**Project Title**

**Project Documentation**

# 1.Introduction

* Project title : **Smart sdlc Ai enhanced software development lifecycle**
* Team leader. :J.Anushiya
* Team member : k.Oviya
* Team member : K.Anuthirya
* Team member :M.Lavanya

# 2.project overview

* Purpose : The purpose of a Smart SDLC is to improve the traditional software development lifecycle by integrating Artificial Intelligence (AI) techniques and automation. This makes the entire process more efficient, adaptive, and intelligent.
* 🔹 Key Purposes:
* 1. Automation of Repetitive Tasks – Reduce manual effort in coding, testing, debugging, and documentation.
* 2. Predictive Analytics – Use AI to predict risks, project delays, cost overruns, and resource requirements.
* 3. Quality Enhancement – Improve software reliability by detecting defects early using AI-powered testing tools.
* 4. Smart Decision-Making – Support project managers and developers with AI-driven recommendations during requirement analysis, design, and deployment.
* 5. Continuous Improvement – Enable adaptive learning from past projects to refine and optimize future SDLC cycles.
* 6. Faster Delivery – Shorten development time with AI-enabled code generation, testing, and deployment pipelines.
* 7. Enhanced Collaboration – Provide intelligent insights that help developers, testers, and stakeholders work more effectively together.
* Features:
* 1. AI-Powered Requirement Analytics
* Natural Language Processing (NLP) to understand client requirements.
* Detect ambiguities, conflicts, or missing details in specifications.
* 2. Automated Design Assistanc
* AI tools generate UML diagrams, wireframes, and architecture suggestions.
* Intelligent recommendation of design patterns
* 3. Smart Code Generation
* AI-assisted code completion and auto-generation.
* Reusable code suggestions to save development time.
* 4. Intelligent TestinG
* Automated test case generation
* AI detects bugs and vulnerabilities early
* Self-healing test scripts that adapt to changes in code
* 5. Predictive Project Management
* AI predicts risks, deadlines, and resource needs.
* Smart dashboards for tracking progress and quality.

# 3. Architecture

1. Requirement Layer

AI-powered requirement analysis (NLP, ambiguity detection).

2. Design Layer

AI-assisted modeling, architecture suggestions.

3. Development Layer

Smart code generation, auto-completion, error prediction.

4. Testing Layer

Automated test case generation, bug detection, self-healing tests.

5. Deployment Layer

AI-optimized CI/CD pipelines, intelligent rollback.

**4. Setup Instructure**

**1. Input Stage**

**Collect client requirements.**

**Feed into AI tools (NLP for analysis).**

**2. Processing Stage**

**Requirement Analysis → AI validates & removes ambiguities.**

**Design → AI suggests patterns & models.**

**Development → AI-assisted coding & automation.**

**Testing → Automated test cases & bug detection.**

**Deployment → AI-optimized CI/CD pipelines.**

**3. Output Stage**

**Software product delivered with AI-enhanced quality.**

**4. Support & Feedback Stage**

**Predictive maintenance.**

**Continuous AI learning to improve next cycle.**

## 5. Folder Structure

Smart-SDLC-AI/

│── 📂 docs/ # Project documentation

│ ├── requirements.md # Requirement analysis

│ ├── design.md # System design docs

│ └── user\_manual.md # End-user documentation

│

│── 📂 data/ # Dataset or input files for AI

│ ├── raw/ # Raw data

│ ├── processed/ # Cleaned/processed data

│ └── models/ # Trained AI/ML models

│

│── 📂 src/ # Source code

│ ├── requirements/ # AI requirement analysis

│ ├── design/ # AI-driven design tools

│ ├── development/ # Smart code generation

│ ├── testing/ # Automated test scripts

│ ├── deployment/ # CI/CD and deployment scripts

│ └── maintenance/ # Monitoring, logs, predictive maintenance

│

│── 📂 tests/ # Unit & integration tests

│

│── 📂 configs/ # Configuration files (YAML, JSON)

