Project architecture for smartSDLC system:

import os

import json

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

import nltk

from nltk.tokenize import word\_tokenize

class SmartSDLC:

def \_\_init\_\_(self, project\_requirements):

self.project\_requirements = project\_requirements

def requirements\_gathering(self):

# AI-powered requirements analysis

tokens = word\_tokenize(self.project\_requirements)

return tokens

def design(self, requirements):

# AI-assisted design generation

architecture\_diagram = {

"components": [],

"relationships": []

}

for requirement in requirements:

if requirement.lower() in ["user", "authentication"]:

architecture\_diagram["components"].append("Authentication Service")

architecture\_diagram["relationships"].append("User -> Authentication Service")

return architecture\_diagram

def implementation(self, design):

# AI-powered code generation

code = {}

for component in design["components"]:

if component == "Authentication Service":

code[component] = """

def authenticate\_user(username, password):

# Authenticate user logic

pass

"""

return code

def testing(self, code):

# AI-driven test generation

tests = {}

for component, code in code.items():

if component == "Authentication Service":

tests[component] = """

def test\_authenticate\_user():

# Test authenticate user logic

pass

"""

return tests

def deployment(self, code):

# AI-powered deployment

print("Deploying code to production environment...")

# Simulate deployment

print("Code deployed successfully!")

def run\_pipeline(self):

requirements = self.requirements\_gathering()

design = self.design(requirements)

code = self.implementation(design)

tests = self.testing(code)

self.deployment(code)

Model selection and archirecture:

Milestone-1:model selection and architecture

1.1:Research and select the apporiate generative AI Model

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, LeakyReLU, Reshape, Flatten

# Define the generator model

def build\_generator():

model = Sequential([

Dense(128, input\_dim=100),

LeakyReLU(alpha=0.2),

Dense(256),

LeakyReLU(alpha=0.2),

Dense(512),

LeakyReLU(alpha=0.2),

Dense(28 \* 28, activation='tanh'),

Reshape((28, 28))

])

return model

# Define the discriminator model

def build\_discriminator():

model = Sequential([

Flatten(input\_shape=(28, 28)),

Dense(512),

LeakyReLU(alpha=0.2),

Dense(256),

LeakyReLU(alpha=0.2),

Dense(1, activation='sigmoid')

])

return model

1.2:Define the architecture of the application.

import os

import json

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

import nltk

from nltk.tokenize import word\_tokenize

class RequirementsGatheringModule:

def \_\_init\_\_(self, project\_requirements):

self.project\_requirements = project\_requirements

def analyze\_requirements(self):

# Tokenize requirements

tokens =word\_tokenize(self.project\_requirements)

return tokens

class DesignModule:

def \_\_init\_\_(self, requirements):

self.requirements = requirements

def generate\_design(self):

# Generate architecture diagram

architecture\_diagram = {

"components": [],

"relationships": []

}

for requirement in self.requirements:

if requirement.lower() in ["user", "authentication"]:

architecture\_diagram["components"].append("Authentication Service")

architecture\_diagram["relationships"].append("User -> Authentication Service")

return architecture\_diagram

class ImplementationModule:

def \_\_init\_\_(self, design):

self.design = design

def generate\_code(self):

# Generate code for each component

code = {}

for component in self.design["components"]:

if component == "Authentication Service":

code[component] = """

def authenticate\_user(username, password):

# Authenticate user logic

pass

"""

return code

class TestingModule:

def \_\_init\_\_(self, code):

self.code = code

def generate\_tests(self):

# Generate tests for each component

tests = {}

for component, code in self.code.items():

if component == "Authentication Service":

tests[component] = """

def test\_authenticate\_user():

# Test authenticate user logic

pass

"""

return tests

class DeploymentModule:

def \_\_init\_\_(self, code):

self.code = code

def deploy(self):

# Deploy code to production environment

print("Deploying code to production environment...")

