ChatGPT

Design Plan: RBAC Model Update for Document Analysis Tool

Introduction and Goals

This design document outlines the **architecture and implementation plan** for updating the existing Document Analysis Tool to a new Role-Based Access Control (RBAC) model with three roles: **Admin, Manager, and Analyst**. The application currently provides document upload, summarization, knowledge graph generation, and chat features. The goal of this redesign is to implement a hierarchical RBAC system that governs data access and user management, replacing the previous station/district/state scheme. We will detail backend changes (database schema and services), frontend component updates (using Shadcn UI for tables), and integration of local services (SQLite, Neo4j, Redis, Tesseract OCR) to meet the new requirements. The deliverables include updated schema designs, component responsibilities, sequence flows, a migration strategy for existing data, and a stepwise implementation roadmap.

RBAC Requirements and Role Definitions

The new RBAC model introduces three user roles with distinct permissions and data access scopes [1]:

- **Admin** – Top-level role responsible for system user management. Admins can create and delete Manager accounts, assign Analysts to Managers, and set credentials (email/password). Admins may also view all users and data for oversight, though their primary function is user administration.
- **Manager** – Mid-level role managing a team of Analysts. A Manager account is created by an Admin (with initial credentials). Managers can log in and perform CRUD operations on their own Analysts (create new Analyst accounts under them or remove them). Managers have **read access to all documents, job results, knowledge graphs, and chats belonging to their Analysts**. They cannot see data from other managers' teams.
- **Analyst** – Base role for end-users who upload and analyze documents. Analysts can log in (credentials created by a Manager or Admin) and use the system's features for **their own documents only**. They can upload documents, trigger summarization, view the generated summaries and translations, see transcriptions (for audio/image files), view the knowledge graph extracted from their documents, and interact with the chat interface for Q&A on their data. Analysts **cannot** access documents or jobs of other analysts.

This hierarchical RBAC ensures that each role only accesses authorized data (a best practice for dashboards and multi-user systems [1]). It also reflects a multi-tenant style separation, where an Admin oversees multiple Managers, and each Manager oversees a group of Analysts [2]. The RBAC enforcement will apply both at the API level (authorization checks on requests) and at the data retrieval level (query filtering by role and ownership).

System Architecture Overview

Figure: High-Level System Architecture – The application follows a client–server architecture with clear separation of frontend and backend responsibilities (diagram omitted for brevity). The **Frontend** is a React application (using Shadcn/UI components for a consistent design) that provides different views depending on the user's role (Admin, Manager, Analyst). The **Backend** is a Python-based API server (e.g. FastAPI or Flask) that exposes endpoints for authentication, user management, document processing, and data retrieval. Supporting services include a SQLite database for persistent storage, a Neo4j graph database for knowledge graphs, and a Redis instance for caching and task queue management. All components are self-hosted (local deployment) as per requirements.

Key backend services/components:

- **Auth & RBAC Middleware:** Handles authentication (likely via JWT or session) and checks the user's role on each request. Ensures that, for example, an Analyst cannot access another user's data (enforce owner-only access), and Managers only see their team's data.
- **User Management Service:** Exposes APIs for Admin to create/delete Managers, assign Analysts to Managers, and for Managers to create/delete their Analysts. This service will interface with the DB to store user accounts and roles.
- **Document Processing Pipeline:** Handles document uploads from Analysts. This component orchestrates file storage, text extraction or transcription (using Tesseract OCR for images/PDFs and possibly speech-to-text for audio), summarization (using the "Gemini" LLM or other local models), translation (for summaries or extracted text), and knowledge graph extraction. It may utilize background workers (with Redis queues) to perform heavy tasks asynchronously.
- **Knowledge Graph Service:** Interfaces with the local Neo4j database to store and query extracted entities and relationships from documents. It builds a graph per job (or per set of documents) and supports queries from the chat or UI to visualize connections.
- **Chat Service:** Allows interactive Q&A and discussion based on the documents' content. Likely uses an LLM (e.g., Gemini) with Retrieval Augmented Generation, pulling context from the summarized text or knowledge graph. It may cache conversation context in Redis for each session.

All configuration (file paths, DB connection strings, API keys for local models, etc.) will be handled via **environment variables** (using `os.getenv()` in the code for settings like `DB_PATH`, `NEO4J_URL`, etc.), in line with twelve-factor app principles. The application's local mode (no external cloud services) means that all services run within the user's environment, and no external network calls are required except possibly for initial model downloads.

Database Schema Design

To support the new RBAC and track documents and jobs, the SQLite database schema will be redesigned with new tables and relationships. Below is the proposed schema (tables and key fields):

- **Users** (`user_id` INTEGER PK, `email` TEXT UNIQUE, `password_hash` TEXT, `role` TEXT, `manager_id` INTEGER, `created_by` INTEGER, ...)
  This table stores user accounts. The `role` field is one of {Admin, Manager, Analyst}. For an Analyst user, `manager_id` references the Users table (the Manager they are assigned to). For a Manager user, `manager_id` may be NULL (or could reference an Admin who created them, if we want to track it). The `created_by` could store the user_id of the Admin or Manager who created that account (for audit). All passwords are stored hashed.

