SmartSDLC – AI-Enhanced Software Development Lifecycle Project Documentation

1.Introduction

- Project title : Smart SDLC
- Team leader : Hemalatha S
- Team member : Vidyashree R
- Team member : Logamithra
- Team member : Nandeeshwari R

2.Project overview

- Purpose :

The primary purpose of a Smart SDLC is to leverage AI technologies to optimize each phase of the software development lifecycle. By embedding AI tools and algorithms across the SDLC stages, software teams can achieve faster development cycles, higher quality products, better risk management, and enhanced decision-making. Smart SDLC focuses on transforming traditional software engineering into a more predictive, adaptive, and intelligent process.

- Features:

1.Requirement Gathering

NLP: AI extracts and analyzes requirements from text and conversations.

Predictive Analysis: AI forecasts project complexity and risks.

2. Planning

Effort Estimation: AI predicts timelines and resource needs.

Risk Management: Machine learning identifies potential risks early.

3. Design

Auto Design Generation: AI creates system diagrams and design patterns.

Security Insights: AI flags design vulnerabilities.

4. Implementation (Coding)

AI Pair Programming: Tools like GitHub Copilot suggest code in realtime.

Code from Text: Converts natural language requirements into code.

Bug Detection: AI highlights code issues early.

5. Testing

Automated Test Generation: AI creates test cases based on requirements or code.

Bug Prioritization: Machine learning identifies and prioritizes critical bugs.

6. Deployment
Smart CI/CD: AI optimizes the deployment pipeline and detects failures.

Automated Rollbacks: AI triggers safe rollbacks in case of deployment failure.

7. Maintenance

Anomaly Detection: AI monitors and identifies system issues in real-time.

Self-Healing: Systems can auto-repair known issues without human intervention.

8. Documentation

Auto Documentation: AI generates code comments and API docs.

AI enhances SDLC by automating, predicting, and optimizing various stages for faster, safer, and more efficient software development

3. Architecture

Front-end(web dashboard)

The frontend is developed using Gradio or a simple web dashboard, providing developers and stakeholders with an interactive interface. It offers modules for requirements input, code review, test case generation, and project monitoring. Users can submit project specs, upload documents, or interact directly with AI assistants, and results are displayed instantly in structured panels.

Backend (Python / Orchestration)

The backend is built in Python and manages the core application logic. It handles requirement parsing, automated code generation, test creation, and

integration with repositories. It also coordinates communication between the user interface, AI models, and external tools (CI/CD, version control).

LLM Integration (AI Models)

The system integrates with large language models (e.g., IBM Granite, OpenAI, or Hugging Face models) for natural language understanding and generation. This layer converts requirements into user stories, suggests code, generates unit tests, and reviews commits. It ensures contextual responses and adapts outputs for different phases of the SDLC.

Analysis & Quality Layer

An automated quality layer performs static code analysis, security scanning, and test coverage evaluation. AI enhances bug detection, recommends fixes, and ensures compliance with coding standards.

Deployment Layer

The deployment layer is powered by CI/CD pipelines integrated with cloud or containerized platforms (Docker/Kubernetes). Applications can be deployed locally for testing or shared through public links, making the system flexible for demonstrations, real-world usage, and continuous delivery.

4. Setup Instructions

Prerequisites: Python

3.10 or later

pip and virtual environment tools

Docker (optional, for containerized deployment)

API keys for OpenAI/Watsonx, GitHub, and Jira integration

Internet access for cloud and API services Installation

Process:

Clone the project repository

Create and activate a virtual environment

Install dependencies from requirements.txt

Configure environment variables in a .env file (API keys, DB URL, etc.)

Run the backend server with FastAPI

Launch the dashboard frontend using Streamlit/React

Connect to external tools (GitHub, Jira, CI/CD pipeline)

Upload project data and interact with AI modules

5. Folder Structure

app/ – Main application folder.

requirements_manager.py – Module for parsing and refining software

requirements using LLM.

code_generator.py – Module for generating code templates and snippets from refined requirements. test_generator.py – Module for creating unit/integration tests automatically. review_assistant.py – Module for AI-assisted code review, bug detection, and optimization tips.

ci_cd_orchestrator.py – Handles CI/CD integration, build triggers, and deployment automation. utils.py – Helper functions (e.g., logging, formatting, API calls).

interface.py – Defines the Gradio/Web UI layout (tabs, inputs, outputs).

main.py – Entry point script to launch the Smart SDLC application.

requirements.txt – List of dependencies (transformers, torch, gradio, scikit-learn, docker-py, etc.).

