```
      ┌────────────────────────┐
      │  User (e.g., Doctor)   │
      └────────────────────────┘
           │
           ▼
┌──────────────────────────────────────────┐
│ FastAPI API (RBAC, JWT, Async, Query Caching) │
│  - JWT Auth & Role Extraction            │◄──────────────────┐
│  - Asynchronous calls for hybrid search  │                   │
│  - Caches common queries in Redis        │                   │
└──────────────────────────────────────────┘                   │
           │                 │
           ▼                 │
    ┌─────────────────────────┐                       │
    │ Pre-filtering Layer     │                       │
    │ - Redis Cache           │                       │
    │ - MongoDB (User-PDF Maps) │                     │
    └─────────────────────────┘                       │
           │                 │
           ▼                 │
   ┌──────────────────────────────────────┐                    │
   │    Hybrid Search Engine         │      │
   │  ┌──────────┐    ┌─────────────┐      │      │
   │  │ BM25 via │    │ Vector Search │    │      │
   │  │OpenSearch│    │ (Pinecone)  │      │      │
   │  └──────────┘    └─────────────┘      │      │
   └──────────────────────────────────────┘                    │
           │                 │
           ▼                 │
   ┌──────────────────────────────────────┐                    │
   │  Ranking & Fusion Service            │◄───────────────┘
│  - Score Normalization (Min-Max, RRF)     │
│  - Dynamic Weighting (λ based on query)   │
│  - Optional Cross-Encoder Reranking       │
└──────────────────────────────────────────────┘
          │
          ▼
┌──────────────────────────────┐
│ PDF Storage (Amazon S3)   │
│  - Pre-signed URLs        │
│  - Metadata, previews     │
└──────────────────────────────┘
          │
          ▼
┌──────────────────────────────┐
│ Logging & Monitoring Stack │
│ - Prometheus, Grafana, ELK │
│ - AWS CloudWatch          │
└──────────────────────────────┘
          │
          ▼
┌──────────────────────────────┐
│ Indexing Pipeline (Lambda) │
│ - Triggered on S3 Upload   │
│ - Text Extraction & Embedding│
│ - Update OpenSearch & Pinecone│
│ - Refresh MongoDB mappings │
└──────────────────────────────┘
```

Key Components & How They Address Issues

1. **User Authentication & RBAC**

- **What: FastAPI validates JWT tokens and extracts user roles.**
- **Why: Ensures that only authorized users access PDFs according to permissions defined in MongoDB.**

2. **Pre-filtering Layer**
   - **What: Queries MongoDB for user–PDF mappings and caches results in Redis.**
   - **Why: Reduces database load and latency on frequent role-to-PDF lookups.**
3. **Query Caching**
   - **What: Caches frequent search queries in Redis (with a short TTL).**
   - **Why: Improves performance by avoiding redundant search calls.**
4. **Hybrid Search Engine**
   - **What: Executes BM25 search via OpenSearch and semantic search via Pinecone in parallel (using async calls).**
   - **Why: Combines exact keyword matching with contextual similarity while minimizing latency.**
5. **Ranking & Fusion Service**
   - **What: Normalizes scores, applies dynamic weighting (or RRF), and optionally uses a cross-encoder for reranking top results.**
   - **Why: Ensures the final ranked list reflects both relevance and query intent, adapting to different query types.**
6. **PDF Storage (Amazon S3)**
   - **What: Stores PDFs securely and provides pre-signed URLs for controlled access.**
   - **Why: Protects PDF access while ensuring that metadata and previews are returned to aid user selection.**
7. **Indexing Pipeline (Lambda)**
   - **What: Automatically triggers on new PDF uploads to extract text, generate embeddings, and update search indices.**
   - **Why: Keeps your search indices up-to-date without manual re-indexing.**
8. **Error Handling & Fallbacks**
   - **What: FastAPI's try-catch blocks, fallback to direct MongoDB queries if Redis fails, partial result returns if a search component is down.**
   - **Why: Enhances reliability and ensures graceful degradation in case of component failures.**
9. **Scalability Management & Observability**

- **What: Monitors system performance with Prometheus, Grafana, ELK, and AWS CloudWatch; scales OpenSearch/Pinecone as needed.**
- **Why: Proactively tracks and manages performance bottlenecks and resource demands.**

