Smart SDLC-AI-Enhanced Software Development Life Cyle

Project Documentation

- **Introduction**
  - Project title :SmartSDLC – AI-Enhanced Software Development Lifecycle

    Team member : RAGADEVAN .M
    Team member : MUKILAN .M
    Team member : LOKESH .R
  - Team member : VENKATESH .G

Project overview

Purposes of SmartSDLC

Introduction

Software Development Lifecycle (SDLC) defines the process followed to develop, maintain, and improve software systems. With the rise of Artificial Intelligence (AI), traditional SDLC is evolving into SmartSDLC, where AI tools enhance every phase of development — from planning to deployment and maintenance. SmartSDLC aims to improve efficiency, reduce errors, accelerate decision-making, and provide deeper insights throughout the development process.

✓Architecture

The SmartSDLC architecture integrates AI technologies into the traditional software development lifecycle. It consists of the following key layers:

1. Data Layer

- Collects data from users, systems, and past projects.
- Used for analysis, predictions, and decision-making.

2. AI Engine Layer

- Machine learning models, NLP tools, and analytics algorithms process data.
- Provides suggestions, automation, and insights.

3. Application Layer

- Development tools, testing frameworks, deployment pipelines, and monitoring systems.
- Works with AI-generated outputs to enhance workflows.

4. User Interface Layer

- Interfaces for developers, testers, and managers to interact with AI tools.
- Provides dashboards, reports, and feedback mechanisms.

✅ Pre-Requisites for SmartSDLC

Before implementing SmartSDLC, the following are required:

1. Technical Skills

- Understanding of traditional SDLC processes.
- Knowledge of programming languages (e.g., Python, Java).
- Familiarity with AI/ML concepts and tools.

2. Tools and Technologies

- AI tools like machine learning frameworks (TensorFlow, PyTorch).
- Code assistance tools (GitHub Copilot, Tabnine).
- Automated testing tools (Testim, Selenium).
- Cloud platforms (AWS, Azure, Google Cloud).
- CI/CD pipelines (Jenkins, GitLab).

3. Data Availability

- Access to historical project data.
- User feedback, logs, and performance metrics for training AI models.

4. Infrastructure

- High-performance computing or cloud resources.
- Secure data storage and handling mechanisms.

5. Team Readiness

- Skilled developers, testers, and project managers.
- Awareness of AI ethics and privacy considerations.
- Collaboration mindset for integrating AI into workflows.

6. Budget and Planning

- Investment for tools, training, and infrastructure.
- Roadmap for phased implementation and scaling.

✅ Milestone 1 – Model Selection and Architecture

In this milestone, the focus is on choosing the right AI models and defining the system architecture to support SmartSDLC.

Objectives

1. Identify the appropriate AI models for different phases (requirement analysis, code generation, testing, etc.).

2. Design the architecture that integrates AI tools seamlessly with the software development process.

Steps Involved

1. Model Selection
    - Requirement Analysis → Use NLP models like GPT, BERT for text understanding.
    - Code Assistance → Use models like Codex, Tabnine for auto-completion and code generation.
    - Testing & Debugging → Use anomaly detection models, reinforcement learning for smarter test case generation.
    - Performance Monitoring → Use predictive models to forecast system failures or slowdowns.

2. Architecture Design
    - Create layers such as:
    - Data Layer → Input from users, logs, datasets.
    - AI Processing Layer → Models, analytics engines.
    - Application Layer → Development tools, deployment pipelines.
    - Interface Layer → Dashboards, alerts, reports.
    - Ensure scalability and security in the architecture.

Deliverables

- ✔ Selected AI models with justification.
- ✔ System architecture diagram.
- ✔ List of tools and technologies to be integrated.
- ✔ Plan for data collection, storage, and model training.

✓ Activity 1.1 – Research and Select the Appropriate Generative AI Model (Brief)

In this activity, the goal is to explore available AI models and select the ones best suited for the needs of SmartSDLC.