- **Jobs** (`job_id` INTEGER PK, `manager_id` INTEGER, `analyst_id` INTEGER, `job_name` TEXT, `timestamp` DATETIME, …)
  Each "job" represents an analysis session or a grouping of one or more uploaded documents processed together by an Analyst. `analyst_id` references Users(user_id) and `manager_id` as well (redundant but denormalized for convenience and quick filtering by manager). The `job_name` or identifier will incorporate the Manager and Analyst context. We will enforce a naming convention such as `managerId_analystId_jobId` for uniqueness. For example, if Manager 5's Analyst 12 runs a job with ID 7, we might store a composite string "5/12/7". This can be stored either as a generated column or constructed on the fly for display. The **format "manager/analyst/job"** will also be used in any file paths or identifiers for easy tracing. The Jobs table may also store metadata like job status (processing/complete), or results summary info (e.g., number of documents, languages detected).

- **Documents** (`doc_id` INTEGER PK, `job_id` INTEGER, `file_name` TEXT, `content_type` TEXT, `text_extracted` BOOLEAN, `transcription_done` BOOLEAN, `summary` TEXT, `translation` TEXT, `created_at` DATETIME, …)
  This table tracks each document uploaded in a job. It stores the original file name and some metadata on what processing has been done. Flags like `text_extracted`, `transcription_done` indicate how the text was obtained (extracted from text PDF or transcribed via OCR/audio). The `summary` and `translation` fields might contain the text summary and translated summary (if applicable). Alternatively, these large texts could be stored in separate tables or files, but for local simplicity they could be here or in separate files on disk with paths stored. Each document links to its Job via `job_id`.

- **KnowledgeGraphNodes** and **KnowledgeGraphEdges** (or a separate Neo4j store)
  Since we have Neo4j, we might not store the graph in SQLite at all. Instead, when a job is processed, we create nodes and relationships in Neo4j labeled/tagged by job or analyst. For instance, all nodes from job X could have a property `job_id = X` for scoping. We will ensure that queries to Neo4j include a filter on job or user context so that a Manager only retrieves nodes for their team's jobs, and analysts only for their own. If needed, a minimal reference could be kept in SQLite (like a table of `graph_id` per job) but likely not necessary.

- **ChatLogs** (`id` PK, `job_id`, `user_id`, `message`, `sender` ENUM('user','assistant'), `timestamp`)
  Optionally, to support persistent chat history, we could log the conversations per job. This can help Managers viewing an Analyst's Q&A or resuming a chat context. This table would store the message transcripts. (If not needed long-term, Redis or memory could hold transient chat context instead.)

ER Diagram (simplified):
Users (user_id, role, manager_id) –< (manager_id) – **Jobs** –< (job_id) – **Documents**. Each Analyst is linked to one Manager (Users.manager_id), each Job is linked to one Analyst and one Manager, and each Document links to a Job. Admins are simply users with no manager_id needed. Managers can have many Analysts; each Analyst has exactly one Manager; each Manager (or Admin) can have many Jobs via their analysts.

Role Hierarchies: While not a strict inheritance, we can view Admin > Manager > Analyst as a hierarchy for permissions (Admin can do all Manager actions; Manager can do all Analyst actions except where limited to their scope). This can be implemented either via code logic or a possible `Roles` table if needed. Given the small number of roles, we may hardcode permissions in code (e.g., a dictionary mapping roles to allowed actions).

Backend Components and Services

User Authentication & RBAC Enforcement

All users (Admin, Manager, Analyst) will authenticate, for example using email/password with secure hashing (e.g., Bcrypt). On login, a token (JWT) or session cookie is issued containing the user's role and ID. For each API request, middleware will verify the token and enforce RBAC rules. We will implement decorators or middleware that check if a user has access to the resource they're requesting. For instance: - Admin endpoints (e.g., create user) will check `user.role == Admin`. - Manager endpoints (e.g., create Analyst or list jobs) will check `user.role == Manager` and allow only actions scoped to their own ID (for listing Analysts, only those with manager_id = their user_id). - Analyst endpoints (document upload, retrieving their job results) will ensure `user.role == Analyst` and allow access only to their own resources (analyst's user_id must match the record's analyst_id).

We'll use the new **manager/analyst/job** identifiers to quickly validate scope: when an Analyst requests job data, the path or query will include their user ID or name, which must match the authenticated user's ID (or if a Manager, must match one of their team's IDs). This naming scheme helps partition data access in code. This approach aligns with common multi-tenant RBAC strategies where tenant-scoping is enforced in middleware [3] [2].

User Management Service (Admin & Manager Operations)

Admin functions: The backend will provide endpoints like `POST /admin/managers` (to create a Manager), `DELETE /admin/managers/{id}`, `POST /admin/analysts` (to create an Analyst under a Manager), etc. Admin can assign an Analyst to a Manager by specifying the manager_id on creation (or later update). These endpoints will perform input validation (e.g., only Admin can create a Manager, ensure unique emails, etc.). Passwords for new accounts can be generated or provided by Admin; in our case Admin sets a temp password which the user can later change. Admin's view of users is global, so the service may also allow listing all Managers and their Analysts for the admin dashboard.