File structure

```
├── app/
│   ├── __init__.py
│   ├── main.py          # FastAPI app entry point with exception handlers
│   ├── config.py        # Environment configurations
│   ├── auth.py          # JWT authentication and RBAC utilities
│   ├── exceptions/      # Custom exceptions
│   │   ├── __init__.py
│   │   └── custom_errors.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── user.py      # Pydantic models for users
│   │   └── pdf.py       # Models for PDF metadata and results
│   ├── routes/
│   │   ├── __init__.py
│   │   └── search.py    # API endpoints
│   ├── services/
│   │   ├── __init__.py
│   │   ├── pdf_filter.py    # Pre-filtering with Redis/MongoDB
│   │   ├── hybrid_search.py   # Parallel BM25 & vector search with snippets
│   │   ├── ranking.py       # Ranking and fusion logic
│   │   └── s3_storage.py    # S3 URL generation
│   ├── workers/         # Background tasks
│   │   ├── __init__.py
│  │  ├── tasks.py        # Cross-encoder reranking, etc.
│  │  └── indexer.py       # Incremental indexing logic
│  └── utils/
│     ├── __init__.py
│     ├── helpers.py       # General utilities
│     ├── query_cache.py    # Query result caching
│     ├── logging.py       # Structured logging
│     └── metrics.py       # Prometheus metrics
├── lambda/              # (Optional) AWS Lambda for indexing
│  ├── indexer_lambda.py     # Lambda handler for S3 events
│  └── requirements.txt
├── tests/               # Unit and integration tests
│  ├── __init__.py
│  ├── test_auth.py
│  ├── test_search.py
│  └── test_services.py
├── .apprunner.yaml        # AWS App Runner config
├── Dockerfile            # Docker build instructions
├── requirements.txt        # Dependencies
└── README.md             # Documentation
```

```
project/
├── app/
│  ├── __init__.py
│  ├── main.py
│  ├── config.py
│   ├── auth.py
│   ├── models/
│   │   ├── user.py
│   │   └── pdf.py
│   ├── routes/
│   │   └── search.py
│   ├── services/
│   │   ├── pdf_filter.py
│   │   ├── hybrid_search.py
│   │   ├── ranking.py
│   │   └── s3_storage.py
│   ├── utils/
│   │   ├── helpers.py
│   │   └── query_cache.py
│   ├── exceptions/
│   │   └── custom_errors.py
│   ├── workers/
│   │   ├── tasks.py
│   │   └── indexer.py
├── lambda/
│   └── indexer_lambda.py
├── tests/
│   └── test_auth.py
├── Dockerfile
├── .apprunner.yaml
├── requirements.txt
├── .env  # Not included here; user must create this
└── README.md
```

Main Functions Defined In our project:-

1. User Uploads PDF: Done ✅

- 📁 **File Upload Component**
- 📤 Uploads PDF to AWS S3

2. Processing the PDF: Done ✅

- 📜 Extract text from the PDF
- ✅ Preprocess (clean, remove unnecessary characters)

3. Chunking the Text: Done ✅

- 🔗 Divide text into smaller chunks
- 📏 Maintain contextual meaning

4. Embedding the Chunks: Done ✅

- 🧠 Convert chunks into vector embeddings using an NLP model Sentence Transformer

5. Storing the Chunks: half Done ✅

- 💾 Store the vector embeddings in Pinecone for retrieval

- Store Metadata in MongoDB

6. Search & Retrieval

- 🔍 Query processing using OpenSearch
- 📊 Find similar text chunks from Pinecone
- 🏆 Return best-matching results

Authentication and Other feature in our project:

1. User Registration: Done ✅

- 📝 User enters details (username, email, password, etc.)
- 📩 System sends OTP to user’s email

2. OTP Verification: Done ✅

- 🔑 User enters OTP received via email
- ✅ System verifies OTP
- 🎉 User is successfully registered

3. User Login: Done ✅

- 🔐 User enters credentials
- 🔑 System verifies credentials
- 🎟 System generates authentication token

4. Upload Route Access: Done ✅

- ✅ Authenticated users can now access the **Upload PDF** route
- 📁 User uploads PDF (leads to PDF processing flow)

Flow Structure for Profile Page (CRUD Operations)

1. User Authentication & Profile Access: Done ✅

- 🔑 User logs in (Token-based authentication)
- 🏠 User navigates to the profile page

2. Update User Details (PUT Request): Done ✅

- 📝 User modifies profile details (name, email, password, etc.)
- 📤 System updates change in the database
- ✅ Confirmation message is shown

3. Fetch Uploaded PDFs (GET Request):

- 📁 System retrieves all PDFs uploaded by the user
- 👀 User sees a list of uploaded PDFs

4. Delete a PDF (DELETE Request):

- 🗑️ User selects a PDF to delete
- ❌ System removes the PDF from AWS S3 & database

- ✅ Confirmation message is shown

5. Delete Account (DELETE Request): Done ✅

- 🚨 User requests account deletion
- 🔐 System verifies user action (optional re-authentication)
- 🗑️ Deletes all user data (profile + PDFs)
- ✅ Logs out user and redirects to the home page

6. Logout (Token Expiry) Done ✅

- 🚪 User clicks logout
- 🔥 System clears authentication token
- 🔄 Redirects to login page

Folder Structure and their Works:

📂 `bucket/` (Manages AWS & Pinecone Initialization)

- 📄 **`AWSBucket.py`**– Initializes AWS S3 for file storage
  - Configures AWS SDK
  - Handles file uploads & deletions
- 📄 **`PineconeBucket.py`** – Initializes Pinecone for vector storage
  - Connects to Pinecone
  - Manages vector embeddings storage & retrieval

🗁 `controller/` (Handles Routing, No Business Logic)

This folder defines API endpoints and delegates business logic to injected services.