Steps to Follow

1. Understand Requirements

- Identify which phases need generative AI support (e.g., requirement analysis, code generation, documentation).
- Determine the type of output required (text, code snippets, test cases).

2. Research Available AI Models

- GPT (Generative Pre-trained Transformer)
  - ➢ Great for text generation, requirement summarization, and conversation-based interactions.
- Codex (by OpenAI)
  - ➢ Designed for code generation and completion tasks.
- BERT (Bidirectional Encoder Representations from Transformers)
  - ➢ Best for understanding context in text for requirement analysis.

- T5 (Text-to-Text Transfer Transformer)
  - ➢ Converts one text format to another, useful for documentation and paraphrasing.
- Stable Diffusion / DALL·E (if UI mockups or diagrams are needed)
  - ➢ For generating visuals or illustrations.

3. Evaluate Based on

- Accuracy and relevance to tasks.
- Integration capabilities with development tools.
- Scalability and computational requirements.
- Open-source vs paid models.
- Ethical considerations (bias, privacy).

4. Select the Model(s)

Example:

- Requirement analysis → GPT-4 or BERT
- Code generation → Codex or Tabnine
- Testing → Reinforcement learning-based anomaly models

Deliverables

- ✔ A list of AI models researched.
- ✔ Justification for selecting specific models based on project needs.
- ✔ Documentation of features, limitations, and integration methods.

✅ Activity 1.2 – Define the Architecture of the Application

In this activity, you design the overall structure of the SmartSDLC application to show how AI models and other components work together.

Steps to Define the Architecture

1. Identify Key Components

User Interface Layer

- ➤ Dashboard for developers, testers, and managers.
- ➤ Input forms for requirements, feedback, and settings.

Application Layer

- ➤ Development tools (IDEs, code editors).
- ➤ Testing frameworks (automated test runners).
- ➤ Deployment pipelines (CI/CD).

AI Processing Layer

- ➤ Generative AI models (e.g., GPT, Codex).
- ➤ Analytics engines for performance monitoring.
- ➤ NLP tools for requirement analysis.

Data Layer

- ➤ User data, historical logs, test results.
- ➤ Cloud storage or on-premises servers.
- ➤ Secure databases.

2. Define Data Flow

- Data from users → Processed by AI models → Results shown on UI.

- Feedback loop for continuous learning.
- Automated triggers for testing and deployment.

3. Ensure Scalability and Security
   - Use cloud services for scalability.
   - Implement authentication and encryption for data privacy.

4. Create Architecture Diagram
   - Show how layers are connected.
   - Highlight where AI models interact with other components.

Sample Architecture Diagram Description

Input → Requirement gathering from users → Sent to AI Engine for analysis → Generates insights and recommendations → Displayed in UI Dashboard → Developers use outputs in Code Editors → Automated testing tools monitor results → Data stored in Databases → Insights fed back to improve AI models.

Deliverables

- ✔ Architecture diagram.
- ✔ Description of each layer's functionality.
- ✔ Data flow explanation.
- ✔ Security and scalability considerations.

✅ Activity 1.3 – Set Up the Development Environment

In this activity, you prepare all the tools, platforms, and configurations needed to build and run the SmartSDLC application.

Steps to Set Up the Development Environment

1. Select Development Tools

Code Editor / IDE

➢ Visual Studio Code, PyCharm, or IntelliJ.

Version Control

➢ Git and GitHub for code management and collaboration.

Containerization (Optional)

➢ Docker for environment consistency.

2. Install AI Libraries and Frameworks

Python libraries:

➢ TensorFlow, PyTorch, transformers, scikit-learn.

Code generation tools:

➢ OpenAI API, Hugging Face models.

NLP tools:

➢ SpaCy, NLTK.

3. Set Up Databases

Choose between:

➢ PostgreSQL, MongoDB, or Firebase.

➢Configure secure access and backups.

4. Configure Cloud / Server Access

- Set up AWS, Google Cloud, or Azure account.
- Create virtual machines or container services if required.
- Set up API keys and authentication protocols.