README.md – Documentation about project purpose, setup, and usage. .env – (Optional) Configuration file for API keys, tokens, or environment variables.data/ – Folder for input artifacts (requirements docs, design specs,

sample code).

outputs/ – Folder for storing generated code, test cases, reviews, and deployment logs. models/ – Folder for storing fine-tuned or downloaded AI models. tests/ – Folder for test scripts to validate Smart SDLC modules.

6. Running the Application

Frontend(webdashboard)

To start the project: run the Python script (e.g. python main.py).

main.py will automatically launch a local server and provide both a local and optional public Gradio link.

Local: http://127.0.0.1:7860 (default)

Public access (optional): shareable Gradio link if share=True is enabled.

Backend (Model & Processing)

Model: uses a fine-tuned LLM (e.g. a Hugging Face / OpenAI model) via a model client loaded into memory.

Functions:

Handles prompt creation and text generation.

Ingests inputs (raw text, uploaded PDFs) and returns structured outputs (summaries, action items, code review suggestions).

Generates:

Issue summaries (text from uploaded artifacts)

Actionable tasks (from meeting notes, PRs)

Traceability reports (linking requirements → design → tests)

Policy & compliance checks (basic rule checks)

Risk & impact analysis (short bullet lists)

Processing details:

PDF/text reading via PyPDF2 or similar.

Tokenization and truncation safeguards to maintain safe max token lengths.

Response sampling & diversity controlled by temperature (default: 0.0–0.7) and top_p.

Long inputs are chunked and processed with overlap to preserve context.

Frontend (UI)

The frontend is implemented with Gradio, providing a clean tabbed interface:

Tabs: Overview, Issue Analyzer, Requirement Trace, Policy Checker, Upload.

Features:

Upload PDFs or paste text directly for immediate processing.

Real-time generated outputs shown in panels with copy/download options.

Export results as Markdown or CSV for integration into issue trackers.

Simple controls for temperature, max tokens, and model selection.

Frontend design goals:

Lightweight and responsive UI.

Clear, actionable outputs (task list, severity labels, recommended fixes).

No model inference interruption; UI shows progress and logs.

LLM Integration

Primary integration with selected LLM backend (local quantized or cloud API).

Optimized for GPU execution if available, otherwise falls back to CPU.

Provides context-aware responses: chain-of-thought is not exposed, only final outputs.

Supports streaming output for progressive feedback.

Deployment Layer

Handles deployment modes and accessibility:

Local deployment:

Launch: python main.py

Local URL: http://127.0.0.1:7860 (default)

Docker

Dockerfile included. Build and run for consistent runtime.

Example:

docker build -t smart-sdlc .

docker run --gpus all -p 7860:7860 smart-sdlc Cloud

/ Production:

Containerize and deploy to cloud provider (AWS ECS, GCP Cloud Run, Azure).

Use reverse proxy / authentication layer for public access.

Set environment variables for API keys, model endpoints, and resource limits.

Public access:

Gradio share=True generates a temporary public link (for demos).

For production use, add OAuth/SSO and HTTPS fronting.

Security & Policy

Input sanitization and size limits enforced to prevent abuse.

Audit logs for each inference: input hash, model used, timestamp, user id (if authenticated).

Policy checks: preset rules (e.g., license checks, PII redaction prompts).

Rate limiting and authentication recommended for public endpoints.

DevOps & CI/CD

Unit tests for parsing, chunking, and output formatting.

CI pipeline runs linting, unit tests, and builds container images.

Canary releases for model updates; model registry with version tags.

Monitoring: metrics for latency, error rates, and token usage.

Notes / Best practices

Keep model weights and API keys out of source control; use environment variables or secret manager.