│

│── 📂 scripts/ # Utility scripts (setup, automation)

│

│── 📂 reports/ # Generated reports (AI analysis, test results)

│

│── README.md # Project overview

│── requirements.txt # Python/AI dependencies

│── .gitignore # Ignore unnecessary files

## 6. Running the Application

1. Clone or Download the Project

git clone https://github.com/your-username/Smart-SDLC-AI.git

cd Smart-SDLC-AI

2. Set Up Virtual Environment (Python Example)

python -m venv venv

source venv/bin/activate # For Linux/Mac

venv\Scripts\activate # For Windows

3. Install Dependencies

pip install -r requirements.txt

4. Configure the Application

Edit settings in configs/ (e.g., config.yaml or config.json).

Provide dataset/model paths in data/ if required.

5. Run the Application

python src/main.py

(Assuming main.py is the entry point that integrates all SDLC stages)

6. Access the Application

If it’s a web app → Open browser at http://localhost:5000/ or http://127.0.0.1:8000/.

If it’s a CLI tool → Output will be shown in the terminal.

7. Run Tests (Optional)

pytest tests/

8. Deployment (Optional)

Use Docker for containerized deployment:

docker build -t smart-sdlc-ai .

docker run -p 8000:8000 smart-sdlc-ai

## 7. API Documentation

🔹 Base URL

http://localhost:8000/api/v1/

---

🔹 Endpoints

1. Requirement Analysis

POST /requirements/analyze

Description: Analyze project requirements using AI (NLP).

Request Body (JSON):

{

"requirement\_text": "The system should allow users to register and login securely."

}

Response (JSON):

{

"status": "success",

"ambiguities\_found": ["What type of authentication is required?"],

"recommendations": ["Use multi-factor authentication."]

}

---

2. Design Suggestions

POST /design/generate

Description: Get AI-based system design suggestions.

Request Body (JSON):

{

"module": "User Authentication"

}

Response (JSON):

{

"design\_patterns": ["MVC", "Singleton"],

"uml\_diagram": "url-to-generated-diagram.png"

}

---

3. Code Generation

POST /development/generate-code

Description: Generate sample code snippets.

Request Body (JSON):

{

"language": "python",

"feature": "user login with JWT authentication"

}

Response (JSON):

{

"generated\_code": "def login(user, password): ..."

}

---

4. Automated Testing

POST /testing/run

Description: Run AI-generated test cases.

Request Body (JSON):

{

"module": "authentication"

}

Response (JSON):

{

"tests\_run": 10,

"passed": 9,

"failed": 1,

"bug\_report": "Password reset link not working."

}

---

5. Deployment

POST /deployment/start

Description: Deploy application using AI-optimized pipeline.

Request Body (JSON):

{

"environment": "staging"

}

Response (JSON):

{

"status": "deployed",

"url": "http://staging.smart-sdlc-ai.com"

}

---

6. Maintenance & Monitoring

GET /maintenance/monitor

Description: Check predictive maintenance status.

Response (JSON):

{

"uptime": "99.9%",

"predicted\_issues": ["Database load may exceed limit in 3 days"],

"recommendations": ["Scale DB cluster"]

}

**8. Authentication**

1. Basic Authentication (Not Recommended for Production)

Username + Password sent with every request.

Easy to implement but less secure.

2. Token-Based Authentication (Preferred)

User logs in → Server issues a token (JWT).

Token is sent in headers with every request.

Example:

Authorization: Bearer <your-jwt-token>

3. OAuth2 (For Enterprise/Cloud Apps)

Secure delegation (Google, GitHub login, etc.).

Recommended if third-party integration is needed.