# Simulate deployment

print("Code deployed successfully!")

class SmartSDLC:

def \_\_init\_\_(self, project\_requirements):

self.project\_requirements = project\_requirements

def run\_pipeline(self):

requirements\_gathering\_module = RequirementsGatheringModule(self.project\_requirements)

requirements = requirements\_gathering\_module.analyze\_requirements()

design\_module = DesignModule(requirements)

design = design\_module.generate\_design()

implementation\_module = ImplementationModule(design)

code = implementation\_module.generate\_code()

testing\_module = TestingModule(code)

tests = testing\_module.generate\_tests()

deployment\_module = DeploymentModule(code)

deployment\_module.deploy()

return {

"requirements": requirements,

"design": design,

"code": code,

"tests": tests

}

# Example usage

project\_requirements = """

The system should provide user authentication and data storage.

"""

smart\_sdlc = SmartSDLC(project\_requirements)

output = smart\_sdlc.run\_pipeline()

print(json.dumps(output, indent=4))

1.3:set up the development environment.

import os

import nltk

from nltk.tokenize import word\_tokenize

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

# Download NLTK data

nltk.download('punkt')

# Check if TensorFlow is installed correctly

print(tf.\_\_version\_\_)

# Check if Scikit-learn is installed correctly

from sklearn.ensemble import RandomForestClassifier

print(RandomForestClassifier().get\_params())

# Set up the project directory

project\_dir = os.getcwd()

print(project\_dir)

# Create a new directory for the project if it doesn't exist

if not os.path.exists(project\_dir):

os.makedirs(project\_dir)

# Create subdirectories for requirements, design, implementation, testing, and deployment

subdirectories = ['requirements', 'design', 'implementation', 'testing', 'deployment']

for subdirectory in subdirectories:

subdirectory\_path = os.path.join(project\_dir, subdirectory)

if not os.path.exists(subdirectory\_path):

os.makedirs(subdirectory\_path)

print("Development environment set up successfully!")

Verifying the Environment

To verify that the environment is set up correctly, you can run the following code:

import nltk

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

print(nltk.\_\_version\_\_)

print(tf.\_\_version\_\_)

print(train\_test\_split.\_\_module\_\_)

Core functionalities:

2.1:Develop the core functionalities:

import os

import json

from nltk.tokenize import word\_tokenize

from sklearn.model\_selection import train\_test\_split

import tensorflow as tf

class SmartSDLC:

def \_\_init\_\_(self, project\_requirements):

self.project\_requirements = project\_requirements

def requirements\_gathering(self):

# Tokenize requirements

tokens = word\_tokenize(self.project\_requirements)

return tokens

def design(self, requirements):

# Generate architecture diagram

architecture\_diagram = {

"components": [],

"relationships": []

}

for requirement in requirements:

if requirement.lower() in ["user", "authentication"]:

architecture\_diagram["components"].append("Authentication Service")

architecture\_diagram["relationships"].append("User -> Authentication Service")

returnarchitecture\_diagram

def implementation(self, design):

# Generate code for each component

code = {}

for component in design["components"]:

if component == "Authentication Service":

code[component] = """

def authenticate\_user(username, password):

# Authenticate user logic

pass

"""

return code

def testing(self, code):

# Generate tests for each component

tests = {}

for component, code in code.items():

if component == "Authentication Service":

tests[component] = """

def test\_authenticate\_user():

# Test authenticate user logic

pass

"""

return tests

def deployment(self, code):

# Deploy code to production environment

print("Deploying code to production environment...")

# Simulate deployment

print("Code deployed successfully!")

def run\_pipeline(self):

requirements = self.requirements\_gathering()

design = self.design(requirements)

code = self.implementation(design)

tests = self.testing(code)

self.deployment(code)

return {

"requirements": requirements,

"design": design,

"code": code,

"tests": tests

}

Example usage

project\_requirements = """

The system should provide user authentication and data storage.

"""

smart\_sdlc = SmartSDLC(project\_requirements)

output = smart\_sdlc.run\_pipeline()

print(json.dumps(output, indent=4))

2.2:Implement the fastAPI backend to manage routring and user input processing ,ensuring smooth api interactions.

