stack Overview:

- Backend: FastAPI (Python)
- AI Model: Granite 3.3B (GGUF) via llama-cpp-python
- Frontend: HTML + CSS (no JavaScript)
- Version Control: GitHub
- Deployment: Local machine (optional Netlify for UI)

Brainstorm, Idea Listing and Grouping

Problem Statement:

Developers spend excessive time on repetitive and manual tasks across the Software Development Life Cycle (SDLC), including requirement analysis, code generation, test creation, bug fixing, and documentation. This slows down software delivery and reduces productivity.

Objective of Brainstorming:

To explore how AI can assist or automate different phases of SDLC using local models (like Granite via llama-cpp-python), thereby making the process faster, smarter, and more efficient.

How Might We Questions

Identify innovative ways to use AI to solve SDLC challenges

Palavalsa Sai Tarun

- How might we auto-generate clean backend code from requirements?
- How might we reduce developer time spent on writing boilerplate code?
- How might we automate the process of fixing bugs from code snippets?
- How might we provide real-time debugging suggestions using AI?
- How might we simplify the developer workflow using a unified tool?
- How might we make AI-generated code editable and customizable?

Eswar Khandavali

- How might we generate test cases directly from source code?
- How might we ensure edge cases are covered through AI-generated tests?
- How might we reduce test coverage gaps in rapidly developed code?
- How might we validate AI-generated code is production-ready?
- How might we provide confidence to developers through test automation?
- How might we compare test output before and after bug fixes?

Greeshma Gudla

- How might we streamline the entire SDLC process using AI?
- How might we reduce project delivery time through automation?
- How might we ensure traceability from requirements to code to tests?
- How might we maintain history logs of SDLC activity for review?
- How might we ensure consistent documentation across releases?
- How might we improve team productivity using a smart dashboard?

Dharmana Gowrav Munindra

- How might we convert natural language input into technical specs?
- How might we identify missing requirements automatically?
- How might we auto-generate user stories and use case flows?
- How might we simplify requirement analysis for non-tech users?
- How might we create system-level documentation from plain text?
- How might we ensure documentation is always up to date?

Must have

Transform user input into functional and non-functional requirements using AI.

Generate basic code structure and boilerplate code from requirement input.

Create unit test cases based on the AI-generated or user-provided code.

Allow input of buggy code and return suggested fixes with explanations.

Convert code or requirements into readable technical documentation.

Should have

Automatically suggest folder structure based on code type or project scale.

Track how many features were used, which modules were requested the most.

AI draws architecture diagrams or flowcharts from the requirement input.

Allow exporting AI responses like documentation and code into downloadable formats.

Maintain a list of user inputs and generated outputs for future reference.

Could have

Support beyond Python if time/ resources permit.

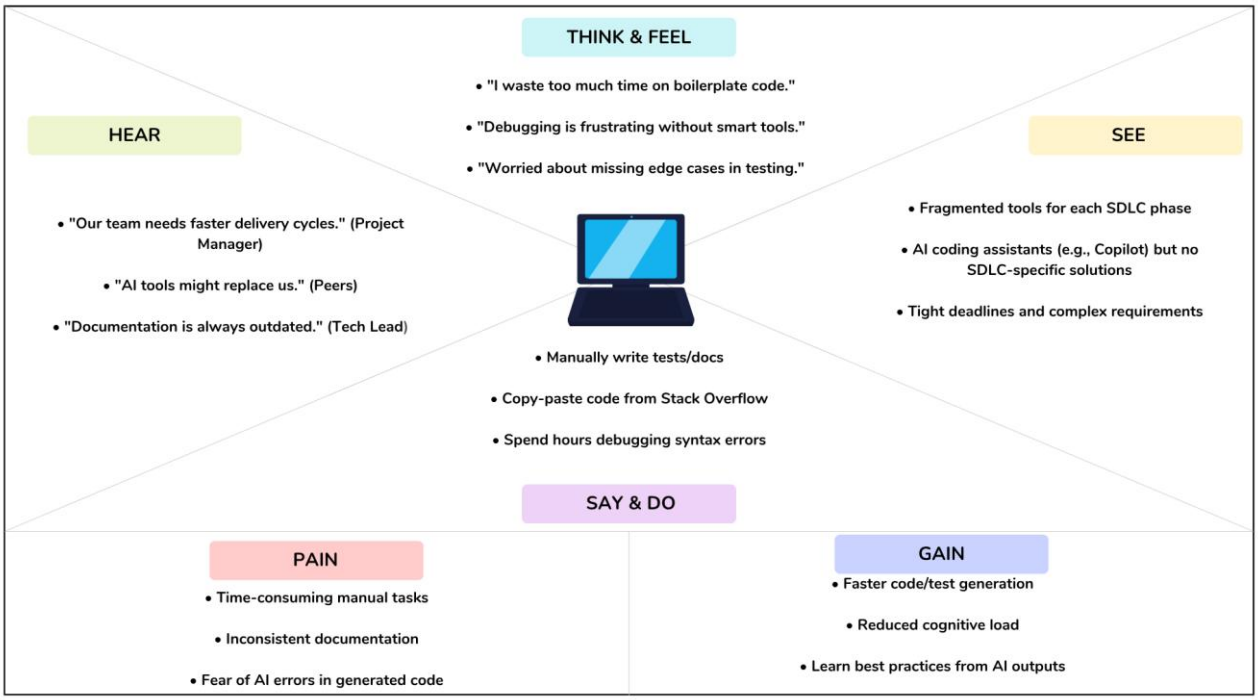
Convert spoken requirements into text and process them (using speech-to-text).

Multiple developers using the same interface collaboratively.

Won't have

See an example

2.2 Empathy Map



Users: Developers, interns, software testers, project managers

Needs:

- Real-time assistance in understanding requirements
- Fast code generation and prototyping
- Help with debugging and testing
- Easy access to documentation support

Pains:

- Time-consuming bug tracing and test writing
- Poor documentation practices
- Redundant and repetitive coding tasks
- Fragmented tooling environments

Gains:

- A unified platform to handle all SDLC assistance
- Higher productivity with lower cognitive load
- More accurate and consistent project outputs
- Offline, local model for enhanced privacy

2.3 Brainstorming

During brainstorming, the following ideas emerged:

