# SmartSDLC-AI — Enhanced Software Development Lifecycle

Generated on: 2025-09-13 08:13:21

Author: Project Team — SmartSDLC-AI

## Executive Summary

SmartSDLC-AI is an integrated application designed to streamline the software development lifecycle  
by leveraging generative AI for requirement analysis, code generation, and developer assistance.  
This document outlines the model selection, architecture, core functionalities, implementation plan,  
frontend and backend design, deployment steps, testing strategy, milestones, and conclusion.

## Table of Contents

1. Introduction

2. Objectives

3. Model Selection and Architecture

4. Core Functionalities

5. Implementation Details

6. Frontend Design (Gradio + React options)

7. Backend Design (FastAPI)

8. Deployment & Testing

9. Timeline & Milestones

10. Conclusion

11. Appendix: Sample Code and Configurations

## 1. Introduction

Modern software development benefits from automation and intelligent assistance. SmartSDLC-AI aims  
to provide teams with rapid requirement extraction from documents, automated code generation  
templates, and a developer-friendly interface for iterative refinement. The application integrates a  
generative language model with a lightweight API backend and an interactive frontend to create an  
efficient productivity toolchain for software engineers, product managers, and QA teams.

## 2. Objectives

- Accelerate requirement analysis by extracting functional, non-functional, and technical requirements from documents.

- Provide safe, explainable code-generation templates for common tasks.

- Offer a user-friendly interface for uploading documents, editing prompts, and retrieving generated artifacts.

- Support a modular architecture that allows model swapping, scaling, and secure deployment.

## 3. Model Selection and Architecture

Milestone 1: Model Selection and Architecture

Activity 1.1: Research and select the appropriate generative AI model

Choose a model that balances capability, latency, cost, and licensing. Options include open-source  
models like Llama family, Mistral, or Avalon-style models and hosted options from cloud providers.  
For this prototype we used an on-device/hosted medium-sized instruct-tuned model (example: 'ibm-  
granite/granite-3.2-2b-instruct') as a placeholder—swap to other models depending on production  
constraints.

Activity 1.2: Define the architecture of the application

The application is organized into three main layers: 1. Frontend: Gradio for quick prototyping or a  
React + Tailwind UI for production-grade UX. 2. Backend API: FastAPI to handle requests, orchestrate  
model calls, manage file uploads, and enforce rate limits. 3. Model Layer: A pluggable inference  
service wrapping the chosen generative model with caching and safety filters. Supporting services  
include logging, authentication, storage (for uploaded PDFs), and CI/CD pipelines for deployment.

## 4. Core Functionalities

Milestone 2: Core Functionalities Development

- Requirement extraction: Parse PDFs and free-text to extract functional, non-functional, and technical requirements.

- Code generation: Generate starter code in multiple languages (Python, JavaScript, Java, C++, C#, PHP, Go, Rust).

- Interactive UI: Allow users to refine prompts, re-run analysis, and download outputs.

- Project scaffolding: Produce file and folder templates for common project types (web, API, data pipeline).

## 5. Implementation Details

This section outlines example code, functions, and file structure for SmartSDLC-AI.

5.1 Code snippets (selected excerpts):

# Example: model wrapper and generation helper (pseudo)  
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,  
 pad\_token\_id=tokenizer.eos\_token\_id  
 )  
 response = tokenizer.decode(outputs[0], skip\_special\_tokens=True)  
 return response

5.2 File structure (recommended):

smart-sdlc-ai/

├─ backend/

│ ├─ main.py (FastAPI app)

│ ├─ models.py (model wrapper)

│ ├─ requirements.txt

│ └─ utils/ (pdf, parsing, auth)

├─ frontend/

│ ├─ app.py (Gradio prototype) or src/ (React app)

├─ infra/

│ ├─ Dockerfile

│ ├─ docker-compose.yml

│ └─ k8s/ (manifests)

└─ docs/

## 6. Frontend Development

Milestone 4: Frontend Development

For rapid prototyping, Gradio offers a minimal path to a working UI. For production, build a React +  
Tailwind UI with components for file upload, prompt editor, results viewer, and code download. Make  
sure to include input validation, progress indicators, and a way to view model usage logs for  
debugging.

## 7. Backend Design (FastAPI)

Milestone 2 & 3: Backend and main.py Development

Use FastAPI to implement endpoints such as: - POST /analyze (file or text) -> returns structured  
requirements - POST /generate-code -> returns code snippet or downloadable archive - GET /status ->  
health checks Implement authentication (JWT), input sanitization, rate limiting, and background  
tasks for long-running conversions.

## 8. Deployment & Testing

Milestone 5: Deployment

Local deployment steps: 1. Build and run Docker containers for backend and frontend. 2. Use docker-  
compose for development and a managed Kubernetes cluster for production. 3. Ensure persistent  
storage for uploaded files. Testing: - Unit tests for parsing utilities and model wrapper  
functions. - Integration tests for API endpoints. - Security tests for file uploads (scan for  
malicious PDFs). - Performance testing to estimate latency and concurrency limits.

## 9. Timeline & Milestones

Proposed milestones and activities:

- Milestone 1: Model Selection and Architecture — Estimated: 1 week

- Milestone 2: Core Functionalities Development — Estimated: 2-3 weeks

- Milestone 3: main.py Development — Estimated: 1 week

- Milestone 4: Frontend Development — Estimated: 2 weeks

- Milestone 5: Deployment — Estimated: 1 week

- Milestone 6: Conclusion & Documentation — Estimated: 1 week

## 10. Conclusion

SmartSDLC-AI is positioned to significantly reduce the time spent on early-stage requirement  
analysis and scaffold initial code artifacts to accelerate development. By maintaining a modular  
design and robust testing, teams can adopt this tooling with confidence and iterate on model and UI  
improvements over time.

## 11. Appendix: Sample Configurations and Commands

Sample Dockerfile (backend):  
FROM python:3.11-slim  
WORKDIR /app  
COPY requirements.txt ./  
RUN pip install -r requirements.txt  
COPY . .  
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]  
  
Sample docker-compose (excerpt):  
version: '3.8'  
services:  
 backend:  
 build: ./backend  
 ports: ['8000:8000']

### Additional Notes & Best Practices (Part 1)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.

### Additional Notes & Best Practices (Part 2)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.

### Additional Notes & Best Practices (Part 3)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.

### Additional Notes & Best Practices (Part 4)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.

### Additional Notes & Best Practices (Part 5)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.

### Additional Notes & Best Practices (Part 6)

Ensure model safety: implement prompt filters, response length limits, and a rejection policy for  
hallucinations. Monitor cost/usage and set guardrails for high-volume endpoints. For private data,  
encrypt uploads at rest and in transit. Maintain a changelog of model versions and prompt templates  
so behavior is auditable. Incorporate user feedback mechanisms and provide a way for users to report  
incorrect or unsafe outputs.