🔹 API Endpoints for Authentication

1. Register User

POST /auth/register

{

“username”: “developer01”,

“password”: “securePass@123”,

“role”: “developer”

}

Response

{

“status”: “success”,

“message”: “User registered successfully”

}

1. Login User

POST /auth/login

{

“username”: “developer01”,

“password”: “securePass@123”

}

Response

{

“access\_token”: “eyJhbGciOiJIUzI1NiIsInR5cCI6…”,

“expires\_in”: 3600

}

1. Access Protected Route

GET /requirements/analyze

Header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6…

Response

{

“status”: “success”,

“analysis”: “No ambiguities found”

}

🔹 Sample JWT Authentication in Python (FastAPI)

From fastapi import FastAPI, Depends, HTTPException

From fastapi.security import OAuth2PasswordBearer

Import jwt

App = FastAPI()

Oauth2\_scheme = OAuth2PasswordBearer(tokenUrl=”auth/login”)

SECRET\_KEY = “your-secret-key”

Def verify\_token(token: str = Depends(oauth2\_scheme)):

Try:

Payload = jwt.decode(token, SECRET\_KEY, algorithms=[“HS256”])

Return payload

Except:

Raise HTTPException(status\_code=401, detail=”Invalid Token”)

@app.get(“/requirements/analyze”)

Def analyze\_requirements(token: dict = Depends(verify\_token)):

Return {“status”: “success”, “message”: “Token Verified, AI Analysis Done”}

## 9. User Interface

1. Login / Authentication Page

Username & Password (JWT-based login).

Role-based access (Admin, Developer, Tester, Project Manager).

2. Dashboard (Home Screen)

Overview of all project modules.

Cards/tiles for each SDLC stage:

📌 Requirements

📌 Design

📌 Development

📌 Testing

📌 Deployment

📌 Maintenance

AI Notifications (risk alerts, recommendations).

1. Requirement Analysis Screen

Text area to enter requirements.

AI highlights ambiguities & recommendations.

Export option (PDF/Doc).

1. Design Screen

Upload requirements → AI generates UML diagrams, design suggestions.

Display architecture diagrams in canvas.

Option to edit/approve AI suggestions.

1. Development Screen

AI-assisted code generation panel.

Drop-down to select language (Python, Java, JS, etc.).

Auto-complete suggestions like GitHub Copilot.

1. Testing Screen

Run AI-generated test cases.

View test results (Passed ✅ / Failed ❌).

Bug reports auto-generated.

1. Deployment Screen

Start AI-optimized CI/CD pipeline.

Show deployment logs & status.

Rollback button if deployment fails.

1. Maintenance & Monitoring Screen

System health (Uptime, Logs, Predictive Failures).

AI Recommendations for scaling/bug fixing.

Chatbot support for quick issue resolution.

1. Reports & Analytics

Graphs for project progress.

Risk predictions.

Quality score (AI-evaluated).

## 10. Testing

1. Types of Testing

✅ Unit Testing – Test individual modules/functions.

✅ Integration Testing – Check how modules interact.

✅ System Testing – Validate complete system behavior.

✅ Regression Testing – Ensure updates don’t break old features.

✅ Performance Testing – Check speed, scalability, and load.

✅ Security Testing – Detect vulnerabilities with AI.

---

2. AI Features in Testing

1. Automated Test Case Generation

AI reads requirements → auto-creates test cases.

2. Self-Healing Test Scripts

Tests adapt automatically when code changes.

3. Bug Prediction & Detection

AI predicts where bugs are most likely.

Finds defects earlier than manual testing.

4. Smart Test Coverage

AI suggests missing test cases to improve coverage.

5. Continuous Testing in CI/CD

Tests run automatically before deployment.

---

3. Sample Testing API Endpoint

POST /testing/run

{

"module": "authentication"

}

Response

{

"tests\_run": 15,

"passed": 14,

"failed": 1,

"bug\_report": "Password reset link not working"

}

---

4. Sample Automated Test (Python – Pytest)

import pytest

from app.auth import login

def test\_valid\_login():

assert login("user1", "password123") == "Login Successful"

def test\_invalid\_login():

assert login("user1", "wrongpass") == "Invalid Credentials"

Run tests:

pytest tests/

**11.screen shots**

**To add**

1. **Known Issues**
2. **Future enhancement**