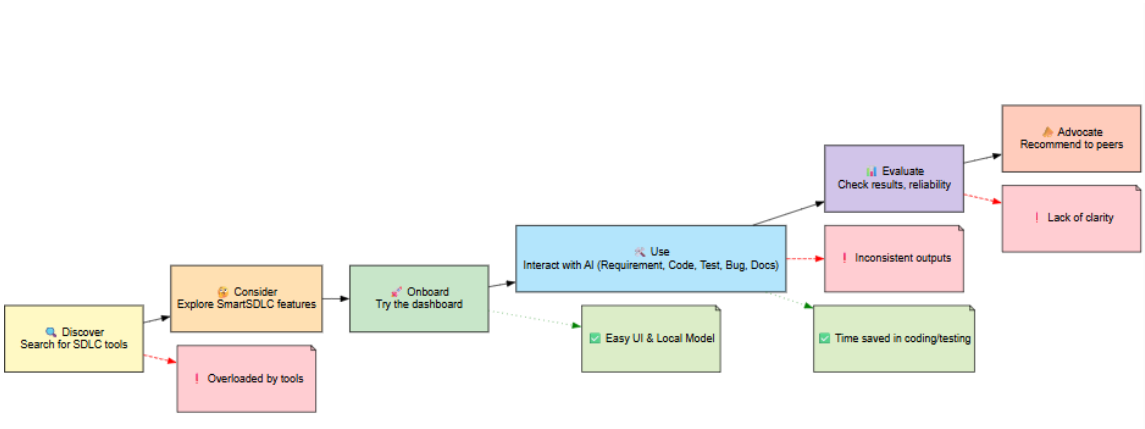
1. **Problem Identification**
    - Long and repetitive SDLC tasks.
    - Lack of real-time automation tools.
2. **User Research**
    - Target users: Developers, Testers, Interns, Project Managers.
    - Needs and pains were gathered using empathy maps and interviews.
3. **Feature Ideation**
    - AI for Requirement Analysis
    - AI-based Code Generation
    - AI-generated Test Cases
    - Bug Detection and Fixing
    - Auto Documentation Support
4. **Tool & Tech Stack Selection**
    - Frontend: Streamlit / HTML-CSS
    - Backend: FastAPI / Flask
    - AI Engine: llama-cpp-python + GGUF (Granite Model)
5. **Workflow Design**
    - Customer journey mapping
    - Modular architecture for each SDLC phase
6. **Wireframing & Prototyping**
    - Dashboard layout
    - Sidebar for navigation
    - Result display area

These ideas were consolidated into a single product: SmartSDLC AI, a generative AI dashboard to support all stages of SDLC.

3. REQUIREMENT ANALYSIS

3.1 Customer Journey Map

1. User opens SmartSDLC AI dashboard in browser
2. Enters input (requirement, code, function, etc.)
3. Selects an operation (analyze, generate-code, fix-bugs, etc.)
4. Backend processes prompt via LLM and returns result
5. Result is displayed in output section
6. User optionally reviews interaction history
7. User logs in and selects the desired SDLC phase (e.g., Requirement Analysis, Code Generation).
8. User submits input (like plain text or code), and the AI provides instant, actionable output.
9. User reviews, downloads, or refines the output, completing the development task more efficiently.



3.2 Solution Requirement

Functional Requirements

1. AI-Based Requirement Analysis
   The system shall accept plain text inputs and extract software requirements using the AI model.
2. Code Generation
   The system shall generate clean, structured source code based on the provided requirements or prompts.
3. Test Case Creation
   The system shall automatically generate relevant unit and functional test cases from the code or requirements.

4. Bug Detection and Fixing
   The system shall detect logical and syntactic bugs in submitted code and suggest or apply fixes.

5. Documentation Generator
   The system shall generate human-readable documentation for the codebase, including function descriptions and usage**.**

Non-Functional Requirements

1. Performance
   The AI responses (code generation, test creation, etc.) should be produced within 5 seconds under normal conditions.

2. Usability
   The interface should be clean, intuitive, and user-friendly even for non-technical users.

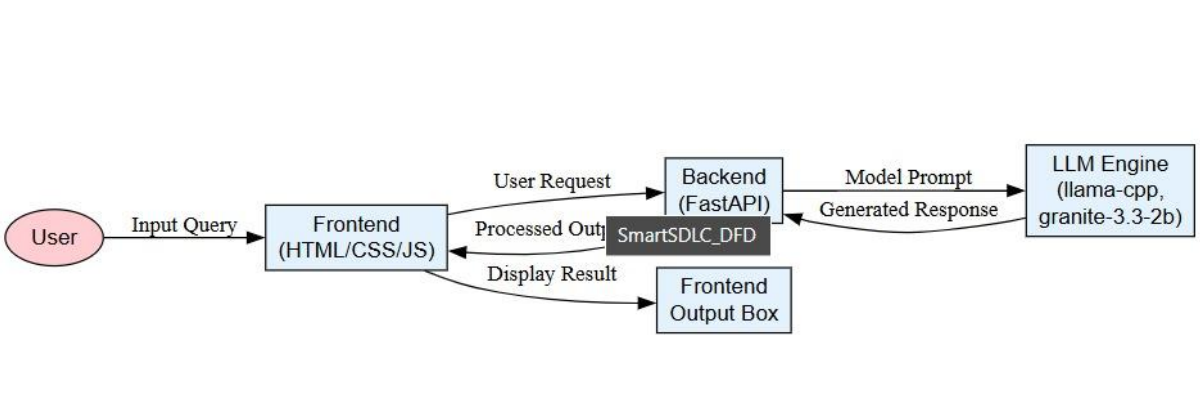
3. Scalability
   The backend should be designed to scale for multiple concurrent users without performance degradation.

4. Reliability
   The system should maintain 99% uptime and consistently deliver correct outputs from the AI model.

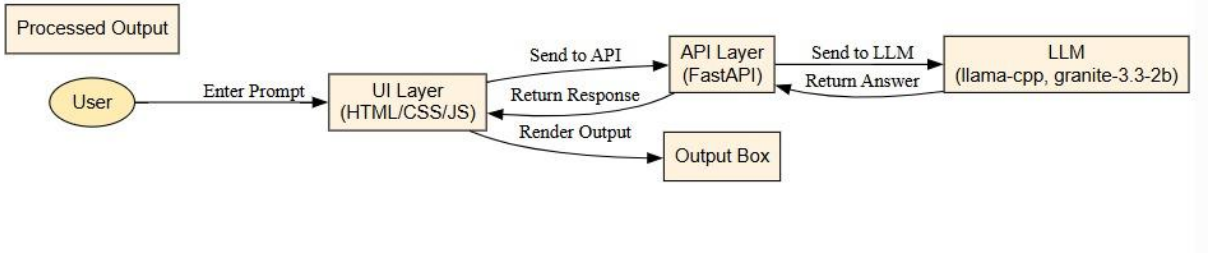
5. Maintainability
   The codebase should be modular and documented to make future updates and debugging easier.

3.3 Data Flow Diagram

Level 0 DFD:



Level 1 DFD:



3.4 Technology Stack

Backend:

- FastAPI (Python)
- llama-cpp for LLM inference
- Model: granite-3.3-2b-instruct-Q4_K_M.gguf

Frontend:

- HTML, CSS, Vanilla JS

Development Tools:

- VS Code
- Python 3.12
- Git for version control

Deployment:

- Localhost based during development
- Possible extension to Docker or cloud in future scope

4. PROJECT DESIGN

4.1 Problem Solution Fit

SmartSDLC AI directly addresses the common pain points faced during the software development lifecycle. It integrates generative AI capabilities into a user-friendly dashboard to improve productivity, reduce manual workload, and enhance software quality. By embedding intelligence into each development phase, the platform provides immediate, actionable insights for developers.