5. Integrate Testing Tools

- Install automated testing frameworks like Selenium, pytest.
- Configure test cases and pipelines.

6. Create Project Structure

Organize folders:

➢ /data, /models, /src, /tests, /docs.

Document setup instructions for new developers.

7. Set Up Continuous Integration / Deployment (CI/CD)

- Use Jenkins, GitHub Actions, or GitLab CI.
- Automate build, test, and deployment processes.

8. Ensure Security

- Configure SSL for web access.
- Implement authentication systems (OAuth, JWT).

- Secure environment variables and API keys.

Deliverables

- ✔ Installed tools and libraries.
- ✔ Configured databases and cloud services.
- ✔ Project folder structure.
- ✔ Working code editor with Git integration.
- ✔ Security and access guidelines documented.

File Structure

```
SmartSDLC/
│
├── data/                  # Raw data,
datasets, logs
│   ├── requirements/
│   ├── test_results/
│   └── models_data/
│
├── models/                # Trained AI
models and checkpoints
│   ├── nlp_model/
│   ├── code_model/
│   └── performance_model/
│
├── src/                   # Source code
│   ├── requirements/      # Code for
gathering and processing requirements
│   ├── development/       # Code
generation tools integration
│   ├── testing/           # Automated
test scripts
│   ├── deployment/        # Deployment
pipeline scripts
│   ├── maintenance/       # Monitoring
and feedback systems
│   └── utils/             # Helper
functions, configurations
│
├── configs/               # Configuration
files
│   ├── database_config.yaml
│   ├── api_keys.env
│   └── model_parameters.json
│
├── tests/                 # Unit tests and
integration tests
│   ├── test_requirements.py
│   ├── test_code_generation.py
│   └── test_monitoring.py
│
├── docs/                  # Documentation
files
│   ├── architecture_diagram.png
│   ├── setup_instructions.md
│   └── user_manual.md
│
├── notebooks/             # Jupyter
notebooks for experimentation
│   ├── model_training.ipynb
│   └── data_analysis.ipynb
│
├── .gitignore             # Files to
exclude from version control
├── requirements.txt      # Python
dependencies
├── README.md              # Project
overview and instructions
└── main.py                # Entry point of
the application
```

Notes

- data/ stores inputs like requirements, logs, and datasets used for training or testing AI models.
- models/ contains saved or pre-trained AI models used in the system.
- src/ is where the main application logic is developed, separated by functionality.
- configs/ holds environment variables and other configuration settings.
- tests/ ensures your code quality with unit and integration tests.
- docs/ includes manuals, diagrams, and guidelines for developers and users.
- notebooks/ is useful for research and experimenting with AI models before integrating them into the source code.

Activity 1.4 – Develop the Core Functionalities

In this activity, you implement the key features that make the SmartSDLC system intelligent, efficient, and automated.

Core Functionalities to Develop

1. Requirement Analysis Module

Input: User requirements in text format.

Function:

➢ Use NLP models (like GPT or BERT) to process and summarize requirements.

➢ Extract key features, constraints, and priorities.

► **Output:** Structured requirement documents, use cases.

2. Code Generation Module

Input: Structured requirements or commands.

Function:

➢ Integrate with AI models like Codex to generate code snippets.

➢ Provide auto-complete and suggestions in real-time.

► **Output:** Ready-to-use code templates or functions.

3. Testing and Validation Module

Input: Code and test scenarios.

Function:

➢ Generate test cases automatically based on inputs.

➢ Run tests and report errors or performance bottlenecks.

► **Output:** Test reports and recommendations.

4. Deployment Module

Input: Verified codebase.

Function:

➢ Automate deployment using CI/CD tools.

➢ Monitor server health and rollback if needed.

Output: Deployed application with monitoring dashboards.

5. Maintenance & Monitoring Module

Input: Live system data.

Function:

- ➤ Track errors, usage patterns, and performance metrics.
- ➤ Provide alerts and automated fixes when anomalies are detected.