For very long documents, use chunked summarization and final aggregation step to preserve fidelity.

Provide a “confidence” or “evidence” section in outputs so users can inspect source excerpts used by the model.

Provide an option to export findings into common issue trackers (JIRA/GitHub) via API.

7.API Documentation

Backend APIs available for Smart SDLC include:

POST /analyze-requirements → Accepts requirement documents and generates AI-driven requirement classification, ambiguity detection, and refinement suggestions.

POST /review-code → Accepts source code files and provides static analysis with AI-driven bug/issue detection.

POST /summarize-test-cases → Accepts uploaded test cases and returns a summarized coverage/traceability matrix.

POST /sdlc-task → Accepts a project stage/task and responds with AI-generated actionable steps (planning, design, testing, deployment). POST /upload-doc → Uploads project documents and stores embeddings for semantic search and traceability.

8.Authentication

Each endpoint is tested and documented in Swagger UI for quick inspection and trial during development.

This version of the project runs in an open environment for demonstration purposes.

However, secure deployments can integrate:

Token-based authentication (JWT or API Keys)

OAuth2 with enterprise credentials (e.g., Azure AD, IBM Cloud)

Role-based access (admin, developer, tester, project manager)

Planned enhancements include user sessions and activity history tracking.

9. User Interface

The user interface is built using Gradio, providing a clean and interactive web platform.

Organized into tabs for different SDLC stages:

Requirements Analyzer – Upload or paste requirements

Design Generator – Generates architecture/design suggestions from refined requirements.

Code Review Tab – Upload code files → AI generates bug reports, security flags, and improvement tips.

Test Case Summarizer – Upload test cases → AI creates coverage summaries and traceability reports.

Deployment Assistant – Provides checklists and automated documentation for deployment readiness.

Outputs are presented as structured summaries, tables, or downloadable reports.

Components like upload boxes, export buttons, and status logs are organized in rows for user-friendly layout.

10. Testing

Unit Testing: Conducted for all core functions, including prompt generation, PDF text extraction, and response formatting to ensure each module works correctly in isolation.

API Testing: Performed using Swagger UI, Postman, and custom test scripts to verify endpoints, request/response accuracy, and error handling.

Manual Testing: Executed on the Gradio interface to validate file uploads, keyword inputs, chat interactions, and output consistency.

Edge Case Handling: Tested scenarios with malformed inputs, large PDF files, empty text fields, and invalid API keys to ensure graceful failure and informative error messages.

Validation: Each module and function was confirmed to work reliably in both offline mode (local testing) and API-connected mode (real-time responses).