Feature	Problem	Solution	Fit
Requirement Analyzer	Unclear requirements	Extracts structured requirements	Saves time in understanding needs
Code Generator	Manual coding is slow	Generates code from prompts	Speeds up development
Test Case Creator	Test writing is tedious	Creates test cases automatically	Improves testing efficiency
Bug Fixer	Debugging is time-consuming	Detects and fixes code issues	Faster and cleaner code
Documentation Generator	Lack of proper docs	Generates documentation from code	Makes maintenance easier

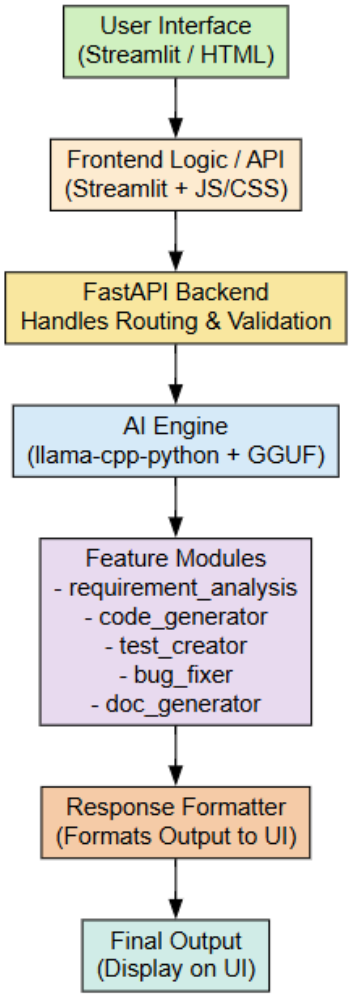
4.2 Proposed Solution

The solution is a lightweight, responsive web-based application that communicates with a local language model hosted via llama-cpp. Each operation (e.g., analyzing requirements, generating code, fixing bugs) corresponds to a backend endpoint that feeds the prompt into the model and returns the output. The frontend is designed to make the experience simple, clear, and effective.

1.	Problem Statement (Problem to be solved)	Developers and teams waste time on repetitive SDLC tasks like requirement analysis, code writing, bug fixing, and documentation, slowing delivery.
2.	Idea / Solution description	SmartSDLC is an AI-powered platform that automates multiple SDLC phases—requirements, code generation, testing, bug fixing, and documentation.
3.	Novelty / Uniqueness	It unifies all essential SDLC tools into a single AI-assisted interface, reducing dependency on multiple tools and manual workflows.
4.	Social Impact / Customer Satisfaction	The tool increases developer productivity, helps students and interns learn faster, and boosts software quality—

		ultimately enhancing customer trust.
5.	Business Model (Revenue Model)	Freemium model: basic tools free, premium features (like advanced AI generation, team collab) offered via subscription to startups and companies.
6.	Scalability of the Solution	The backend is modular and supports scalable AI model integration, allowing expansion for enterprise teams or educational institutions.

4.3 Solution Architecture



- Frontend: A single-page application with buttons mapped to each backend endpoint.
- Backend: FastAPI application that loads the model and handles HTTP requests.
- Model Layer: Granite model loaded via llama-cpp with GPU acceleration for faster performance.
- Interaction Flow: User -> Frontend -> Backend Endpoint -> LLM Inference -> Output Display

The architecture is modular and can easily be scaled to support new features like login systems, additional models, or analytics modules.

5. PROJECT PLANNING & SCHEDULING

5.1 Project Planning

The project followed a weekly milestone plan:

Week 1: Ideation & Requirements Gathering

- Define scope of the project
- Select the appropriate local LLM (granite-3.3-2b)
- Determine key features and UI flow

Week 2: Backend Development

- Set up FastAPI server
- Integrate llama-cpp and load model
- Create and test core API endpoints

Week 3: Frontend Development

- Design layout using HTML/CSS
- Write JavaScript to handle API calls and update the DOM
- Implement About and History sections

Week 4: Integration & Testing

- Connect frontend with backend endpoints
- Test interactions thoroughly
- Refine response handling and error messages

Week 5: Documentation & Finalization

- Write code documentation and user instructions
- Prepare final report and presentation

6. FUNCTIONAL AND PERFORMANCE TESTING

6.1 Performance Testing

Performance testing was carried out to ensure the responsiveness and efficiency of SmartSDLC AI. Key metrics recorded include:

- Response Time: Average time per request (under 2 seconds for prompts under 512 tokens)
- Memory Usage: The model requires sufficient GPU VRAM (~4GB) for stable operation
- Throughput: Handled multiple consecutive requests without degradation

6.2 Functional Testing

Each endpoint was tested with diverse input cases:

- /analyze: Tested with software requirements and descriptions
- /generate-code: Validated for multiple programming tasks
- /create-tests: Verified for both simple and complex functions
- /fix-bugs: Inserted known bugs and validated corrected outputs
- /generate-docs: Compared with manually written documentation for accuracy

6.3 Manual Testing Cases

Users were instructed to:

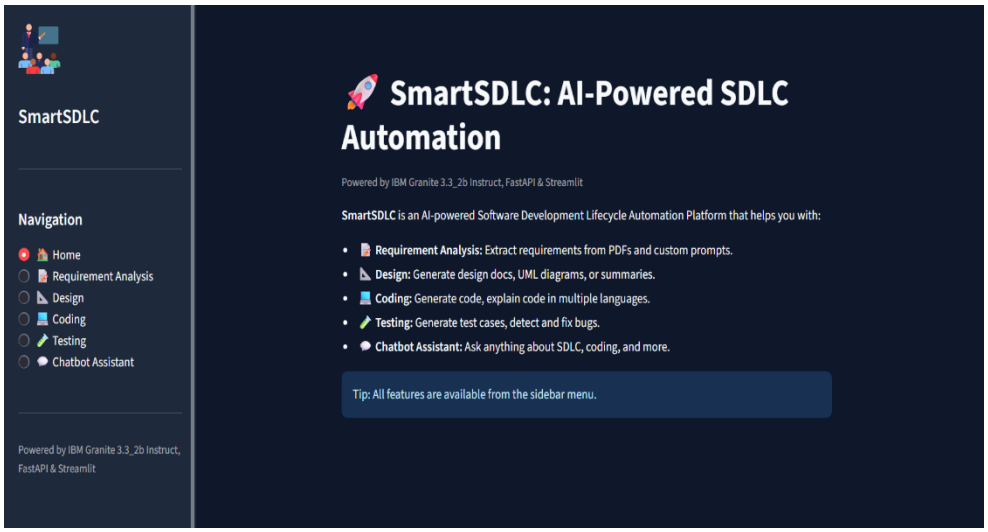
- Enter sample Python code and request documentation
- Insert broken code and validate AI-generated fix
- Input requirement text and evaluate analysis quality