Manager functions: The backend will have similar endpoints but limited in scope. E.g., `GET /manager/analysts` returns all Analysts where manager_id = (current Manager's user_id). `POST /manager/analysts` creates a new Analyst user with that manager_id. Managers cannot create other Managers or change anything outside their team. They also can fetch all jobs or documents of their analysts (e.g., `GET /manager/jobs` could return all jobs for analysts under that manager, possibly with filter by analyst). When a Manager deletes an Analyst, we must decide what to do with that Analyst's jobs and documents – likely we will keep them for record (mark Analyst inactive) or possibly allow cascading delete. We will specify that deletion might just deactivate accounts to avoid data loss.

Document Processing Pipeline

When an **Analyst uploads documents**, the frontend will call an endpoint like `POST /analyst/jobs` with files. The backend will create a new Job entry in the database, constructing the `job_id` and storing the manager_id (by looking up the analyst's manager) and analyst_id. The files will be saved to a local storage directory (e.g., under `./data/{managerId}/{analystId}/{jobId}/` for easy organization). Each file will get stored and a record in the Documents table.

Processing Steps: (These can be executed synchronously for simplicity or asynchronously using a background worker if jobs are long-running.)

1. **Text Extraction or Transcription:** For each uploaded document, determine if it's text-based (PDF, DOCX, etc.), or image-based (scanned PDF/image) or audio. If text-based, extract text directly (e.g., PDF parsing); if image or scanned, use **Tesseract OCR** to transcribe text. Tesseract will be configured with support for Indian regional languages as needed (by installing appropriate `.traineddata` files for languages like Hindi, Tamil, etc., and even Mandarin Chinese). *Tesseract supports over 100 languages with proper training data* [4], so integrating languages is a matter of including the language codes (e.g., using `pytesseract.image_to_string(..., lang='hin+eng')` for Hindi+English). We'll ensure that for multilingual documents, the OCR is run with the appropriate language packs.
2. **Summarization:** The extracted or transcribed text then goes through a summarization module. We'll use the **Gemini** model (or an equivalent local large language model) to generate a concise summary of each document's content. This could be done via an API call to a local model server or library. The summary text is then saved (either in the Documents table or as a separate text file).
3. **Translation (optional):** If required by the user or by default for certain languages, the summary (or full text) is translated into another language (perhaps English if the original was not, or vice versa). This adds an additional step using an LLM or translation library. Not all analyses will include translation, but the system should support it.
4. **Knowledge Graph Construction:** In parallel with summarization, an entity extraction step runs on the text. This can use an NLP pipeline or LLM to identify key entities (people, organizations, dates, etc.) and relations. For example, if the documents are research papers, entities might be authors, topics, etc. These entities and their relationships are then inserted into the Neo4j graph database. Each node and relationship is tagged with the job context. For instance, we might create a Neo4j node with labels `Entity` and properties like `name="XYZ", type="Person", job_id=7, analyst_id=12`. Relations (edges) connecting entities are similarly labeled. This allows query and visualization of the knowledge graph for that job. (We might also generate a static JSON of the graph for frontend visualization if not querying Neo4j directly from the client).
5. **Storing Outputs:** After processing, each Document record is updated with flags indicating what outputs are available (text extracted, summary done, translation done, etc.). Additionally, the system will name or label output files to reflect the data. According to the specification, we use a notation in file names:
6. `2'` (2 prime) – indicates **extracted text + summary** are available (two pieces of content, obtained via text extraction).
7. `3'` (3 prime) – indicates **extracted text + summary + translation** (three pieces, with translation of the text or summary).
8. `2''` (2 double-prime) – indicates **transcribed text + summary** (text was obtained via transcription, e.g., OCR or audio, and summarized).
9. `3''` – indicates **transcribed text + summary + translation**.

We will incorporate this in the file naming. For example, if an image PDF "Invoice123.pdf" was processed and summary generated, we might store the combined text+summary file as "Invoice123_2''.txt" (double quote to show it was OCR transcription). If translation to another language was also done, it becomes "Invoice123_3''.txt". These file names (or the information to derive them) will be stored in the Document record, so the frontend knows which outputs exist. This naming scheme makes it immediately clear what processing level has been done for each file.

The **Pipeline Orchestration** can be done within a Celery worker (with Redis as broker) to avoid blocking the web request. The upload endpoint would respond quickly with a job ID and perhaps an initial status, then a worker picks up the job to do steps 1–5. The frontend can poll a status endpoint or use WebSocket for completion notification. Given local deployment, a simpler approach is to process synchronously but with a spinner on the UI.

Knowledge Graph and Querying

Neo4j will store the graph data per job. We will design a Cypher query template for retrieving the subgraph of a given job or analyst. For example, when an Analyst or Manager opens the knowledge graph view for job X, the backend will run `MATCH (n)-[r]->(m) WHERE n.job_id = X RETURN ...` to get all nodes and relationships, which the frontend can render (perhaps using a D3.js graph or a component). The graph provides an interactive way to explore connections in the documents. This component is crucial for the **Manager**, who might collate insights from all an analyst's docs.

