**📘 System Design Documentation**

**Project:** Context-Aware PDF QA Application  
**Authors:** [Your Name]  
**Date:** [Date]

**🔧 1. Overview**

This system allows users to upload PDF documents and ask questions based on their content. The backend leverages an LLM to extract and answer questions from uploaded documents. It features:

* A React frontend (with Material-UI) for user interaction.
* A Flask backend for file handling, context management, and LLM interaction.
* Dockerized deployment for both frontend and backend.

**🧱 2. Architecture**

**High-Level Components:**

csharp

CopyEdit

[Client Browser]

|

[React + MUI Frontend (Docker)]

|

[REST API]

|

[Flask Backend (Docker)]

|

[LLM Model / API]

|

[Vector Store / In-Memory DB / LangChain]

**🖼️ 3. Frontend**

**Tech Stack:**

* React 19
* Material-UI (MUI)
* React Scripts (Webpack-based)
* Dockerized via Nginx

**Key Features:**

* Upload PDFs
* Display current backend health
* Show model currently in use
* Display response to user questions
* Error and status messages

**Component Flow:**

* App.js manages state and effects
* Uses REST APIs (/api/health, /api/upload, /api/ask, /api/model)
* Conditional rendering and loading states (e.g., spinner, alerts)

**🧠 4. Backend**

**Tech Stack:**

* Python 3.10+
* Flask + Flask-CORS
* LangChain (or custom LLM logic)
* PyPDF or PDFPlumber for PDF parsing
* Vector DB (e.g., FAISS, ChromaDB, or in-memory)
* Optional: Hugging Face / OpenAI API

**API Endpoints:**

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /api/health | GET | Check backend health |
| /api/model | GET | Return current model name |
| /api/upload | POST | Upload and parse PDF |
| /api/ask | POST | Ask a question to the model |

**LLM Interaction Flow:**

1. Parse PDF and chunk text
2. Store in memory/vector DB
3. Receive a question
4. Retrieve relevant chunks
5. Generate answer using the LLM
6. Return response

**🐳 5. Deployment (Docker)**

**Services:**

* frontend: React app served via Nginx
* backend: Flask server running on port 5000
* Communication via Docker network
* Mounted volumes for live development

**Docker Compose Example:**

yaml

CopyEdit

services:

frontend:

build: ./frontend

ports:

- "3000:80"

backend:

build: ./backend

ports:

- "5000:5000"

**🔐 6. Security**

* CORS enabled on Flask
* PDF validation on backend
* No hardcoded secrets (use .env)
* Read-only mode for model endpoint

**🧪 7. Testing**

* Frontend: React Testing Library
* Backend: Pytest for API/unit tests
* Health endpoint for integration testing

**📈 8. Scalability**

* Replace Flask with FastAPI for async processing
* Replace in-memory vector DB with persistent options (e.g., Pinecone, Weaviate)
* Use a message queue (e.g., RabbitMQ, Celery) for async PDF parsing

**📚 9. Future Improvements**

* User authentication
* File history/dashboard
* Chat-based conversational memory
* Multiple model support & switching
* Upload limits, logging, metrics

**1. Software Architecture Components**

**Separation of Concerns**

* **Frontend (React):**  
  Handles user interface, user input, and presentation logic. It sends requests to the backend and renders responses.
* **Backend (Flask):**  
  Handles business logic, API endpoints, PDF processing, querying LLMs, and serving data. It does not handle UI.
* **Data Layer:**  
  Manages persistent or temporary data (e.g., uploaded PDFs, extracted content).
* **Middleware:**  
  Includes cross-cutting concerns like CORS handling, error handling, and security layers that apply uniformly across API endpoints.

**Modularity and Maintainability**

* Each component is developed, tested, and deployed independently to simplify maintenance and upgrades.
* Clear API contracts between frontend and backend enable flexible development.

**Scalability and Performance**

* Dockerized services for easy scaling and deployment.
* Backend designed to handle asynchronous requests for PDF processing and querying.

**2. Cross-Origin Resource Sharing (CORS)**

**Overview**

* Browsers enforce a **same-origin policy** preventing frontend code from making API calls to a different domain or port for security.
* Since frontend (localhost:3000) and backend (localhost:5000) run on different ports (different origins), **CORS headers** must be configured to allow requests.

**Implementation in Backend**

* The Flask backend uses the **flask-cors** middleware to allow cross-origin requests from the frontend.
* Example configuration:

python

CopyEdit

from flask import Flask

from flask\_cors import CORS

app = Flask(\_\_name\_\_)

CORS(app, origins=["http://localhost:3000"]) # Only allow frontend origin

* This setup adds appropriate Access-Control-Allow-Origin headers in HTTP responses, enabling browsers to allow frontend API calls.

**Security Implications**

* Enables legitimate frontend-backend communication while preventing unauthorized cross-site requests.
* Helps prevent attacks like **Cross-Site Request Forgery (CSRF)** by controlling allowed origins.
* Should be configured with **strict origin whitelisting** in production.