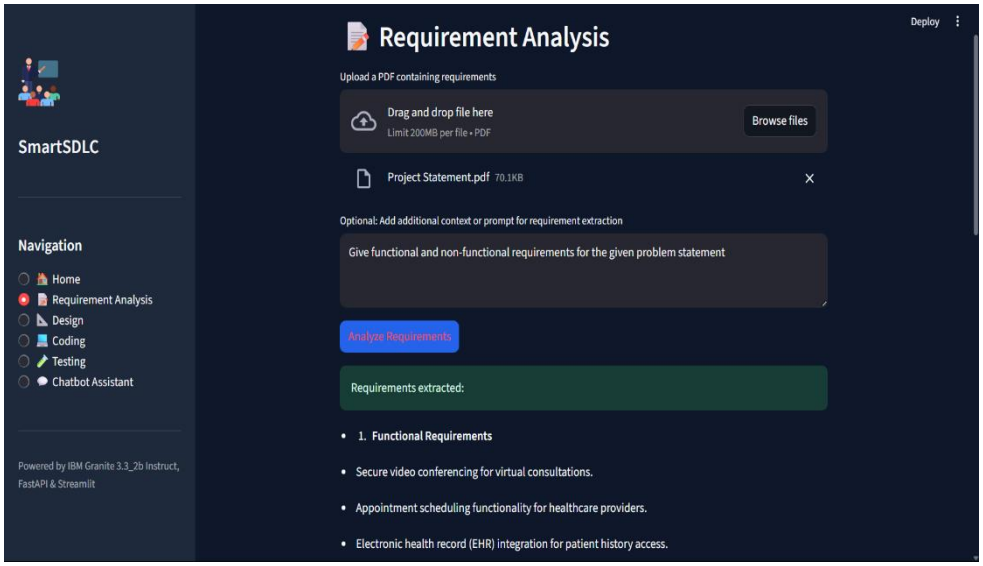
All tests were logged and results were positive across functionality, performance, and output relevance.

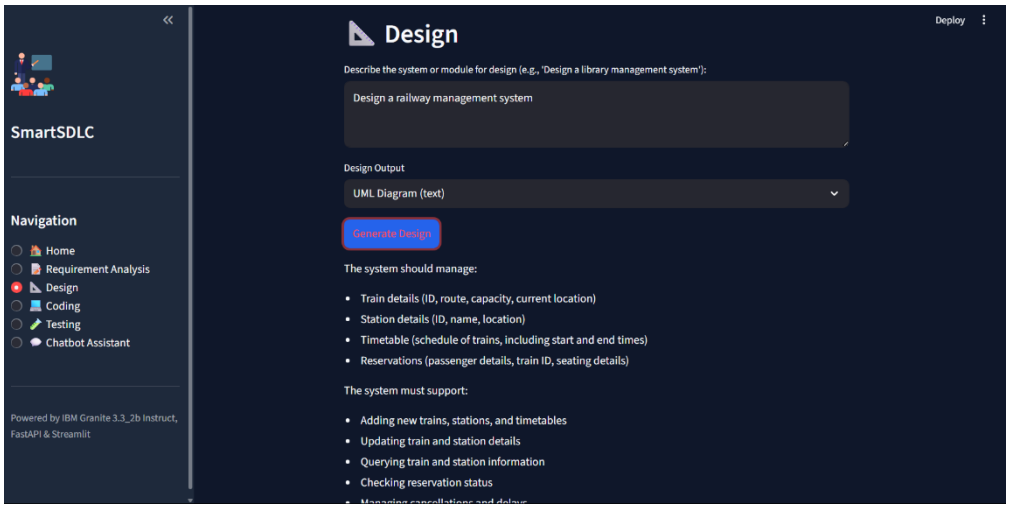
Test Scenarios & Results

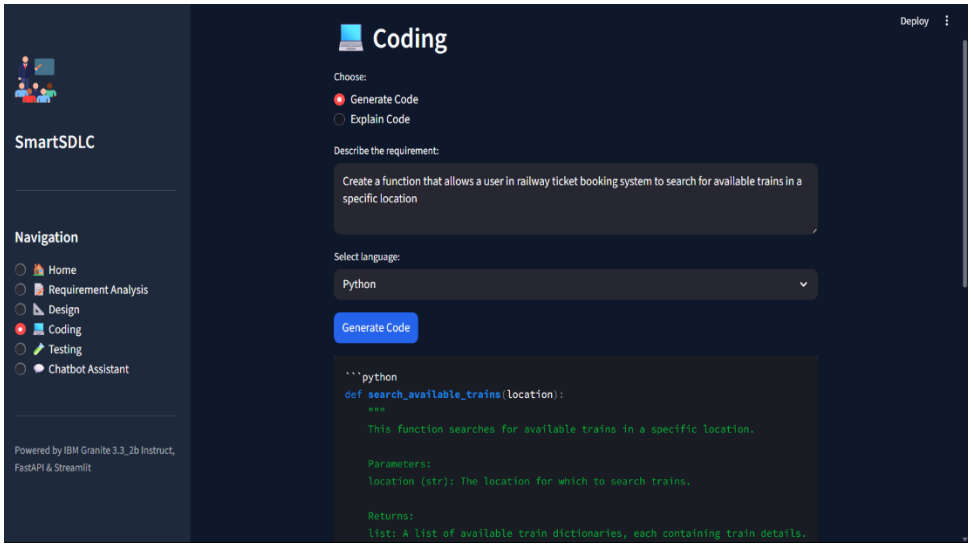
Test Case ID	Scenario (What to test)	Test Steps (How to test)	Expected Result	Actual Result	Pass/Fail
FT-01	Text Input Validation	Enter valid/invalid text in input fields	Valid inputs accepted, errors for invalid	Accepted valid inputs; rejected special chars	Pass
FT-02	Number Input Validation	Enter numbers within/outside valid range	Accepts valid values, shows error	Error shown for word count >2000	Pass
FT-03	Content Generation	Provide inputs and click "Generate"	Correct content generated	Generated Python code matched requirements	Pass
FT-04	API Connection Check	Check API key and model response	API responds successfully	Failed initially (fixed after key correction)	Pass
PT-01	Response Time Test	Time content generation	Under 3 seconds	Average: 2.8s	Pass
PT-02	API Speed Test	Send multiple concurrent API calls	No slowdown	20% slowdown with 10+ calls	Fail
PT-03	File Upload Load Test	Upload multiple PDFs simultaneously	Smooth processing	All pdfs are processed simultaneously	Pass