Chat Service Integration

The chat feature allows users to ask questions about the uploaded documents and receive answers (like a chatbot that knows the content). We will integrate this by leveraging the documents' text, summaries, and knowledge graph as context for an LLM. For instance, when a user asks a question, the backend will retrieve relevant text snippets (possibly vector search or graph traversal) from the job's data and then prompt an LLM (Gemini or GPT) with those details to generate an answer. The **Redis** cache might be used to store the chat session context (conversation history) so the LLM can have memory of previous questions. This is a form of Retrieval-Augmented Generation: it first retrieves from the private corpus (the uploaded docs) then answers with an LLM [5] [6].

Managers can use the chat on any of their analysts' jobs. The backend will verify that the Manager has access (by checking manager_id match) and then proceed with the same retrieval+LLM process on that job's data. Admins, if needed, could be allowed to specify a manager and analyst to view their chat, but that's outside normal use.

Local Deployment Considerations

All components will run locally: - **SQLite** for the DB (file-based, no separate server needed). Migrations will be done via scripts or an ORM. - **Neo4j** will run locally (the app will connect via Bolt or HTTP driver to `localhost` Neo4j instance). - **Redis** local for caching and queue – ensure to namespace keys by environment if needed, but since all local, it's straightforward. - **Tesseract OCR** must be installed on the machine with appropriate language packs for Indian languages (Hindi – `hin`, Tamil – `tam`, etc. and Chinese – `chi_sim` for Simplified, etc.). The code will use `pytesseract` to call Tesseract. For example: `pytesseract.image_to_string(image, lang='eng+hin')` to handle mixed English-Hindi documents. Tesseract's multi-language support is robust [7] [4], but quality depends on training data – we will document that the user should install Tesseract language data for all required languages. - **Gemini/LLM**: The plan assumes a local LLM (maybe hosted via an API or running within the app). We will use environment variables to configure its endpoint or model path (e.g., `GEMINI_API_URL`). The design does not rely on cloud APIs, aligning with data privacy for sensitive documents.

Security: Although local, we still implement proper authentication and data segregation. Also, audit logging can be added (e.g., log whenever a Manager accesses an analyst's data).

Frontend Component Design

The frontend will be structured into distinct views for each role, using a modern component-based framework (React) and the Shadcn UI library for consistent styling (tables, forms, modals). We will outline the component hierarchy and responsibilities for each role's interface:

- **Admin Dashboard:** When an Admin logs in, they see a management dashboard. The main element is a **User Management Table** (using Shadcn's Data Table component) listing all Managers and their associated Analysts. This could be presented as a master-detail or tree table: each row is a Manager, and expanding a row reveals that Manager's Analysts. Alternatively, it can be two tables side by side – select a Manager on the left, see their Analysts on the right. The table supports CRUD operations:
- Create Manager (opens a form modal to input manager details, then calls backend API).
- Delete Manager (with confirmation; this might also cascade delete their analysts or require reassigning those analysts – we'll define a safe default, e.g., require no analysts or auto-delete them too).

- For Analysts, Admin can also create or delete them (when needed to assist managers). Possibly Admin can reassign an Analyst to a different Manager (this would be a special action – e.g., an "edit" on an Analyst row to select a different manager from a dropdown).
  The Admin Dashboard component tree might look like: `AdminDashboardPage -> ManagerTable -> AnalystTable`. Each table row has action buttons (edit/delete), and uses Shadcn UI's styling for a polished look.

- **Manager Dashboard:** When a Manager logs in, they see a dashboard focused on their team and their team's work. There are two main functions: managing Analysts and reviewing jobs/documents. We can split the UI into two tabs or sections:

- **Analyst Management:** a table (similar to Admin's view but limited to their analysts). Here the Manager can add a new Analyst (enter name, email, password), or remove an Analyst. This likely reuses a Table component for listing Analysts with create/delete actions.
- **Jobs/Documents Access:** The Manager should be able to view any job from any of their Analysts. We will provide a way to select an Analyst (e.g., a dropdown or a list at left) and then see a list of that Analyst's past jobs. Upon selecting a job, the Manager is taken to a **Job Detail View** very much like what the Analyst would see for their own job. Essentially, the Manager can "impersonate" the analyst's view in a read-only manner. The Job Detail includes the list of documents and all the results (summaries, etc.), plus the knowledge graph and chat interface for that job.

Concretely, the Manager Dashboard could have an `AnalystList` on the side, and a `JobList` for the selected analyst. Clicking a job opens the detailed view (perhaps in a modal or new page component). We will reuse the Analyst's components for viewing summary, graph, chat, etc., but with a breadcrumb indicating whose job it is.

The knowledge graph for a job might be visualized using a graph component (if available in Shadcn or custom). The chat interface will be a shared component as well.

Component tree sample: `ManagerDashboardPage -> AnalystList -> {selected Analyst} -> JobList -> {selected Job} -> JobDetailView (with DocumentList, SummaryViewer, KnowledgeGraphViewer, ChatInterface)`.