\*smart\_sdlc\_api\*\*

\*\*app\*\*

\*\*main.py\*\*

\*\*routes\*\*

\*\*\_\_init\_\_.py\*\*

\*\*requirements.py\*\*

\*\*design.py\*\*

\*\*implementation.py\*\*

\*\*testing.py\*\*

\*\*deployment.py\*\*

\*\*models\*\*

\*\*\_\_init\_\_.py\*\*

\*\*requirement.py\*\*

\*\*design.py\*\*

\*\*implementation.py\*\*

\*\*test.py\*\*

\*\*database\*\*

\*\*\_\_init\_\_.py\*\*

\*\*config.py\*\*

\*\*requirements.txt\*\*

\*\*venv\*\*

# app/main.py

from fastapi import FastAPI

from app.routes.requirements import requirements\_router

from app.routes.design import design\_router

from app.routes.implementation import implementation\_router

from app.routes.testing import testing\_router

from app.routes.deployment import deployment\_router

app = FastAPI()

app.include\_router(requirements\_router, prefix="/requirements")

app.include\_router(design\_router, prefix="/design")

app.include\_router(implementation\_router, prefix="/implementation")

app.include\_router(testing\_router, prefix="/testing")

app.include\_router(deployment\_router, prefix="/deployment")

# app/routes/requirements.py

from fastapi import APIRouter, Request

from pydantic import BaseModel

requirements\_router = APIRouter()

class Requirement(BaseModel):

description: str

@requirements\_router.post("/gather")

async def gather\_requirement(requirement: Requirement):

# Process the requirement

return {"message": "Requirement gathered successfully"}

Main:

# main.py

from fastapi import FastAPI

from app.routes.requirements import requirements\_router

from app.routes.design import design\_router

from app.routes.implementation import implementation\_router

from app.routes.testing import testing\_router

from app.routes.deployment import deployment\_router

from app.services.smart\_sdlc import SmartSDLC

app = FastAPI()

app.include\_router(requirements\_router, prefix="/requirements")

app.include\_router(design\_router, prefix="/design")

app.include\_router(implementation\_router, prefix="/implementation")

app.include\_router(testing\_router, prefix="/testing")

app.include\_router(deployment\_router, prefix="/deployment")

smart\_sdlc = SmartSDLC()

@app.get("/")

async def root():

return {"message": "Welcome to SmartSDLC"}

@app.post("/run\_pipeline")

async def run\_pipeline(project\_requirements: str):

output = smart\_sdlc.run\_pipeline(project\_requirements)

return output

3.1: wring the main application logic in main.py using python code for smartsdlc.

main.py

from fastapi import FastAPI

from pydantic import BaseModel

import nltk

from nltk.tokenize import word\_tokenize

app = FastAPI()

class ProjectRequirements(BaseModel):

requirements: str

class SmartSDLC:

def \_\_init\_\_(self):

pass

def run\_pipeline(self, project\_requirements: str):

# Tokenize requirements

tokens = word\_tokenize(project\_requirements)

# Generate design

design = self.generate\_design(tokens)

# Generate code

code = self.generate\_code(design)

# Generate tests

tests = self.generate\_tests(code)

# Deploy code

self.deploy\_code(code)

return {

"requirements": tokens,

"design": design,

"code": code,

"tests": tests

}

def generate\_design(self, tokens):

# Generate architecture diagram

architecture\_diagram = {

"components": [],

"relationships": []

}

for token in tokens:

if token.lower() in ["user", "authentication"]:

architecture\_diagram["components"].append("Authentication Service")

def generate\_code(self, design):

# Generate code for each component

code = {}

for component in design["components"]:

if component == "Authentication Service":

code[component] = """

def authenticate\_user(username, password):

# Authenticate user logic

pass

"""

return code

def generate\_tests(self, code):

# Generate tests for each component

tests = {}

for component, code in code.items():

if component == "Authentication Service":

tests[component] = """

def test\_authenticate\_user():

# Test authenticate user logic

pass

"""

return tests

def deploy\_code(self, code):

# Deploy code to production environment

print("Deploying code to production environment...")