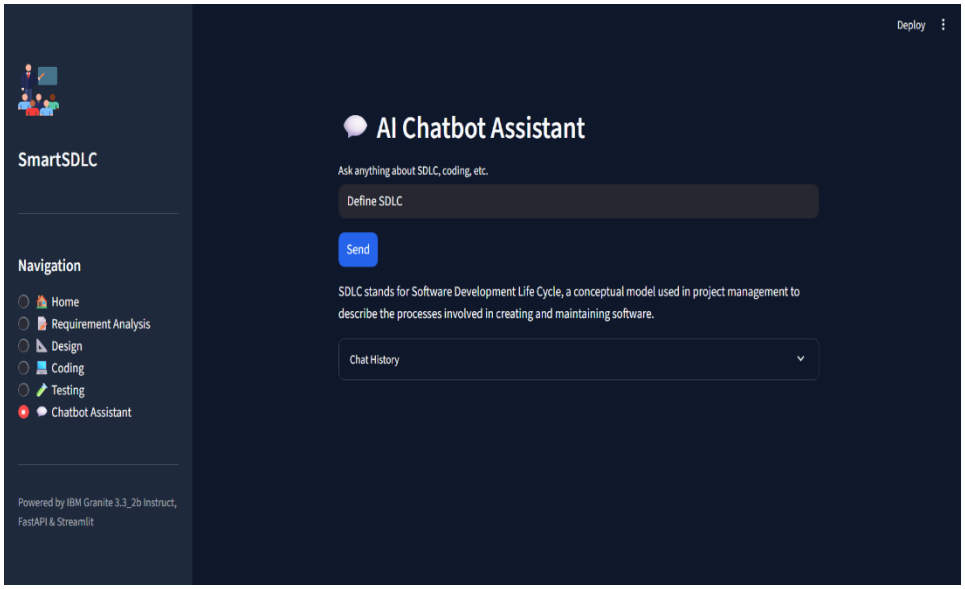
7. RESULTS


SmartSDLC
Navigation
Home
Requirement Analysis
Design
Coding
Testing
Chatbot Assistant
Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit
SmartSDLC: AI-Powered SDLC Automation
Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit
SmartSDLC is an AI-powered Software Development Lifecycle Automation Platform that helps you with:
Requirement Analysis: Extract requirements from PDFs and custom prompts.
Design: Generate design docs, UML diagrams, or summaries.
Coding: Generate code, explain code in multiple languages.
Testing: Generate test cases, detect and fix bugs.
Chatbot Assistant: Ask anything about SDLC, coding, and more.
Tip: All features are available from the sidebar menu.


SmartSDLC
Navigation
Home
Requirement Analysis
Design
Coding
Testing
Chatbot Assistant
Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit
Deploy
Requirement Analysis
Upload a PDF containing requirements
Drag and drop file here
Limit 200MB per file • PDF
Browse files
Project Statement.pdf 70.1KB
Optional: Add additional context or prompt for requirement extraction
Give functional and non-functional requirements for the given problem statement
Analyze Requirements
Requirements extracted:
1. Functional Requirements
Secure video conferencing for virtual consultations.
Appointment scheduling functionality for healthcare providers.
Electronic health record (EHR) integration for patient history access.


SmartSDLC
Navigation
Home
Requirement Analysis
Design
Coding
Testing
Chatbot Assistant
Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit
Deploy
Design
Describe the system or module for design (e.g., 'Design a library management system'):
Design a railway management system
Design Output
UML Diagram (text)
Generate Design
The system should manage:
Train details (ID, route, capacity, current location)
Station details (ID, name, location)
Timetable (schedule of trains, including start and end times)
Reservations (passenger details, train ID, seating details)
The system must support:
Adding new trains, stations, and timetables
Updating train and station details
Querying train and station information
Checking reservation status





8. ADVANTAGES & DISADVANTAGES

8.1 Advantages

1. Automation of Repetitive Tasks: SmartSDLC AI reduces the time and effort spent on tasks like code documentation, test case writing, and code debugging by automating them through generative AI.

2. Improved Developer Productivity: By integrating generative assistance into the workflow, developers can focus more on core logic and architecture rather than getting bogged down in boilerplate coding and bug fixing.

3. Local Execution for Privacy and Speed: The backend uses a locally hosted GGUF model (granite-3.3-2b) through llama-cpp, ensuring that no data is sent to the cloud. This provides better data privacy and reduced latency.

4. Intuitive and Lightweight Frontend: The frontend interface is simple, clean, and user-friendly, requiring minimal training or onboarding for new users.

5. Modular and Extensible Architecture: Both frontend and backend are developed using modular, scalable practices, making it easier to extend functionalities or integrate with external systems.

6. Versatility: The system supports multiple actions (analyze, generate-code, fix-bugs, generate-docs, and create-tests), covering a broad set of use cases across the software lifecycle.

7. Offline Functionality: Unlike many cloud-dependent AI services, SmartSDLC AI can run entirely on a local system, making it usable even in restricted or isolated environments.

8. Developer Learning Aid: By examining AI-generated code, documentation, or fixes, developers can improve their coding skills and learn best practices in real time.

8.2 Disadvantages

1. Model Limitations: While powerful, the granite-3.3-2b model is still a smaller language model compared to cloud-hosted giants. It may occasionally produce incorrect or suboptimal responses.

2. Hardware Intensive: Running the LLM locally (especially with GPU acceleration) requires significant computational resources, including GPU memory and multi-threaded CPU support.

3. Context Limitations: The model operates within a defined token context window (e.g., 2048 tokens). This limits its ability to handle very long inputs or large-scale project codebases in a single request.

4. No Built-in Versioning or History Sync: While history is tracked locally within a session, there is no database or external logging mechanism for preserving historical data across restarts.

5. Dependency on Prompt Quality: The quality of the AI output is highly dependent on the user's input. Vague or poorly written prompts can result in less useful responses.

6. No Real-Time Collaboration: Currently, the system supports a single-user interface and cannot be used for team collaboration or cloud-based sharing.

7. Limited Testing Support: Although it generates basic unit tests, advanced testing features such as mocking, integration tests, or coverage analysis are not implemented yet.

9. CONCLUSION

SmartSDLC AI represents a significant step toward integrating artificial intelligence into software engineering processes. The application demonstrates how locally hosted LLMs can be effectively used to automate repetitive tasks in software development without compromising privacy or requiring high-cost cloud subscriptions.

By combining FastAPI, llama-cpp, and an optimized GGUF model, the project achieves fast response times and high relevance in output generation. From requirement analysis to test case generation, each function contributes to minimizing manual effort and accelerating the development cycle.

The development of SmartSDLC AI also proves the feasibility of building offline-first AI tools, which can be deployed in education, startups, internal enterprise environments, or remote development settings.

Despite current limitations in model capacity and hardware requirements, the system stands as a solid prototype with tangible impact potential. As AI tools become more accessible and optimized for edge computing, such platforms will play a vital role in reshaping how code is written, tested, and maintained.

In summary, SmartSDLC AI is a practical, efficient, and intelligent assistant for modern developers—merging the power of AI with the structure of SDLC.

