# High-Level Design (HLD)

## Overview

**This system allows users to upload, store, and interact with documents through NLP and RAG agents. It will have the following components:**

1. **Frontend (React.js)**
   - User Interface to interact with the system.
   - Document upload and management features.
   - Query interface to ask questions based on document content.
   - Session-based authentication or OAuth2.0 for user management.

2. **Backend (FastAPI)**
   - Handles business logic, user authentication, and API requests.
   - Provides endpoints for document upload, metadata extraction, and NLP-based query handling.
   - Integrates with LangChain for document processing and RAG querying.

3. **Document Storage**
   - File Storage: AWS S3 or equivalent, used to store the actual documents (PDF, PPT, CSV).
   - Database: MongoDB

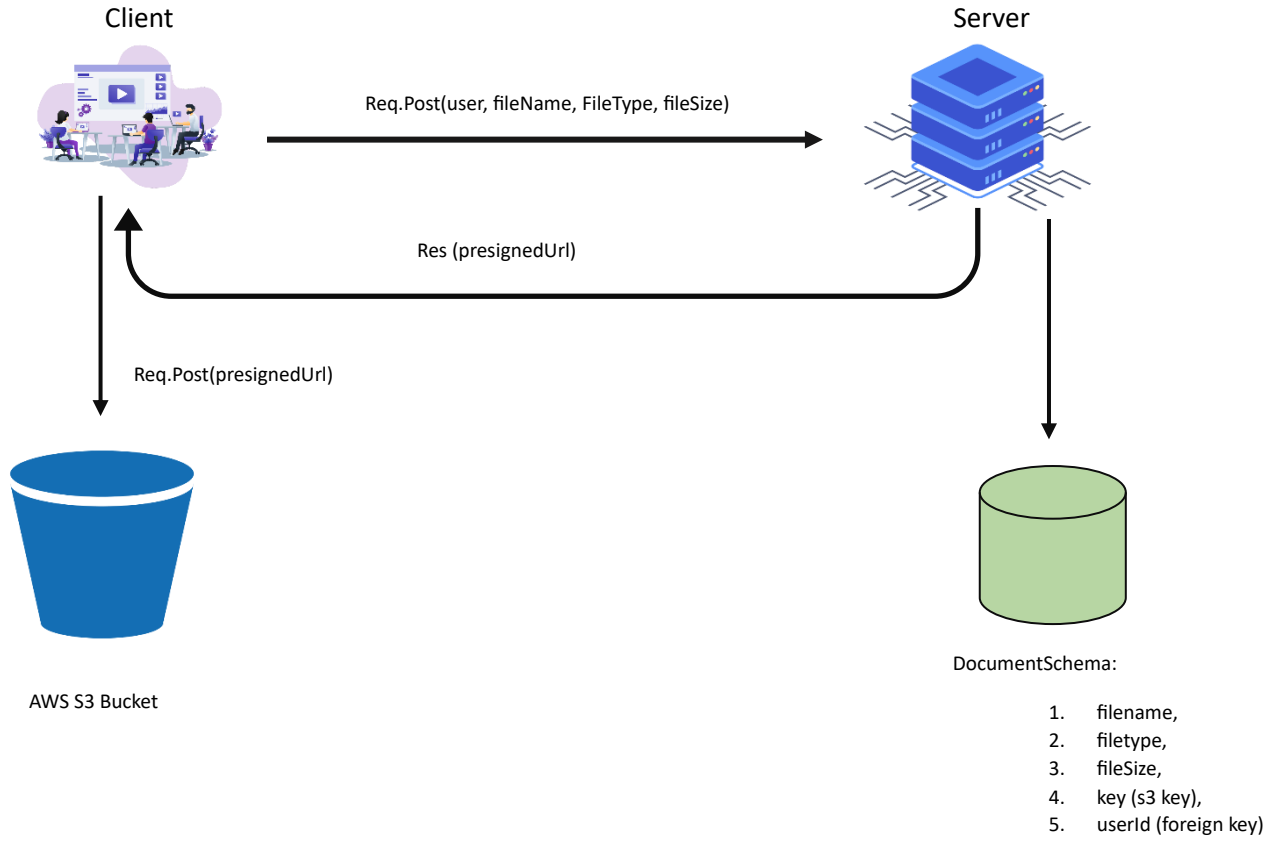
4. **NLP Processing**
   - Use unstructured.io for parsing document content and extracting metadata.
   - LangChain for indexing and search capabilities.
   - Pinecone for querying the document database with RAG agents, providing contextual answers.