# Simulate deployment

print("Code deployed successfully!")

@app.post("/run\_pipeline")

async def run\_pipeline(project\_requirements: ProjectRequirements):

smart\_sdlc = SmartSDLC()

output = smart\_sdlc.run\_pipeline(project\_requirements.requirements)

return output

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)

Frontend Development:

4.1:Designing and Deveioping the user Interface

# app.py

import streamlit as st

from smart\_sdlc import SmartSDLC

st.title("SmartSDLC")

# Input field for project requirements

project\_requirements = st.text\_area("Enter project requirements")

# Button to run the pipeline

if st.button("Run Pipeline"):

smart\_sdlc = SmartSDLC()

output = smart\_sdlc.run\_pipeline(project\_requirements)

# Display the output

st.write("Requirements:")

st.write(output["requirements"])

st.write("Design:")

st.write(output["design"])

st.write("Code:")

st.write(output["code"])

st.write("Tests:")

st.write(output["tests"])

# smart\_sdl()

class SmartSDLC:

# ... (same implementation as before)

4.2:Creating dynamic interaction with Backend reaction

# app.py

import streamlit as st

from smart\_sdlc import SmartSDLC

st.title("SmartSDLC")

# Input field for project requirements

project\_requirements = st.text\_area("Enter project requirements")

# Button to run the pipeline

if st.button("Run Pipeline"):

smart\_sdlc = SmartSDLC()

output = smart\_sdlc.run\_pipeline(project\_requirements)

# Display the output

st.write("Requirements:")

st.write(output["requirements"])

st.write("Design:")

st.write(output["design"])

st.write("Code:")

st.write(output["code"])

st.write("Tests:")

st.write(output["tests"])

# Dynamic interaction: allow user to modify code

code = output["code"]

modified\_code = st.text\_area("Modify Code", code)

if st.button("Save Changes"):

# Save the modified code

output["code"] = modified\_code

st.write("Changes saved!")

# Dynamic interaction: allow user to run tests

if st.button("Run Tests"):

# Run the tests

test\_results = smart\_sdlc.run\_tests(output["tests"])

st.write("Test Results:")

st.write(test\_results)

Deployment:

5.1:Preparing the application for local development

Step 1: Create a Dockerfile

Create a Dockerfile to containerize your application:

# Dockerfile

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["streamlit", "run", "app.py"]

Step 2: Create a file for dependencies

Create a requirements.txt file to list the dependencies:

streamlit

nltk

Step 3: Build the Docker Image

Build the Docker image:

bash

docker build -t smart\_sdlc .

Step 4: Run the Docker Container

Run the Docker container:

bash

docker run -p 8501:8501 smart\_sdlc

5.2:Testing and verifying local deployment

import unittest

from smart\_sdlc import SmartSDLC

class TestSmartSDLC(unittest.TestCase):

def test\_run\_pipeline(self):

smart\_sdlc = SmartSDLC()

project\_requirements = "This is a test project"

output = smart\_sdlc.run\_pipeline(project\_requirements)

self.assertIsNotNone(output)

self.assertIn("requirements", output)

self.assertIn("design", output)

self.assertIn("code", output)

self.assertIn("tests", output)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

Conclusion:

import streamlit as st

from smart\_sdlc import SmartSDLC

st.title("SmartSDLC")

project\_requirements = st.text\_area("Enter project requirements")

if st.button("Run Pipeline"):

smart\_sdlc = SmartSDLC()

output = smart\_sdlc.run\_pipeline(project\_requirements)

st.write("Requirements:")

st.write(output["requirements"])

st.write("Design:")

st.write(output["design"])

st.write("Code:")

st.write(output["code"])

st.write("Tests:")

st.write(output["tests"])