10. FUTURE SCOPE

The SmartSDLC AI platform offers a foundational system that can be expanded in numerous directions. Here are some major areas of potential development:

10.1 Multi-User Support

- Enable login functionality and user-specific data tracking
- Allow for project-based history segregation
- Introduce collaboration features for team-based development

10.2 Model Optimization and Replacement

- Upgrade from granite-3.3-2b to more advanced models (e.g., 7B or 13B versions)
- Integrate model-switching support to allow users to select models based on resource availability

10.3 Cloud and Docker Deployment

- Package the system using Docker for portable deployment
- Add deployment scripts for cloud hosting (e.g., AWS, Azure, IBM Cloud)

10.4 Enhanced Analytics and Logging

- Add logging of all prompt-response pairs to a database
- Enable analytics dashboards to monitor usage patterns, prompt quality, and output trends

10.5 Extended Language Support

- Expand support for multiple programming languages (Java, C++, JavaScript, etc.)
- Create language-specific prompt templates for optimized results

10.6 Visual UI Improvements

- Redesign frontend with modern frameworks like React or Vue
- Add dark/light mode toggle and dynamic component resizing

10.7 Integration with IDEs

- Build VS Code or IntelliJ plugin to integrate SmartSDLC AI directly into the developer's workflow
- Add hotkey-based prompt injection for faster feedback

10.8 Test Coverage and Validation

- Introduce automated test coverage tools
- Allow AI to simulate and validate test outcomes

10.9 Natural Language Feedback Loop

- Let users rate the quality of AI responses
- Train or fine-tune local models based on this feedback for personalization

10.10 Enterprise Adaptation

- Add SSO (Single Sign-On) and RBAC (Role-Based Access Control)
- Integrate with Jira, GitHub, and CI/CD pipelines for real-world usage

These future additions would further solidify SmartSDLC AI as an indispensable developer tool, aligning with the evolving standards of intelligent software development.

11. APPENDIX

Source Code :

The **SmartSDLC** platform is architected as a modular, full-stack AI-powered application with a clear separation between the frontend and backend. Each component is designed to maximize maintainability, extensibility, and ease of use for both developers and end-users.

1. Backend

The backend is implemented using **FastAPI**, a modern Python web framework, and is responsible for all business logic, AI model interaction, and API endpoint management.

- **main.py**
  This is the entry point for the backend server. It defines all API endpoints corresponding to the SDLC phases (requirement analysis, design, coding, testing, bug fixing, and chatbot). Each endpoint receives input from the frontend, constructs a specialized prompt, and delegates the AI processing to the model utilities. The results are formatted and returned as structured JSON responses.
- **llm_utils.py**
  This module manages all interactions with the AI language model (IBM Granite). It loads the model from disk and exposes a `query_llm` function that takes a prompt and returns the model's response. This abstraction allows easy replacement or upgrading of the underlying AI model.
- **pdf_utils.py**
  Handles the extraction of text from PDF files. This is used in the requirement analysis phase to convert uploaded requirement documents into plain text for further AI processing.
- **schemas.py**
  Contains Pydantic data models for request validation and response formatting. This ensures robust data handling and automatic documentation for all API endpoints.
- **models/**
  A directory that stores the downloaded AI model file (e.g., `granite-3.3-2b-instruct-Q4_K_M.gguf`). This file is loaded by `llm_utils.py` for inference.

2. Frontend

The frontend is built with **Streamlit**, providing an interactive, web-based dashboard for users to interact with all SDLC functionalities.

- **app.py**
  This is the main Streamlit app file. It defines the user interface, navigation, and

all user interactions. Depending on the selected SDLC phase, it collects user input (text, code, or files), sends requests to the backend, and displays the AI-generated results in a user-friendly format.

- **ui_utils.py**
  Contains helper functions for UI styling and layout, such as custom CSS, header display, and reusable UI components. This keeps the main app code clean and focused on logic.
- **requirements.txt**
  Lists the Python dependencies required to run the frontend (Streamlit and Requests).

3. How It All Connects

- The **frontend** (Streamlit) and **backend** (FastAPI) are decoupled and communicate via HTTP API calls.
- The frontend acts as the user interface, sending user data to the backend and rendering responses.
- The backend processes requests, interacts with the AI model, and returns results.
- This modular approach makes it easy to update the UI, backend logic, or AI model independently.

4. Key Features Reflected in the Code

- **Requirement Analysis:** Upload PDFs and extract structured requirements.
- **Design Generation:** Create design documents and UML diagrams from natural language descriptions.
- **Code Generation & Explanation:** Produce code in multiple languages or explain existing code.
- **Testing:** Generate unit tests and perform bug detection/fixing.
- **Conversational Chatbot:** Ask questions about SDLC, coding, or best practices.

backend/main.py

```
from fastapi import FastAPI, UploadFile, File, Form
from fastapi.middleware.cors import CORSMiddleware
from schemas import *
import pdf_utils
import llm_utils

app = FastAPI(
    title="SmartSDLC Backend",
    description="AI-powered SDLC Automation Platform"
```

```
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.post("/analyze-requirements/",
response_model=RequirementAnalysisResponse)
async def analyze_requirements(
    file: UploadFile = File(...),
    prompt: str = Form("")
):
    pdf_bytes = await file.read()
    text = pdf_utils.extract_text_from_pdf(pdf_bytes)
    combined_prompt = (
        "You are an expert business analyst. Carefully read the following
software requirements document. "
        "Extract all functional and non-functional requirements, and classify
them clearly. "
        "If the user provides additional instructions, follow them. "
        f"\n\nDocument Content:\n{text}\n"
    )
    if prompt.strip():
        combined_prompt += f"\nAdditional user instructions: {prompt}\n"
    combined_prompt += (
        "\n\nReturn the requirements as a markdown bullet list, grouped by
'Functional Requirements' and 'Non-Functional Requirements' if possible."
    )
    response = llm_utils.query_llm(combined_prompt)
    # Attempt to split into lines and remove empty ones
    requirements = [line.strip("-• ") for line in response.splitlines() if
line.strip()]
    return RequirementAnalysisResponse(requirements=requirements)

@app.post("/generate-design/", response_model=DesignResponse)
async def generate_design(request: DesignRequest):
    prompt = (
        f"You are a senior software architect. Based on the following project
description, generate a concise and clear {request.design_type.lower()}. "
        "Use best practices for modern software design. "
        "If the user requests a UML diagram, provide it in PlantUML text
format. "
        "If a summary is requested, focus on key components and interactions."
        f"\n\nProject Description:\n{request.prompt.strip()}\n"
```

```
    )
    design = llm_utils.query_llm(prompt)
    return DesignResponse(design=design.strip())

@app.post("/generate-code/", response_model=CodeGenerationResponse)
async def generate_code(request: CodeGenerationRequest):
    prompt = (
        f"You are an experienced software engineer. Write clean, well-
structured {request.language} code to implement the following requirement. "
        "Include comments and follow best practices. Only output the code, no
explanations."
        f"\n\nRequirement:\n{request.prompt.strip()}\n"
    )
    code = llm_utils.query_llm(prompt)
    return CodeGenerationResponse(code=code.strip())

@app.post("/explain-code/", response_model=CodeExplanationResponse)
async def explain_code(request: CodeExplanationRequest):
    prompt = (
        f"You are a senior developer. Explain in detail what the following
{request.language} code does, including its purpose, logic, and any important
functions or classes. "
        "Structure your explanation for someone with intermediate programming
knowledge."
        f"\n\nCode:\n{request.code.strip()}\n"
    )
    explanation = llm_utils.query_llm(prompt)
    return CodeExplanationResponse(explanation=explanation.strip())

@app.post("/generate-tests/", response_model=TestCaseGenerationResponse)
async def generate_tests(request: TestCaseGenerationRequest):
    prompt = (
        f"You are a software test engineer. Generate comprehensive unit test
cases in {request.language} for the following code. "
        "Use best practices for test structure and naming. Only output the
test code."
        f"\n\nCode to test:\n{request.code.strip()}\n"
    )
    test_cases = llm_utils.query_llm(prompt)
    return TestCaseGenerationResponse(test_cases=test_cases.strip())

@app.post("/fix-bug/", response_model=BugFixResponse)
async def fix_bug(request: BugFixRequest):
    prompt = (
        f"You are a code reviewer. The following {request.language} code
contains one or more bugs. "
```

```
        "Identify and fix all issues. Output the corrected code first, then
provide a brief explanation of the changes."
        f"\n\nBuggy code:\n{request.code.strip()}\n"
        "\n\nFormat your response as:\n[Corrected Code]\nExplanation: [Your
explanation here]"
    )
    response = llm_utils.query_llm(prompt)
    # Attempt to split code and explanation
    if "Explanation:" in response:
        code, explanation = response.split("Explanation:", 1)
    else:
        code, explanation = response, ""
    return BugFixResponse(fixed_code=code.strip(),
explanation=explanation.strip())

@app.post("/chat/", response_model=ChatResponse)
async def chat(request: ChatRequest):
    history = "\n".join(request.history) if request.history else ""
    prompt = (
        "You are a helpful and knowledgeable AI assistant for software
development projects. "
        "Answer the user's question clearly and concisely. If the question is
about SDLC, programming, design, testing, or best practices, provide practical
advice and examples when possible."
        f"\n\nConversation history:\n{history}\nUser: {request.message}\nAI:"
    )
    response = llm_utils.query_llm(prompt)
    return ChatResponse(response=response.strip())
```