11.Screen shots

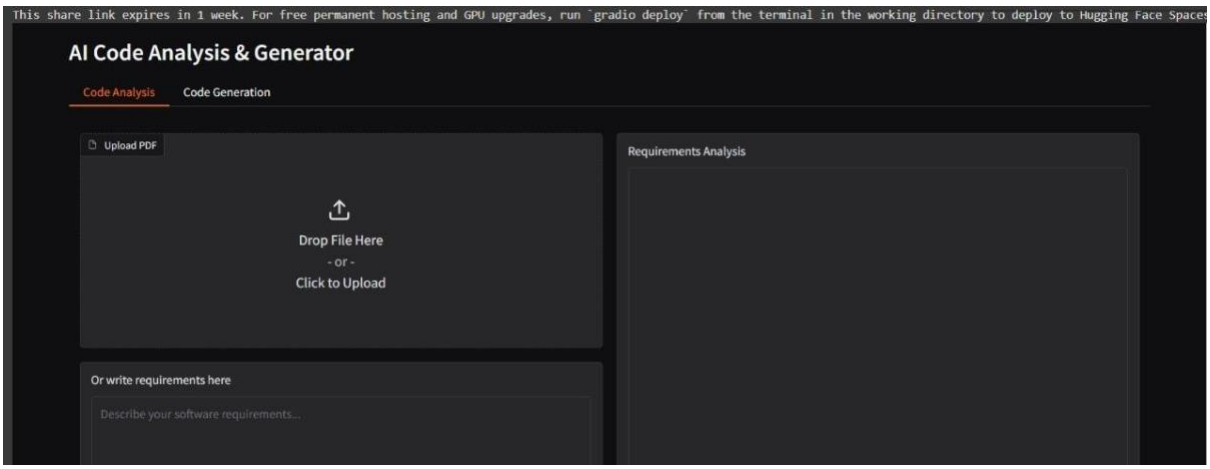

Figure 1

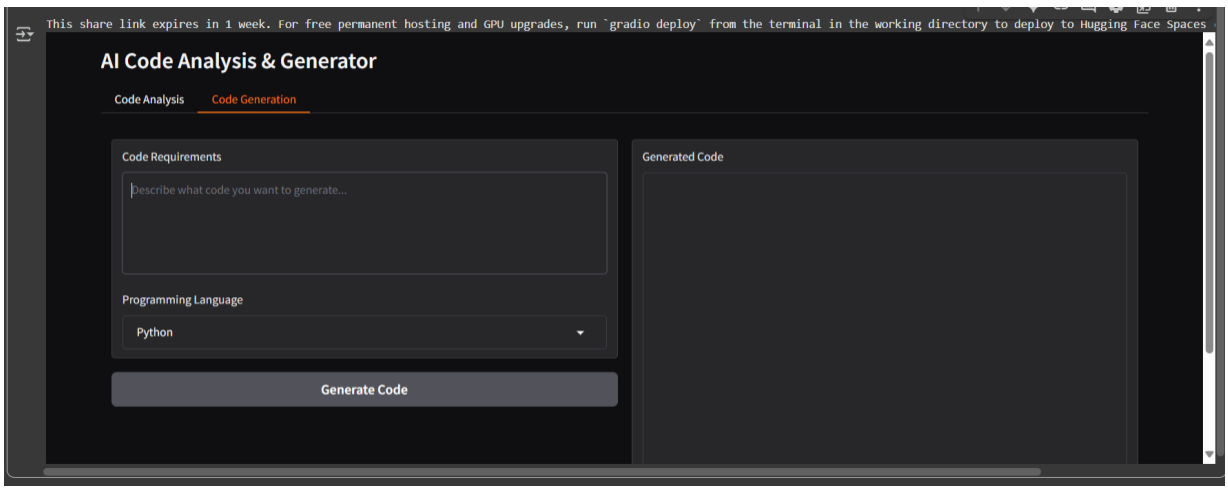

Figure 2

12. Known Issues

Dependency on Tools: Some SDLC stages rely heavily on third-party tools (e.g., testing, CI/CD, project management). Integration issues may occur if versions are incompatible.

User Training Requirement: New users may require additional training to understand the automated workflows of Smart SDLC.

Limited Customization: Certain project methodologies (e.g., Agile, Waterfall, Hybrid) may not be fully customizable in the initial version.

Scalability Constraints: High workload or multiple parallel projects may affect response time in resource-intensive stages like testing or deployment.

Environment Compatibility: The platform may not perform consistently across all operating systems or browsers without fine-tuning.

Error Handling: Unexpected inputs during requirement or design stages may produce generic error messages instead of detailed diagnostics.

Resource Usage: Continuous monitoring and automation may lead to higher CPU/memory usage if optimization is not implemented.

13. Future Enhancements

AI-Driven Insights: Integration of AI/ML for predictive analytics, risk detection, and automated effort estimation across different SDLC phases.

Advanced Automation: Expanding automation to cover end-to-end processes such as requirement gathering, test case generation, and deployment pipelines.

Enhanced Collaboration: Real-time dashboards and multi-user support for better collaboration among developers, testers, and project managers.

Improved Customization: Providing flexible configuration for Agile, Waterfall, DevOps, or Hybrid methodologies tailored to user preferences.

Cloud Deployment: Cloud-based hosting for scalability, remote access, and seamless integration with other enterprise platforms.

Security & Compliance: Implementation of robust authentication, authorization, and compliance checks (e.g., GDPR, ISO standards).

User Experience (UX): Building more intuitive, mobile-friendly, and accessible interfaces for broader adoption.

Continuous Integration: Adding support for popular CI/CD tools like Jenkins, GitHub Actions, or GitLab CI to streamline delivery pipelines.