- **Analyst Interface:** The Analyst's experience centers on processing documents and exploring results. There are two primary screens for Analysts:
- **Upload/New Job Workflow:** A page where the Analyst can create a new job by uploading one or multiple documents. The UI will have an upload form (file picker, maybe drag-and-drop area). After selecting files, the user triggers "Process" which calls the backend. While processing, the UI can show a progress indicator or "Job is processing..." message. Once complete (the front-end might poll an endpoint for job status), it navigates to the Job Results page.

- **Job Results (Browse Jobs) View:** This page shows the outcome of a processed job or allows browsing previous jobs. It might open automatically after an upload completes, but the Analyst should also have a way to see old jobs. So we will have a sidebar or dropdown of **Job IDs** (or job names/timestamps) the analyst has done. Selecting a job populates the detail view of that job. If the Analyst just finished an upload, that job will be selected.

  In the Job detail view, for each uploaded document, the available outputs are shown. For example, list the file names and provide links or sections for: - **Extracted Text** (if needed – could be large, so maybe we skip showing raw text in UI and focus on summary), - **Summary** – display the summary text. - **Translation** – if available, this could be toggled or in a separate tab (original summary vs translated). - **Transcription** – if the file was audio or image, the recognized text can be shown or downloaded.

  We can use a tabbed interface or accordion for each document. For instance, clicking on a document name could expand to show its summary and translation. Alternatively, if multiple documents were part of one job, we might aggregate the summary (maybe the system generates a combined summary for all, depending on use case).

  The **Knowledge Graph** for the job is accessible via a "Graph" tab or section. When the user opens it, it loads the graph visualization of entities.

  The **Chat** interface will likely be on the Job page as well – possibly in a sidebar or a bottom panel. The user can ask questions in a textbox, and answers appear in a chat log view. This chat is contextual to the selected job, so the queries the user asks are only about the documents in that job. The component will call an endpoint like `/chat` with the question and job_id; responses stream or come back and are displayed.

Component tree sample: `AnalystJobsPage -> JobList (sidebar) -> JobDetailView -> [DocumentResultsList, KnowledgeGraphView, ChatInterface]`. Within DocumentResultsList, each Document might have subcomponents for summary, etc.

All UI components will utilize **Shadcn UI** for a consistent look. For example, the tables will use Shadcn's Table which includes features like sorting, filtering out of the box [8] [9]. Forms and modals will use Shadcn form and dialog components. We will also ensure the frontend has proper state management (perhaps using React Context or Redux) to store user info (including role, so we conditionally render appropriate nav items).

Responsive design & Accessibility: The use of Shadcn (Radix UI under the hood) ensures basic accessibility. We will confirm that Managers and Analysts (who may upload large files or see complex graphs) have a responsive layout that works on common screen sizes (mostly desktop use is assumed for heavy analysis, but we'll use responsive flex/grid as needed).

Sequence Flows and Diagrams

To illustrate how the updated system will function, we describe key use-case flows:

1. Analyst Uploads Documents (New Job Creation)

1. **Analyst initiates upload:** On the Analyst UI, the user selects one or more files and clicks "Upload & Analyze". The frontend sends a request `POST /analyst/jobs` with the files (multipart form data).
2. **API receives request:** The backend Auth middleware confirms the token is valid and the user role is Analyst. The request handler creates a new Job record in the DB (`Jobs` table) with analyst_id = current user, manager_id = current user's manager. It generates a new `job_id`. It responds immediately with a job identifier and status "processing" (and perhaps some initial data).
3. **File storage:** The server saves the uploaded files to disk (e.g., in a directory named after job_id). It also creates Document records for each file (with `summary`, `translation` fields empty initially, and flags set to false).
4. **Processing pipeline:** The backend either starts processing synchronously or dispatches a background job (e.g., via Celery). For each document:
5. If PDF/text: extract text. If image-based: run Tesseract OCR. If audio: run speech-to-text.
6. Summarize the text using the LLM (Gemini). Save the summary (and also keep the full text if needed for chat).
7. If configured, translate the summary or text (e.g., to English) and save it.
8. Identify entities/relations and upsert into Neo4j (with job context).
9. Mark in the Document record which outputs are ready. Determine the naming suffix: e.g., a text PDF yields 2' or 3' depending on translation; an image OCR yields 2'' or 3''. Store the output files or texts accordingly (files might be named with those suffixes and saved).
10. **Completion:** Once all files are processed, the Job record is updated (e.g., status = "complete" and maybe finished timestamp).
11. **Frontend polling:** Meanwhile, the frontend might be polling an endpoint like `GET /analyst/jobs/{job_id}` to check if processing is done. When done, it receives the job data including list of documents and their available outputs. The UI then navigates the Analyst to the Job Results page for that new job, displaying summaries, etc.
12. **Result display:** The Analyst can now view each document's summary and other outputs. They can switch to the Knowledge Graph tab to see a visualization of extracted entities. They can also open the Chat tab to ask questions about the documents.

Sequence Diagram (Analyst Upload Flow):

```
Analyst UI -> Backend API: POST /analyst/jobs (files)
API -> DB (SQLite): INSERT job, docs (analyst_id, manager_id)
API -> FileSystem: save files under manager/analyst/job path
API -> Queue/Processor: trigger process_job(job_id)
Processor -> Tesseract: OCR text (if needed)
Processor -> LLM (Gemini): Summarize text
Processor -> LLM: Translate summary (if needed)
Processor -> Neo4j: CREATE nodes/edges for entities
Processor -> DB: UPDATE documents (flags, summary text, etc.), UPDATE job
status
Analyst UI -> Backend API: GET /analyst/jobs/{job_id} (polling)
Backend API -> DB: SELECT job & docs (with outputs)
Backend API -> Analyst UI: returns job data (status complete, summaries,
etc.)
Analyst UI: renders Summary, Graph, Chat components for that job
```

2. Manager Views Analyst's Job Results

1. **Manager selects analyst:** On the Manager's dashboard, they choose an Analyst from their list. The UI calls `GET /manager/{analyst_id}/jobs` to fetch that analyst's jobs. Backend verifies the manager_id matches the authenticated Manager and returns the list of jobs (IDs, names, dates).
2. **Manager selects job:** The Manager picks a specific job from that list. The frontend then requests `GET /manager/jobs/{job_id}`. The backend confirms this job's manager_id matches the current Manager. It then retrieves the job's documents, summaries, etc., similar to an analyst would, and returns that data.
3. **Viewing data:** The Manager UI now shows the job detail view. The components for documents, summary, etc., are the same as the Analyst's view, but possibly in read-only mode (Managers won't upload in this context). The Manager can read all summaries and even download original files if allowed.
4. **Knowledge Graph and Chat:** The Manager can open the knowledge graph view for that job – the data is fetched from Neo4j (likely via the same endpoint the analyst UI would use, e.g., `GET /jobs/{job_id}/graph` returning nodes/edges JSON). Since the Manager has rights, the backend supplies the graph data. The Manager can also use the chat interface to ask questions about this job's documents. The chat component calls `POST /chat` with the question and job context; the backend processes it with the LLM and returns an answer. The Manager sees the Q&A thread, which might be logged or ephemeral. This allows managers to derive insights or verify the analyst's work.

3. Admin Manages Users

1. **Create Manager:** On the Admin UI, the admin fills a "New Manager" form (name, email, temp password) and submits. The frontend calls `POST /admin/managers` with this data. Backend authenticates as Admin and then inserts a new Users record with role=Manager. Optionally an email could be sent out-of-band, but since local, perhaps just provide the credentials to the admin. The new manager appears in the Admin's table immediately (the API returns the created user object).
2. **Create Analyst:** The admin chooses a Manager in the table (perhaps using an action "Add Analyst" on that row). The admin enters the analyst info, and the frontend calls `POST /admin/analysts` with the data and the selected manager's ID. Backend verifies admin, creates Users record with role=Analyst and manager_id set. The new analyst appears under that manager's list in the UI. (Similarly, a Manager could perform this action via their own view, using their own endpoint `POST /manager/analysts` – the flow is the same except the manager_id is implicitly the current user).
3. **Delete User:** If an Admin deletes a Manager, the system must decide what to do with child records. We might implement this as a cascade that first deletes (or reassigns) that manager's analysts. Since the requirement explicitly allows Admin to delete Managers and Managers to

delete Analysts, we implement safe deletions. For example, when a Manager is deleted by Admin, we could also delete all their analysts and jobs (with a warning). Alternatively, mark them as inactive. In our plan, we will implement **soft deletes** (set a `active` flag false) to avoid losing data. The UI will filter out inactive users. Deletion endpoints will thus either perform soft delete or require the caller to confirm cascade.

4. **View Overview:** The Admin's table view provides a quick overview of system users. Admin can see how many analysts each manager has, etc. If needed, Admin could also have access to see all jobs (a global view), though not explicitly required – we mention it as a possibility since Admin has the highest privilege. We could include an "Impersonate Manager/Analyst" feature for Admin (for support/debugging), but that's beyond core scope and would be a future enhancement.

These flows ensure each role's responsibilities are fulfilled while respecting access limits. The system will log actions like creation/deletion of users and jobs for audit (especially important in enterprise contexts [10], though for local setup this can be minimal logging to console or a log file).

Migration Plan for Existing Data and Code

Updating to the new RBAC model will require migrating both **database records** and possibly **code paths** that assumed the old station/district/state roles. Below are recommendations for a smooth transition:

- **Database Migration:** We will write a migration script to alter the existing schema:
- Drop or ignore old role tables/fields (station, district, state). If the old model stored roles in separate tables or as a field like `role = station|district|state`, we'll map those to the new roles. For example, "state" level users likely become Admin, "district" become Manager, "station" become Analyst. We will confirm with domain context – assuming it aligns (state=top level -> Admin, district=mid -> Manager, station=local -> Analyst).
- Introduce the new Users table or alter existing user table to have `role` and `manager_id`. If an existing table of users exists, add a column `role` and populate it based on old data: e.g., if a user had `level = "district"` in old system, set role = "Manager". If the old schema linked users to a district or station entity, we use that to infer manager relationships.
- Create the Jobs and Documents tables. If the old system already had a concept of "jobs" or similar, we will migrate those. Possibly earlier the documents were just listed under a station or user. Now we encapsulate them into jobs. We may need to create one job per existing document if previously each upload was separate. Or if there was already a grouping, map it accordingly. For initial migration, we can create a job for each document owned by an analyst with the current date as timestamp, or group by some session if available. Mark manager_id and analyst_id properly.
- Migrate document records: If old system stored documents with station/district references, we now replace that with job references and user references. File storage paths might change to the new structured scheme (manager/analyst/job). We can choose to physically reorganize files into new folders named by the new scheme, or we can update references in DB and leave files in place if not critical. Given local environment and presumably not millions of files, reorganizing is feasible. We'll write a script to move files: e.g., for each document record, determine its analyst and manager, create a new directory if needed, move the file, and update the file path accordingly in the DB.

- Neo4j data: If knowledge graphs were stored with old labels or properties linking to old roles or IDs, we need to update those to the new scheme. For example, if nodes had a property station_id, replace with analyst_id/manager_id. If feasible, wiping and regenerating the graphs might be simpler unless there's a lot of historical data to preserve. Since we can regenerate from source docs, one approach is to flush Neo4j and allow graphs to be rebuilt on first access or via a one-time migration job per document (this depends on how easily we can run entity extraction again; if not, we map existing graph relationships to new owners by matching old IDs).

- **Code Refactoring:** Identify all places in code that used the old RBAC logic:

- Remove checks or references to station/district/state. For instance, if code did `if user.role == "station": ...` replace with `if user.role == "Analyst": ...` etc. Update any authorization logic to the new hierarchy.
- Update API endpoint URLs if they included old terms. Perhaps previously endpoints were like `/api/v1/district/{id}/...` – these should change to `/manager/{id}/...` or similar. Ensure the frontend is updated accordingly to call new endpoints. If backward compatibility is needed for a short time, we could alias old endpoints to new logic, but since this is a major change, it’s fine to have the frontend switch entirely to new API paths.
- Update any hardcoded role names or enumerations. E.g., if a dropdown in UI listed “Station, District, State”, change to “Analyst, Manager, Admin”.

- Testing: After migration, test each role’s typical actions to ensure they only see what they’re supposed to. Particularly, verify that an Analyst cannot access another’s job by manipulating IDs (the API should prevent it). Also test Manager access to ensure they can see their analysts but not others. Possibly attempt an admin function as manager to confirm it’s blocked, etc.

- **Data Integrity Post-Migration:** For each Analyst user, ensure the `manager_id` is correctly set. This likely means each station-level user had a parent district – so assign that as manager. Each Manager user (district-level) in old system might not have had an explicit parent (state-level oversight) – but now we might not need to assign them an admin_id since Admin is more of a global role. If multiple Admins are allowed, we might add a field created_by, but not strictly needed.

- **Communication to Users:** Since roles and capabilities are changing, provide documentation or training for users if this were a production scenario. In a local dev scenario, ensure the development team knows the new login flow (e.g., an Admin must exist – possibly we create a default Admin account in the new system or migrate an existing “state” user to Admin role).

In summary, the migration will involve a database schema update (can be done with Alembic or manual SQL scripts) and careful data transformation to fit the new relational structure. Because the model is simpler (3 roles instead of geographic hierarchy), the data volume for roles is small – this migration should be straightforward with scripted queries. Testing the migration on a copy of the data is recommended before applying to the live environment.

Implementation Roadmap

We propose an iterative implementation with clear milestones:

1. **Schema Setup (Milestone 1):** Define the new SQLite schema. Write migrations to create new tables (Jobs, Documents) and alter Users for `role` and `manager_id`. Test the schema with sample data. Verify that constraints (foreign keys) work for cascading deletes if used.
2. **Basic RBAC Backend (Milestone 2):** Implement authentication and role checking middleware. Create dummy endpoints for user management and have them enforce roles (e.g., protect `/admin/*` routes to Admin only). At this stage, seed the database with one Admin user for

testing. Confirm that a login returns a token and that token's role is interpreted correctly in an authorized route.

3. **User Management Features (Milestone 3):** Develop the Admin endpoints (create Manager, list Managers with Analysts, etc.) and Manager endpoints (create Analyst, list Analysts). Simultaneously, build the corresponding frontend pages: Admin dashboard with table, Manager view with their analysts. Use static data or simple API calls returning JSON from the new endpoints. Ensure create/delete flows work end-to-end (UI form -> API -> DB changes -> UI update). This milestone completes the RBAC user hierarchy management.
4. **Document Processing Backend (Milestone 4):** Implement the upload handler and processing pipeline. Start with synchronous processing for simplicity: get a file, run OCR (if needed), produce a fake summary (for initial integration testing) and store records. Gradually integrate real OCR (ensure Tesseract is installed and tested on a sample image), real summarization (if Gemini API or local model is available – otherwise stub this out with a placeholder text to not block development). Implement storing results and marking document status. By end of this milestone, an Analyst can upload a document and get a summary from the system.
5. **Knowledge Graph Integration (Milestone 5):** Set up the Neo4j instance and add code to connect (using Neo4j Python driver). Implement a simple entity extraction (could use a Python NLP library or even regex as a placeholder). When processing docs, create some sample nodes/edges in Neo4j. Develop an API endpoint like `/jobs/{job_id}/graph` that queries Neo4j and returns a list of nodes/links. On the frontend, create a component to display the graph. This could be as simple as listing entities for now, or integrate a graph visualization library for React. Ensure that this data is only accessible to the correct users (check job's manager_id/analyst_id vs requester).
6. **Chat Feature (Milestone 6):** Integrate the chat service. Possibly use a local LLM (if available) or an API call stub. Focus on the mechanics: an endpoint `/chat` that accepts {job_id, question} and returns an answer string. For now, the "answer" could concatenate some dummy info (or use the summary text as knowledge). Later, integrate the actual LLM (Gemini) with proper prompt construction using the job's content. Build the frontend chat UI (text input, send button, display messages). Use web sockets or polling for response streaming if needed, or simply one-shot Q&A for now. This milestone will significantly enhance the interactive capability for users.
7. **Finalize Frontend UI/UX (Milestone 7):** At this point, all major features are in place. Now refine the frontend: add loading spinners and status messages (e.g., during processing of documents or waiting for chat answer). Polish the layout for the job results page (maybe use tabs for Summary/Translation/Graph/Chat). Ensure the Shadcn UI components are properly styled (maybe adjust Tailwind classes as needed). Also implement error handling on UI (e.g., show a notification if an API call fails or if user tries to access unauthorized data).
8. **Testing & Quality Assurance (Milestone 8):** Conduct thorough testing:
9. Unit tests for backend functions (e.g., permission checks, OCR function on sample images, summarization output given known input, database CRUD ops).
10. Integration tests where possible (simulate an Analyst uploading and ensure a Manager can retrieve the results).
11. Manual testing of the UI for all roles. Particularly test edge cases like: an Analyst account with no jobs (empty state), a Manager with no analysts yet, deletion of a user who has existing data (does the UI handle it gracefully?), large documents for OCR (performance).
12. If possible, user acceptance testing with a sample of actual documents to ensure the summarization and graph meet expectations.
13. **Documentation & Deployment (Milestone 9):** Update README or user guide for the application to explain the new roles and how to use the system. Document how to set environment variables for local services (like pointing the app to the local Neo4j URI, etc.). Then prepare the deployment (if it's just local, ensure all dependencies like Tesseract and Neo4j are

installed and configured on the host machine). Migrate the existing database using the scripts from step 1, and run the new application.

Throughout these milestones, we will use version control to manage changes and possibly feature flags to merge incomplete features without breaking the main branch. By completing all steps, the application will have a robust RBAC framework and improved features ready for use.

Conclusion

In this design, we established a clear RBAC hierarchy and restructured the application around Admin, Manager, and Analyst roles to improve security and manageability. We detailed database changes supporting hierarchical ownership of jobs and documents, and how backend services will enforce role-based access at every step. The frontend will provide tailored experiences for each role – from an admin control panel to manager oversight dashboards to analyst workbenches – all using modern UI components for clarity. We also integrated the analytical features (OCR, summarization, knowledge graphs, chat) into this RBAC model, ensuring that the flow of data adheres to role permissions. The plan includes visualizing architecture and processes and offers a careful migration strategy to transition from the old model without data loss. By following the outlined implementation roadmap, the development team can iteratively build and verify each piece of the system, resulting in a secure, user-friendly application that meets the new requirements and enhances the overall functionality for document analysis.

References: Role-based access control best practices emphasize restricting access by user roles (e.g., admin, manager, analyst) [1], which aligns with our design. Similar multi-tenant architectures implement Admin/Manager/Analyst hierarchies and graph databases for data isolation [2]. The OCR component leverages Tesseract, which supports over 100 languages including Indic scripts and Mandarin for robust text extraction [4]. These principles and technologies guided the architecture proposed here.

---

[1] Security And Access Control In Dashboard Design - FasterCapital
https://fastercapital.com/topics/security-and-access-control-in-dashboard-design.html/1

[2] [3] SaaS B2B Multi-Tenant Design - AI Prompt
https://docsbot.ai/prompts/technical/saas-b2b-multi-tenant-design

[4] [7] How to Run Tesseract OCR for Hindi-English Language: Full Setup, Best Config & Sample Result | by Aditya Mangal | Medium
https://adityamangal98.medium.com/how-to-run-tesseract-ocr-for-hindi-english-language-full-setup-best-config-sample-result-8af763d68f73

[5] [6] GraphRAG: Practical Guide to Supercharge RAG with Knowledge Graphs
https://learnopencv.com/graphrag-explained-knowledge-graphs-medical/

[8] Data Table - Shadcn UI
https://ui.shadcn.com/docs/components/data-table

[9] Shadcn Data Table
https://www.shadcn.io/ui/data-table

[10] 6 Examples of Role Based Access Control (RBAC) Architecture | DEVOPSdigest
https://www.devopsdigest.com/6-examples-of-role-based-access-control-rbac-architecture