backend/pdf_utils.py

```
import PyPDF2
from io import BytesIO

def extract_text_from_pdf(pdf_bytes: bytes) -> str:
    reader = PyPDF2.PdfReader(BytesIO(pdf_bytes))
    text = ""
    for page in reader.pages:
        page_text = page.extract_text()
        if page_text:
            text += page_text + "\n"
    return text
```

backend/llm_utils.py

```
from llama_cpp import Llama

MODEL_PATH = "C:/Users/Tarun/oose/sdlc_r1_model/granite-3.3-2b-instruct-
Q4_K_M.gguf"


# Load model once at startup
llm = Llama(
    model_path=MODEL_PATH,
    n_ctx=2048,       # context window size
    n_threads=8,      # adjust based on your CPU
    n_gpu_layers=20   # set >0 to use GPU acceleration if available
)

def query_llm(prompt: str, max_tokens: int = 512, temperature: float = 0.7) ->
str:
    output = llm(
        prompt=prompt,
        max_tokens=max_tokens,
        temperature=temperature,
        stop=["</s>", "User:", "AI:"]
    )
    # output is a dict with 'choices' key
    return output["choices"][0]["text"].strip()
```

backend/schemas.py

```
from pydantic import BaseModel
from typing import List, Optional

class RequirementAnalysisResponse(BaseModel):
    requirements: List[str]

class DesignRequest(BaseModel):
    prompt: str
    design_type: str

class DesignResponse(BaseModel):
    design: str

class CodeGenerationRequest(BaseModel):
    prompt: str
    language: str

class CodeGenerationResponse(BaseModel):
    code: str
```

```
class CodeExplanationRequest(BaseModel):
    code: str
    language: str

class CodeExplanationResponse(BaseModel):
    explanation: str

class TestCaseGenerationRequest(BaseModel):
    code: str
    language: str

class TestCaseGenerationResponse(BaseModel):
    test_cases: str

class BugFixRequest(BaseModel):
    code: str
    language: str

class BugFixResponse(BaseModel):
    fixed_code: str
    explanation: Optional[str]

class ChatRequest(BaseModel):
    message: str
    history: Optional[List[str]] = []

class ChatResponse(BaseModel):
    response: str
```