Output: System health reports and feedback loop for improvements.

6. User Interface

Input: User interactions and data.

Function:

- ➤ Provide a clean dashboard for developers and managers.
- ➤ Show analytics, logs, recommendations, and reports.

Output: Interactive and easy-to-use interface.

✅ Deliverables

- ✔ Working requirement analysis tool with sample inputs.
- ✔ Code generation integrated with AI models.

✔ Automated testing and reporting system.

✔ Deployment scripts and health monitoring tools.

✔ Dashboard interface with data visualization.

✅ Technologies Used

- Python, Flask/Django for backend development.
- GPT/Codex API for AI-powered tasks.
- Selenium, pytest for testing.
- Docker, Jenkins for deployment.
- React or Angular for UI development.
- PostgreSQL or MongoDB for data storage

✅ Activity 1.5 – Implement the FastAPI Backend to Manage Routing and User Input Processing

In this activity, you'll set up a FastAPI backend that handles requests, processes user inputs, and integrates with the AI models for smooth API interactions.

Steps to Implement the FastAPI Backend

1. Set Up the Project

- Install FastAPI and related libraries:
- pip install fastapi uvicorn pydantic
- Create the project structure:

SmartSDLC/

```
└── src/
    └── api/
        ├── main.py
        ├── routes/
        │   ├── requirement.py
        │   ├── codegen.py
        │   └── testing.py
        └── models/
            └── schemas.py
```

2. Create API Routes

Example routes:

/requirements/ → Accepts requirement text and returns structured output.

/codegen/ → Takes requirements and returns generated code.

/test/ → Runs tests on submitted code.

Use FastAPI's routing system to manage endpoints.

3. Define Data Schemas with Pydantic

Example schema:

```
from pydantic import BaseModel
class RequirementInput(BaseModel):
    user_id: str
    description: str
```

4. Process User Input

- Validate and parse inputs using Pydantic.
- Preprocess text before sending to AI models (e.g., clean, normalize).
- Return structured results or suggestions.

5. Integrate with AI Models

Connect endpoints with models (GPT, Codex, etc.).

Example:

```
@app.post("/requirements/")
async def analyze_requirement(input: RequirementInput):
    result = generate_summary(input.description)
    return {"summary": result}
```

6. Handle Errors and Logs

- Add exception handlers.
- Log requests and responses for debugging and analytics.

7. Run the FastAPI Server

- uvicorn src.api.main:app --reload

8. Test API Interactions

- Use tools like Postman or curl to test endpoints.
- Ensure smooth and fast responses.

✅ Sample Code Snippet – main.py

```
from fastapi import FastAPI
from routes import requirement, codegen, testing

app = FastAPI(title="SmartSDLC API")

# Include the routes
app.include_router(requirement.router, prefix="/requirements",
tags=["Requirements"])
app.include_router(codegen.router, prefix="/codegen",
tags=["Code Generation"])
app.include_router(testing.router, prefix="/test",
tags=["Testing"])
@app.get("/")
async def root():
 return {"message": "Welcome to SmartSDLC API"}
```

✅ Deliverables

- ✔ FastAPI setup with structured routes.
- ✔ Input validation and preprocessing using Pydantic.
- ✔ Smooth interaction with AI models.
- ✔ Logging and error handling.
- ✔ Documentation for API endpoints.

✅ Activity 1.6 – Writing the Main Application Logic in main.py

Here’s how you can structure and implement the core logic in main.py using FastAPI, ensuring it manages routing, processes user inputs, and interacts with AI models smoothly.

Key Features in main.py

1. Initialize the FastAPI app.
2. Include all necessary routes.
3. Handle basic requests and responses.
4. Connect input processing with AI model functions.
5. Manage error handling and logging.