4. **Authentication**
   - JWT/OAuth2.0 for secure authentication.
   - Session management to ensure user-specific data handling.

## ❖ System Components and Interactions

- **User Interaction:** Users will upload documents via the frontend, which is sent to the backend for storage and parsing.
- **Document Parsing**: Once a document is uploaded, unstructured.io parses it to extract key content (text, metadata). The parsed data is indexed using LangChain.
- **NLP & RAG Query**: When a user queries the system, the backend uses Pinecone to retrieve relevant content from the indexed documents and generate a response based on context.

## ❖ Data Flow:



## ❖ Low-Level Design (LLD)

### 1. Database Schema (PostgreSQL)

- **Tables Structure:**

| Users | Documents | Queries |
|---|---|---|
| • user_id (PK | • document_id (PK) | • query_id (PK) |
| • username | • user_id (FK to Users) | • user_id (FK to Users) |
| • email | • file_url (URL to S3 file) | • query_text |
| • password_hash | • metadata (JSONB, for parsed metadata like author, date, etc.) | • timestamp |
| • created_at | • uploaded_at | • response_text |
| | • parsed_at | |

- **Foreign Keys:**
- **Users → Documents (user_id).**
- **Documents → Document_Metadata (document_id).**

### 2. Classes and Functions

❖ **Classes:**

- **User:** Handles user-related operations (registration, authentication, password hashing).
- **Document**: Manages document upload, parsing, and metadata extraction.
- **Query Agent:** Interfaces with the RAG system to generate responses to user queries.
- **Document Parser**: Uses unstructured.io to parse documents and extract key data.
- **Search Engine**: Interfaces with Elasticsearch to search indexed documents.

❖ **Functions:**

- ✓ **User.signup()** : Creates a new user.
- ✓ **User.verifyJWT()** : Authenticates a user via JWT/OAuth2.0.
- ✓ **User.login()** : Authenticate user and respond with Access Token
- ✓ **OTP.sendOTP()** : Verify User Creation and Send OTP to email.
- ✓ **OTP.verifyOTP():** verify OTP and make user verified
- ✓ **Document.uploadDocument()** : Uploads a document to S3 and records metadata in MongoDB.
- ✓ **Document.getAllDocument()**: load all documents and send response
- ✓ **Document.getDocument()** : load document with provided id
- ✓ **Upload-to-pincone()** : Invokes unstructured.io to parse the document.
- ✓ **query()** : Uses the RAG agent to generate a response based on a query.

## 3. Interaction Between Classes

- **User & Document:**
  A user can upload multiple documents. Each document is associated with the user through a foreign key.
- **Document & DocumentParser:**
  When a document is uploaded, it triggers the parsing process using unstructured.io. Metadata is then extracted and stored in the database.
- **Document & SearchEngine:**
  The document is indexed in Elasticsearch after parsing to enable quick retrieval for user queries.
- **QueryAgent & SearchEngine:**
  When a query is made, the QueryAgent retrieves the relevant documents using SearchEngine, and uses the RAG approach to generate a response.

## 5. OPEN-CLOSE Relationships and Dependency Handling

### ❖ OPEN-CLOSE Principle:

The design allows for easy extension. For example, adding a new document format (e.g., Word) would require extending the DocumentParser class without modifying existing functionality.

### ❖ Dependency Injection:

Components like DocumentParser, SearchEngine, and QueryAgent are loosely coupled using dependency injection to allow for easy testing and swapping of components.

## 6. Authentication

### ❖ JWT or OAuth2.0:

- ✓ Used JWT tokens for session-based authentication, and alternatively, OAuth2.0 for integrating third-party authentication services.
- ✓ For each API request, the user's token is validated before accessing protected routes.

## 7. Key Features:

- Multi-format document handling.
- NLP-based query responses with RAG agents.
- Metadata-driven search and categorization.
- User authentication with JWT/OAuth2.0.
- calable and secure deployment using Docker and Kubernetes.