🖹 *`AccountController.py`*

🔹 **Manages User Authentication Routes**

- `POST /register` → Calls service to register a new user & send OTP.
- `POST /login` → Calls service to authenticate user & return token.

✅ **Dependencies Passed to Services:**

- `Auth service` (Handles authentication & token generation)

🖹 *`UploadController.py`*

🔹 **Manages File Upload & Processing Routes**

- `POST /upload` → Calls service to handle PDF upload to AWS S3.
- `GET /fetch-files` → Calls service to retrieve user's uploaded PDFs.

✅ **Dependencies Passed to Services:**

- `s3_service` (Uploads & deletes files from AWS S3)
- `text service` (Extracts & processes text from PDFs)
- `embedding service` (Embeds chunks & stores in Pinecone)
- `pinecone service` (Manages vector storage & retrieval)

🗁 db/ (Manages Database Connection & GridFS Integration)

🖹 *connection.py*

🔹 **Initializes database connection & GridFS for file storage**

- Connects to MongoDB using **Mongo Client**
- Creates a user's collection for storing user data
- Initializes **GridFS** for handling large file uploads (PDFs)
- Ensures the database connection is accessible across the app

✅ **Key Components:**

- `client = Mongo Client (MONGO_URI)` → Connects to MongoDB
- `dB = client["your_database_name"]` → Selects database
- `users_collection = db["users"]` → Defines user collection
- `grid_fs = GridFS(db)` → Initializes GridFS for file storage

🗁 `collection/` (Defines Database Models)

🖹 *`userCollection.py`*

🔹 **Defines User Registration Schema**

- Specifies the **structure of user documents** in MongoDB
- Ensures fields like `email`, `password_hash`, `created_at`, etc.
- Implements any pre-processing (e.g., hashing passwords before storing)

🗁 `diInjector/` (Manages Dependency Injection for Services)

🖹 *`diExtension.py`*

🔹 **Injects Core Services into the Application**

- **Registers services globally** so controllers can access them
- Ensures **loose coupling** (controllers don't directly instantiate services)
- Helps with **scalability & testing** (easily replaceable service implementations)

🗁 `dtos/` (Data Transfer Objects)

- Structures and validates incoming user data before processing.
- Ensures correct format for **register, login, upload, etc.**
- Example: `LoginDTO`, `RegisterDTO`, `UploadDTO`.

📂 `schema/` (Validation Layer)

- Defines schemas to validate user input before storing in the database.
- Ensures that required fields are provided and formatted correctly.
- Example:
  - `register_schema.py`: Validates user registration details.
  - `upload_schema.py`: Ensures correct file format and size.

3️⃣ `routes/` (API Endpoints)

- Defines all API routes for user interaction.
- Forwards requests to the **controller**.
- Example routes:
  - `POST /register` → Calls `AccountController`
  - `POST /upload` → Calls `UploadController`

📂 `service/` (Business Logic Layer)

- Implements actual **business logic** for register, login, and upload.
- Calls **database operations** and handles **tokens, file uploads, etc.**
- Example:
  - `account_service.py`: Handles user creation, password hashing, and authentication.
  - `upload_service.py`: Processes PDFs, extracts text, creates embeddings.

🗁 `JWTConfig/` (Authentication & Token Management)

- Generates and verifies authentication tokens for users.
- Manages token expiration and refresh logic.

🗁 `middleware/` (Access Control & Security)

- Restricts unauthorized users from accessing certain routes.
- Ensures only authenticated users with **client ID** can perform actions.

🗁 `utils/` (Helper & Utility Functions)

- Contains reusable helper functions like:
  - **Embedding chunks of text** (for Pinecone storage).
  - **Sending emails** (for OTP verification, password reset, etc.).
  - **File handling** (e.g., processing PDFs before uploading).

🗁 Folder Responsibilities

Folder	Responsibility
`bucket/`	Initializes AWS S3 & Pinecone for filestorage
`controller/`	Defines API endpoints but no business logic
`db/`	Manages MongoDB connection (GridFS for PDF storage)

`collection/`	Defines user schema & database models
`diInjector/`	Injects services into controllers using Dependency Injection
`dtos/`	Defines **Data Transfer Objects (DTOs)** for structured API requests
`enum/`	Stores **predefined constants** (user roles, status codes, etc.)
`exception/`	Handles custom **error handling & predefined exceptions**
`helper/`	Stores helper functions (password hashing, JWT handling)
`interface/`	Defines **service interfaces** for register, login, upload, etc.
`JWTconfig/`	Manages JWT configuration (secret key, token expiry)
`lambda/`	AWS Lambda functions (if applicable)
`middleware/`	Restricts user access based on `client_id`
`model/`	Defines expected **input/output** data models
`routes/`	Registers all API routes
`schema/`	Handles **schema validation** (registration, login, upload)
`service/`	Implements **business logic** (register, login, upload, embedding)

`template/`	Stores **email templates** (OTP, welcome email, etc.)
`utils/`	Stores **utility functions** (embedding, email sending, file processing)