✅ Sample Implementation – main.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from routes import requirement, codegen, testing

app = FastAPI(
    title="SmartSDLC AI Backend",
    description="API for managing requirements, code generation, testing, and more using AI.",
    version="1.0"
)

# Include the routes
app.include_router(requirement.router, prefix="/requirements",
```

```
tags=["Requirements"])
app.include_router(codegen.router, prefix="/codegen", tags=["Code
Generation"])
app.include_router(testing.router, prefix="/test", tags=["Testing"])

# Root endpoint to check if the API is running
@app.get("/")
async def root():
    return {"message": "Welcome to SmartSDLC API"}

# Example of a simple endpoint for health check
@app.get("/health")
async def health_check():
    return {"status": "API is running smoothly"}

# Example of handling exceptions globally
@app.exception_handler(HTTPException)
async def http_exception_handler(request, exc):
    return {"error": exc.detail}

# Example schema for demonstration
class DemoInput(BaseModel):
    text: str

# Example endpoint that uses AI logic (mocked for demonstration)
@app.post("/process/")
async def process_input(input: DemoInput):
    try:
        # Mock AI processing function
```

```
        result = f"Processed text: {input.text.upper()}"
        return {"result": result}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

✅ Explanation of Code

✅ FastAPI Initialization

The FastAPI() instance is created with metadata like title and version.

✅ Routing Setup

Routes from other modules (requirement, codegen, testing) are included for modularity.

✅ Root and Health Check Endpoints

Simple endpoints to confirm the service is working.

✅ Exception Handling

Global exception handlers are used to catch and report errors gracefully.

✅ Input Processing Example

A demonstration endpoint shows how user input is accepted, processed, and returned.

✅ AI Model Integration Placeholder

In actual use, you'd replace result = f"Processed text: {input.text.upper()}" with a call to your AI model function.

✅ Next Steps After Writing main.py

✔ Implement the actual AI logic in functions and link them to endpoints.
✔ Validate user inputs thoroughly using Pydantic schemas.
✔ Secure the API with authentication and rate limiting if needed.
✔ Write tests to ensure endpoints handle all edge cases.

✅ 3.1 – Writing the Main Application Logic in main.py

This activity focuses on implementing the central logic of your SmartSDLC application using FastAPI in the main.py file. This file acts as the entry point where requests are routed, inputs are processed, and responses are handled.

Goals of this Activity

1. Create and configure the FastAPI app.
2. Define the necessary endpoints (routes).
3. Process user inputs and pass them to AI modules.

4. Handle errors and return meaningful responses.

5. Ensure smooth interactions between users and the system.

✅ Step-by-Step Implementation

1. Install Dependencies

Make sure you have FastAPI and Uvicorn installed:

```
pip install fastapi uvicorn pydantic
```

2. Project Structure Reminder

```
SmartSDLC/
├── src/
│   ├── api/
│   │   ├── main.py
│   │   ├── routes/
│   │   │   ├── requirement.py
│   │   │   ├── codegen.py
│   │   │   └── testing.py
│   │   └── models/
│   │       └── schemas.py
├── requirements.txt
└── README.md
```

3. Sample Code for main.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from routes import requirement, codegen, testing
# Import your route modules
```

```
    # Initialize the FastAPI app
app = FastAPI(
  title="SmartSDLC API",
  description="An AI-powered software development lifecycle
management system",
  version="1.0"
)

# Include routers for modular endpoints
 app.include_router(requirement.router, prefix="/requirements",
 tags=["Requirements"])
app.include_router(codegen.router, prefix="/codegen",
 tags=["Code Generation"])
app.include_router(testing.router, prefix="/test",
 tags=["Testing"])

# Health check endpoint
@app.get("/")
async def home():
 return {"message": "Welcome to SmartSDLC API"}

  @app.get("/health")
  async def health_check():
return {"status": "Healthy and running"}

# Example input schema for demonstration
class DemoInput(BaseModel):
  user_id: str
```

```
    description: str

# Example endpoint demonstrating input processing
@app.post("/process/")
async def process_input(input: DemoInput):
    try:
        # Simulate calling an AI function (replace with real logic)
        processed_text = input.description.upper()
        return {
            "user_id": input.user_id,
            "original": input.description,
            "processed": processed_text
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Processing error:
{str(e)}")