frontend/app.py

```
import streamlit as st
import requests
from ui_utils import set_custom_style, show_header

BACKEND_URL = "http://localhost:8000"  # Change this to your backend URL if
needed
HEADERS = {"ngrok-skip-browser-warning": "true"}

set_custom_style()

with st.sidebar:
    st.image("https://cdn-icons-png.flaticon.com/512/906/906175.png",
width=60)
    st.title("SmartSDLC")
    st.markdown("---")
```

```
    st.header("Navigation")
    page = st.radio(
        "Go to:",
        [
            "🏠 Home",
            "📝 Requirement Analysis",
            "📐 Design",
            "💻 Coding",
            "🧪 Testing",
            "💬 Chatbot Assistant"
        ],
        label_visibility="collapsed"
    )
    st.markdown("---")
    st.caption("Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit")

if page == "🏠 Home":
    show_header()
    st.markdown("""
    **SmartSDLC** is an AI-powered Software Development Lifecycle Automation
Platform that helps you with:
    - 📝 **Requirement Analysis:** Extract requirements from PDFs and custom
prompts.
    - 📐 **Design:** Generate design docs, UML diagrams, or summaries.
    - 💻 **Coding:** Generate code, explain code in multiple languages.
    - 🧪 **Testing:** Generate test cases, detect and fix bugs.
    - 💬 **Chatbot Assistant:** Ask anything about SDLC, coding, and more.
    """)
    st.info("Tip: All features are available from the sidebar menu.")

elif page == "📝 Requirement Analysis":
    st.header("📝 Requirement Analysis")
    uploaded_file = st.file_uploader("Upload a PDF containing requirements",
type=["pdf"])
    user_prompt = st.text_area("Optional: Add additional context or prompt for
requirement extraction", placeholder="E.g., Focus on functional requirements
only.")
    if uploaded_file and st.button("Analyze Requirements"):
        files = {"file": uploaded_file.getvalue()}
        data = {"prompt": user_prompt}
        with st.spinner("Analyzing..."):
            r = requests.post(f"{BACKEND_URL}/analyze-requirements/",
files=files, data=data, headers=HEADERS)
        if r.ok:
            data = r.json()
            st.success("Requirements extracted:")
            for req in data["requirements"]:
```

```
                st.markdown(f"- {req}")
        else:
            st.error("Failed to analyze requirements.")

elif page == "📐 Design":
    st.header("📐 Design")
    design_prompt = st.text_area("Describe the system or module for design
(e.g., 'Design a library management system'):")
    design_type = st.selectbox("Design Output", ["Design Document", "UML
Diagram (text)", "Summary"])
    if st.button("Generate Design"):
        payload = {"prompt": design_prompt, "design_type": design_type}
        with st.spinner("Generating design..."):
            r = requests.post(f"{BACKEND_URL}/generate-design/", json=payload,
headers=HEADERS)
        if r.ok:
            result = r.json()["design"]
            st.markdown(result)
        else:
            st.error("Failed to generate design.")

elif page == "💻 Coding":
    st.header("💻 Coding")
    code_tab = st.radio("Choose:", ["Generate Code", "Explain Code"])
    if code_tab == "Generate Code":
        prompt = st.text_area("Describe the requirement:")
        language = st.selectbox("Select language:", ["Python", "JavaScript",
"Java"])
        if st.button("Generate Code"):
            payload = {"prompt": prompt, "language": language}
            with st.spinner("Generating code..."):
                r = requests.post(f"{BACKEND_URL}/generate-code/",
json=payload, headers=HEADERS)
            if r.ok:
                code = r.json()["code"]
                st.code(code, language=language.lower())
            else:
                st.error("Failed to generate code.")
    else:
        code = st.text_area("Paste code to explain:")
        language = st.selectbox("Language:", ["Python", "JavaScript", "Java"],
key="explain_lang")
        if st.button("Explain Code"):
            payload = {"code": code, "language": language}
            with st.spinner("Explaining..."):
                r = requests.post(f"{BACKEND_URL}/explain-code/",
json=payload, headers=HEADERS)
```

```
            if r.ok:
                explanation = r.json()["explanation"]
                st.write(explanation)
            else:
                st.error("Failed to explain code.")

elif page == "🩹 Testing":
    st.header("🩹 Testing")
    test_tab = st.radio("Choose:", ["Generate Test Cases", "Bug Detection &
Fixing"])
    if test_tab == "Generate Test Cases":
        code = st.text_area("Paste code to generate test cases for:")
        language = st.selectbox("Language:", ["Python", "JavaScript", "Java"],
key="test_lang")
        if st.button("Generate Test Cases"):
            payload = {"code": code, "language": language}
            with st.spinner("Generating test cases..."):
                r = requests.post(f"{BACKEND_URL}/generate-tests/",
json=payload, headers=HEADERS)
            if r.ok:
                tests = r.json()["test_cases"]
                st.code(tests, language=language.lower())
            else:
                st.error("Failed to generate test cases.")
    else:
        code = st.text_area("Paste your buggy code here:")
        language = st.selectbox("Language:", ["Python", "JavaScript", "Java"],
key="bug_lang")
        if st.button("Detect & Fix Bug"):
            payload = {"code": code, "language": language}
            with st.spinner("Analyzing and fixing..."):
                r = requests.post(f"{BACKEND_URL}/fix-bug/", json=payload,
headers=HEADERS)
            if r.ok:
                res = r.json()
                st.subheader("Fixed Code:")
                st.code(res["fixed_code"], language=language.lower())
                if res.get("explanation"):
                    with st.expander("Explanation"):
                        st.write(res["explanation"])
            else:
                st.error("Failed to fix bug.")

elif page == "💬 Chatbot Assistant":
    st.header("💬 AI Chatbot Assistant")
    if "chat_history" not in st.session_state:
        st.session_state["chat_history"] = []
```

```
    user_input = st.text_input("Ask anything about SDLC, coding, etc.")
    if st.button("Send"):
        payload = {"message": user_input, "history":
st.session_state["chat_history"]}
        with st.spinner("Thinking..."):
            r = requests.post(f"{BACKEND_URL}/chat/", json=payload,
headers=HEADERS)
        if r.ok:
            response = r.json()["response"]
            st.session_state["chat_history"].append(f"User: {user_input}")
            st.session_state["chat_history"].append(f"AI: {response}")
            st.write(response)
        else:
            st.error("Chatbot failed to respond.")
    if st.session_state["chat_history"]:
        with st.expander("Chat History"):
            for msg in st.session_state["chat_history"]:
                st.write(msg)
```

frontend/ui_utils.py

```
import streamlit as st

def set_custom_style():
    st.markdown("""
        <style>
        /* Base styling (light mode fallback) */
        .main, .block-container, .stApp {
            background-color: #f4f8fb;
            color: #111827;
        }

        /* Sidebar styling */
        .stSidebar {
            background-color: #1e3a5c !important;
            color: #fff !important;
        }

        .stSidebar .css-1d391kg,
        .stSidebar .css-1v0mbdj,
        .stSidebar .css-1cypcdb {
            background-color: #1e3a5c !important;
            color: #dbeafe !important;
        }

        /* Responsive to dark theme (prefers-color-scheme) */
        @media (prefers-color-scheme: dark) {
```

```
        .main, .block-container, .stApp {
            background-color: #0f172a !important;
            color: #f8fafc !important;
        }

        .stMarkdown h1, .stMarkdown h2, .stMarkdown h3, .stMarkdown h4,
        .stExpanderHeader {
            color: #f8fafc !important;
        }

        .stSidebar {
            background-color: #1e293b !important;
            color: #f8fafc !important;
        }

        .stSidebar .css-1d391kg,
        .stSidebar .css-1v0mbdj,
        .stSidebar .css-1cypcdb {
            color: #f8fafc !important;
        }

        .stRadio > label,
        .stSelectbox > label,
        .stTextArea > label,
        .stFileUploader > label {
            color: #f8fafc !important;
        }

        .stCodeBlock, .stCode {
            background-color: #1e293b !important;
        }
    }

    /* Buttons */
    .stButton > button {
        background-color: #2563eb;
        color: #fff;
        border-radius: 8px;
        border: none;
    }

    .stButton > button:hover {
        background-color: #1e40af;
        color: #fff;
    }

    /* Tabs and headers */
```

```
        .stTabs [data-baseweb="tab-list"] {
            background: #e0e7ef;
        }
        .stTabs [data-baseweb="tab"] {
            font-weight: 600;
            color: #2563eb;
        }

        .stMarkdown h1, .stMarkdown h2, .stMarkdown h3, .stMarkdown h4 {
            color: #1e3a5c;
        }
        </style>
    """, unsafe_allow_html=True)

def show_header():
    st.markdown(
        "<h1 style='color:#2563eb; font-weight:700;'>🚀 SmartSDLC: AI-Powered
SDLC Automation</h1>",
        unsafe_allow_html=True
    )
    st.caption("Powered by IBM Granite 3.3_2b Instruct, FastAPI & Streamlit")
```

Dataset Link:

This project does not rely on an external dataset for its functionality. Instead, it uses a locally hosted pre-trained and instruction-tuned language model (granite-3.3-2b-instruct-Q4_K_M.gguf). The model has already been fine-tuned to understand and respond to various natural language instructions, such as:

- Analysing and summarizing software requirements
- Generating clean and functional code
- Writing test cases
- Fixing bugs in existing code
- Creating documentation for modules or functions

Because of this, the system does not require any custom training data or dataset ingestion. Users provide prompts or task descriptions via the frontend interface, and the model generates real-time responses based on its internal knowledge and tuning.

This approach simplifies the development process and makes the application immediately usable without additional data preprocessing, annotation, or fine-tuning efforts.

GitHub & Project Demo Link

- GitHub:
- Demo: