Nice — I'll explain the **problem** clearly and then describe a concrete, implementable **solution** you can build as the MVP (matching your requirements). I'll cover architecture, core components, data flow, algorithms, security/access, frontend UX, evaluation metrics, and a prioritized implementation checklist you can act on immediately.

Problem (concise)

Organizations receive large volumes of unstructured documents (PDFs, DOCX, TXT, emails) that must be organized, searched, and audited. Manual tagging is slow, inconsistent, and scales poorly; keyword search misses semantic intent. Key challenges:

- **Scale & variability:** many file types, formats, scanned docs, inconsistent metadata.
- **Semantic retrieval:** users want "meaning-based" search (find contract clauses, invoices with overdue amounts) not just keyword matches.
- **Metadata quality:** titles/authors/dates are embedded in unpredictable ways.
- **Privacy/security:** documents are often confidential — using public LLMs/services may be unacceptable.
- **Role-based access:** different teams (HR / Finance) must only see authorized documents.
- **Timebox:** build a working, demo-ready MVP quickly (hackathon-style).

Desired outcome

A working system that:

1. Automatically classifies uploaded documents into categories (Finance, HR, Legal, Contracts, Technical Reports, etc.).
2. Extracts metadata (title, author, date, entities) and an extractive summary (top 3–5 sentences).
3. Indexes documents using dense embeddings so semantic queries return ranked, relevant results.
4. Enforces simple role-based access and logs actions.
5. Provides a frontend dashboard for upload, view, search, download.

High-level solution (one-line)

An offline-first pipeline that extracts text → extracts metadata & entities → summarizes (extractive) → creates sentence-transformer embeddings → indexes them in FAISS → exposes role-protected REST endpoints + a small dashboard for upload/search/view.

Architecture & components

- **Frontend:** Streamlit (fast demo) or Next.js + Tailwind (production-looking). UI: Login, Upload, Document list, Document view, Search+filters.
- **Backend:** FastAPI (async), serves upload/search/document endpoints, enforces RBAC.
- **Storage:** Local filesystem for files, **SQLite** for metadata, users, access logs.
- **Text extraction:** `pdfplumber` for PDFs, `python-docx` for DOCX, plain read for TXT. (Optional OCR later)
- **Metadata & NER:** spaCy (en_core_web_sm) + regex heuristics for emails, dates, amounts. DOCX bold detection for title heuristics.
- **Summarization:** Extractive TF-IDF / sentence scoring (scikit-learn) — deterministic and fast.
- **Embeddings:** SentenceTransformers (`all-MiniLM-L6-v2` recommended) — lightweight and high quality.
- **Vector index:** FAISS (local) for similarity search; simple idmap persisted to map vectors → document IDs.
- **Classification:** Zero-shot NLI (transformers `bart-large-mnli`) or embedding + small k-NN against seeded labeled examples.
- **Auth & security:** SQLite users + hashed passwords (bcrypt/passlib), JWT or simple session tokens, role checks on endpoints. Log all uploads/views/searches.
- **Orchestration:** Optionally use LangChain Document loaders + chains to make swapping LLMs or adding declarative chains easier.

Data flow (step-by-step)

1. **Upload** → backend saves file, stores filename/uploader/time in SQLite.
2. **Text extraction** from file (PDF/DOCX/TXT).

3. **Metadata extraction**:
    - Title: DOCX bold-first line, else first substantial line.
    - Author: regex patterns for "Author:", "By:", or first email.
    - Date: regex for common date formats.
    - Entities: spaCy + regex for amounts/emails.
4. **Summarization**: split into sentences → TF-IDF scoring → pick top 3–5 sentences.
5. **Classification**: run zero-shot classifier or embedding-based kNN to pick category.
6. **Embedding & indexing**: embed document (summary or chunked text) → add to FAISS → persist mapping doc_id → vector_idx.
7. **Persist metadata**: write final record (title, author, date, category, summary, entities) to SQLite.
8. **Access control**: when a user queries/list/downloads, filter by role and log action.

Key algorithms & rationale

- **Extractive summarization (TF-IDF):** computed per-sentence TF-IDF score, fast and non-proprietary — returns relevant sentences without calling LLMs.
- **Semantic search:** SentenceTransformers → FAISS with cosine similarity (L2-normalize vectors + inner product) gives fast approximate nearest neighbors.
- **Classification options:**
    - *Zero-shot NLI* for quick setup (no training).
    - *Embedding + Seeded k-NN* for fully local approach and lower inference cost.
    - *Supervised* (scikit-learn) if you have labeled training docs.
- **NER & metadata:** spaCy covers common entities; add regex for structured fields (email, amounts, dates). DOCX bold detection improves title extraction.

Security & privacy considerations

- Keep all processing local / self-hosted to avoid sending documents to third-party APIs.

- Hash passwords (bcrypt) and use JWT/session tokens for API auth.
- Role-based enforcement at query level: DB queries must always include `WHERE category = <allowed>` for limited roles.
- Store access logs (user, action, doc_id, timestamp) for audit.
- For production: encrypt files at rest, use HTTPS, and rotate tokens. If using external LLMs later, consider private deployment or on-prem models with strict data controls.

Frontend UX (core screens)

- **Login:** username/password → receives token; UI hides unauthorized categories.
- **Dashboard:** Upload button; table of documents (Title, Category, Author, Upload Date) filtered by role.
- **Document view:** shows extracted metadata, the summary, entities list, and a Download link.
- **Search UI:** search box for semantic search; filters for Category, Author, Date range; results show snippet + relevance score.

Evaluation metrics (how to validate)

- **Classification accuracy**: sample labeled test set (precision/recall per category).
- **Search quality**: human-rated relevance (NDCG or MAP) on sample queries.
- **Summary quality**: ROUGE or user inspection for top-3 sentences.
- **Extraction precision**: fraction of documents where title/author/date extracted correctly.
- **Performance:** index build/query latency (ms for top-k), memory usage.
- **Security audit:** verify missing role access doesn't leak docs.

Demo script (what to show)

1. Login as HR user.
2. Upload HR onboarding DOCX → show title/author extraction, automatic HR classification, summary, entities.
3. Login as Finance user → upload invoice → show invoice classified correctly and searchable.

4. Run a semantic search (“overdue payment”, “severance”) and show ranked results.
5. Show that HR user cannot see Finance document (RBAC enforced) and view access logs.

Prioritized implementation checklist (what to build first)

1. Core backend + DB models (users, documents, entities, logs).
2. File upload endpoint + local save + text extraction.
3. Metadata extraction + extractive summarizer → persist summary.
4. Simple classification (zero-shot or k-NN seed).
5. Embeddings + FAISS index + search endpoint.
6. Basic auth & role-based filtering.
7. Minimal frontend (Streamlit) for upload/search/list/view.
8. Add logging, mapping of vectors→doc ids, and small demo documents.

(You can build a fully demo-able MVP with these pieces without adding OCR or heavy LLM calls.)

Risks & mitigations

- **Large PDFs or scanned docs**: add OCR (Tesseract) only if needed; otherwise show “unsupported” or recommend text-based docs.
- **FAISS/embedding memory pressure**: start with in-memory or small FAISS; persist index to disk; switch to a managed vector DB if scale increases.
- **Model download size**: choose lightweight `all-MiniLM-L6-v2`; avoid huge NLI models unless necessary.
- **Incorrect metadata extraction**: provide a manual edit UI in frontend for quick corrections during demo.

Extensibility (next steps after MVP)

- Add OCR for scanned PDFs.
- Add abstractive summarization via private LLMs if allowed.
- Migrate SQLite → PostgreSQL for concurrent multi-user use.
- Add worker queue (Celery/RQ) for async embedding & heavy tasks.
- Add advanced RBAC & audit export for compliance.

Final takeaway (one-paragraph)

Build a self-contained, privacy-first pipeline: upload → extract → classify → summarize → embed → index → search, all orchestrated by FastAPI and backed by lightweight models (sentence-transformers, spaCy) and FAISS. The result is a demo-ready system that delivers robust semantic search, metadata extraction, and role-based access — exactly what organizations need to tame unstructured document collections while keeping data private.

If you want, I can now:

- Produce a concrete code scaffold (single-file runnable FastAPI demo + Streamlit UI), or
- Output the SQLAlchemy models + FastAPI endpoints skeleton you can paste into a repo.

Pick which scaffold you want and I’ll generate it right away.