# Global error handler example
@app.exception_handler(HTTPException)
async def http_exception_handler(request, exc):
    return {"error": exc.detail}
```

4. How It Works

- ✔ **Initialization**.
- ✔ **Routing**
- ✔ **Endpoints**
- ✔ **Input Handling**
- ✔ **Error Handling**

✅ Next Steps

- Implement real AI functions inside the route handlers.
- Secure the endpoints using authentication.
- Connect to databases and external services as needed.
- Write unit tests to validate functionality.

✅ 3.2 – Creating Dynamic Interaction with the Backend

This activity focuses on making the backend interactive and responsive to user requests in real-time. The goal is to ensure that user inputs are dynamically processed, AI models are integrated, and results are returned without delays, giving users a seamless experience.

Goals of this Activity

1. Allow users to send data dynamically through API requests.
2. Process inputs using AI models and algorithms.
3. Return meaningful responses instantly.
4. Provide feedback, logs, and error handling for better user interaction.
5. Optionally, connect the backend with a frontend interface or testing tool like Swagger UI.

✅ Steps to Create Dynamic Interaction

Define Input Schemas Using Pydantic

Use models to validate and structure incoming requests.

Example (schemas.py):

```
from pydantic import BaseModel
```

```
class RequirementInput(BaseModel):
    user_id: str
    description: str

class CodeGenInput(BaseModel):
    requirement_id: str
    language: str
```

2.Implement Interactive Endpoints

Example in requirement.py route:

```
    from fastapi import APIRouter, HTTPException
    from models.schemas import RequirementInput
router = APIRouter()

@router.post("/")
async def analyze_requirement(input: RequirementInput):
    try:
        # Simulate AI processing (replace with real model call)
        structured_output = {
            "user_id": input.user_id,
            "key_features": ["authentication", "user profile", "notifications"],
            "priority": "High"
        }
        return {"message": "Requirement processed successfully", "data":
structured_output}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to process
requirement: {str(e)}")
```

3.Handle Real-Time Requests

Use asynchronous functions (async def) to handle requests without blocking.

Allow multiple users to interact at once without waiting.

4.Integrate AI Functions

Example of integrating an AI model:

```
def generate_code(description):
    # Mock function – replace with actual AI call like OpenAI Codex
    return f"def main():\n    print('Feature: {description}')"
```

Usage inside a route:

```
@router.post("/generate")
async def generate_code_endpoint(input: CodeGenInput):
    try:
        code = generate_code(input.requirement_id)
        return {"code": code}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error generating code: {str(e)}")
```

5.Use Swagger UI for Testing

- FastAPI automatically generates Swagger UI.
- Access it at: http://localhost:8000/docs

- Test endpoints interactively without writing any frontend.

6.Add Logging for Interaction Insights

```
import logging
logging.basicConfig(level=logging.INFO)

@router.post("/")
async def analyze_requirement(input: RequirementInput):
    logging.info(f"Processing requirement for user {input.user_id}")
    # Proceed with logic...
    return {"message": "Processed"}
```

✅ Benefits of Dynamic Interaction

- ✔ Users can send requests at any time and get instant feedback.
- ✔ AI models process data in real-time, making the system smarter.
- ✔ Developers and testers can easily explore functionalities through tools like Swagger.
- ✔ Logs help track usage and debug issues.

✅ Next Steps

- Implement authentication for user-specific interactions.
- Add real AI integration with APIs like OpenAI or Hugging Face.
- Create a simple frontend using React or Angular

connect with the backend.

- Write comprehensive tests to ensure stability.

✅ 3.3 – Preparing the Application for Local Deployment

This activity focuses on configuring your SmartSDLC FastAPI application so it can run smoothly on your local machine for testing and development purposes.

Goals of this Activity

1. Set up all dependencies and configurations for local use.
2. Run the application in a development environment.
3. Test endpoints locally using Swagger or other tools.
4. Ensure proper environment management and security.

✅ Steps to Prepare for Local Deployment

1. Create a Virtual Environment
2. Install Dependencies
3. Environment Configuration
4. Run the FastAPI Server Locally
5. Database Setup (Optional)
6. Test All Endpoints
7. Organize Project Files

✅ Best Practices for Local Deployment

✔ Use .env files to manage secrets safely.

✔ Activate the virtual environment before running the server.

✔ Keep logs to debug issues during development.

✔ Regularly test endpoints using Swagger or Postman.

✔ Document setup steps in README.md for team members.

✅ Next Steps After Local Deployment

- Move towards containerization using Docker for portability.
- Set up automated testing workflows.
- Plan for cloud deployment once the application is stable locally.

✅ 3.4 – Testing and Verifying Local Deployment

This activity ensures that your SmartSDLC application is correctly set up and working as expected in the local environment before further development or production deployment.

Goals of this Activity

1. Verify that the FastAPI server is running locally.
2. Test all API endpoints to ensure they handle requests and responses correctly.
3. Validate that the application integrates with AI models and other services.
4. Confirm environment configurations, logging, and error handling are working.
5. Ensure security measures like environment variables are correctly applied.

✅ Steps to Test and Verify Local Deployment

1. Start the Local Server
2. Verify Environment Variables
3. Test Endpoints Using Swagger UI
4. Use Postman or curl for API Testing
5. Check Logging
6. Test Error Handling
7. Check Database Connection (If Applicable)

✅ Checklist – Local Deployment Verification

- ✔ Server is running and accessible via browser.
- ✔ Swagger UI shows all available endpoints.
- ✔ Inputs are validated and processed correctly.
- ✔ Responses are returned in the correct format.
- ✔ Environment variables are loaded and secure.
- ✔ Logs are displayed for requests and errors.
- ✔ Error handling is working for invalid requests.
- ✔ Database connections are active and stable.

✅ Next Steps After Testing

Document the testing steps and outcomes.
Implement automated tests using frameworks like pytest.
Prepare for staging or cloud deployment once verified.
Secure the API using authentication or permissions.

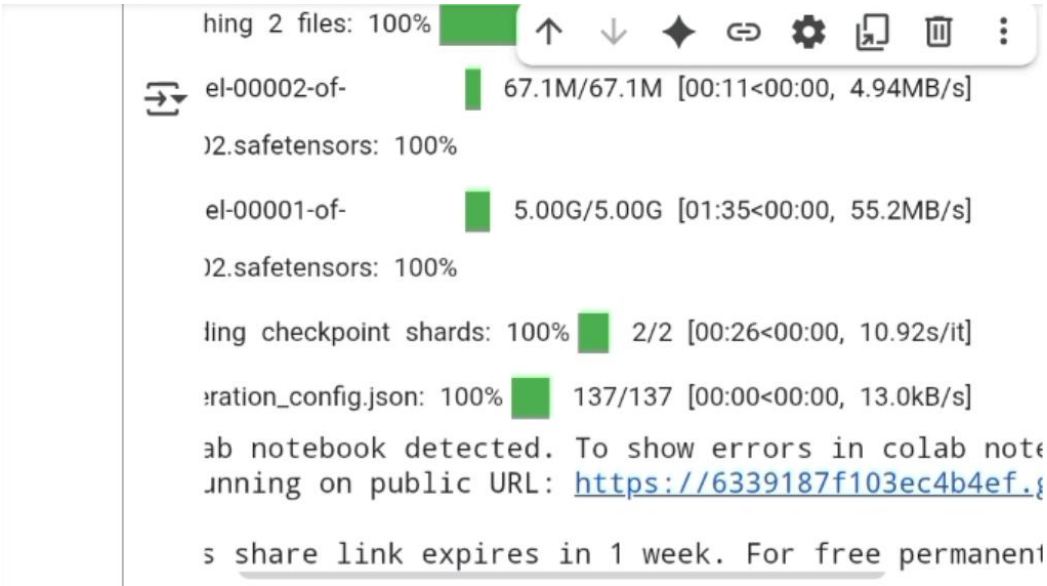
Screenshots:

```
[ ]  !pip install transformers torch gradio PyPDF2 -q
```

```
[ ]  import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCa
import PyPDF2
import io

# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_availab
    device_map="auto" if torch.cuda.is_available() els
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", tr

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in in

    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_speci
    response = response.replace(prompt, "").strip()
    return response

def extract_text_from_pdf(pdf_file):
    if pdf_file is None:
        return ""

    try:
        pdf_reader = PyPDF2.PdfReader(pdf_file)
        text = ""
        for page in pdf_reader.pages:
            text += page.extract_text() + "\n"
        return text
    except Exception as e:
        return f"Error reading PDF: {str(e)}"

def requirement_analysis(pdf_file, prompt_text):
    # Get text from PDF or prompt
    if pdf_file is not None:
        content = extract_text_from_pdf(pdf_file)
        analysis_prompt = f"Analyze the following docu
```

```
                        placeholder="Describe your software requirements..
                        lines=5
                    )
                    analyze_btn = gr.Button("Analyze")

                with gr.Column():
                    analysis_output = gr.Textbox(label="Requirements Analy

            analyze_btn.click(requirement_analysis, inputs=[pdf_upload, pr

        with gr.TabItem("Code Generation"):
            with gr.Row():
                with gr.Column():
                    code_prompt = gr.Textbox(
                        label="Code Requirements",
                        placeholder="Describe what code you want to genera
                        lines=5
                    )
                    language_dropdown = gr.Dropdown(
                        choices=["Python", "JavaScript", "Java", "C++", "(
                        label="Programming Language",
                        value="Python"
                    )
                    generate_btn = gr.Button("Generate Code")

                with gr.Column():
                    code_output = gr.Textbox(label="Generated Code", lines

            generate_btn.click(code_generation, inputs=[code_prompt, langu

app.launch(share=True)
```

```
/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: User
: `HF_TOKEN` does not exist in your Colab secrets.
icate with the Hugging Face Hub, create a token in your settings tab
e able to reuse this secret in all of your notebooks.
e that authentication is recommended but still optional to access pub
.warn(
fig.json: 8.88k/? [00:00<00:00, 272kB/s]
777k/? [00:00<00:00, 4.71MB/s]
442k/? [00:00<00:00, 5.64MB/s]
3.48M/? [00:00<00:00, 35.2MB/s]
.json: 100% 87.0/87.0 [00:00<00:00, 1.59kB/s]
s_map.json: 100% 701/701 [00:00<00:00, 15.9kB/s]
00% 786/786 [00:00<00:00, 27.5kB/s]
/pe` is deprecated! Use `dtype` instead!
nsors.index.json: 29.8k/? [00:00<00:00, 625kB/s]
les: 100% 2/2 [01:57<00:00, 117.62s/it]
-of- 67.1M/67.1M [00:15<00:00, 4.58MB/s]
nsors: 100%
-of- 5.00G/5.00G [01:57<00:00, 86.7MB/s]
nsors: 100%
kpoint shards: 0% 0/2 [00:00<?, ?
it/s]
n_config.json: 0%| | 0.00/137 [00:00<?, ?B/s]
ebook detected. To show errors in colab notebook, set debug=True in la
on public URL: https://e9a88094bc447a7d34.gradio.live

e link expires in 1 week. For free permanent hosting and GPU upgrades,
```



Generated Code

```
        self.password = password
        self.is_valid = False

    def validate(self, password):
        if self.password == password:
            self.is_valid = True
        else:
            self.is_valid = False
        return self.is_valid

    def __str__(self):
        return f"User({self.username},
{self.password}, {self.is_valid})"

# Example usage:
user = User("john_doe", "secret_password")
print(user.validate("secret_password")) # True
print(user.validate("wrong_password")) #
False
print(user) # User(john_doe,
secret_password, True)
```