Brief Project Report: Document Research & Theme Identification Chatbot

1. Introduction

1.1 Problem Statement

Organizations and researchers often need to analyze large collections of documents (e.g., legal cases, technical reports, policy papers) to extract insights, answer specific questions, and identify overarching themes. Manual review is time-consuming and error-prone. While Retrieval-Augmented Generation (RAG) approaches provide baseline document question-answering, this internship project aims to go deeper:

- Extract precise answers from each document with accurate citations (page, paragraph or sentence).
- Identify and synthesize common themes across all documents per query.
- Offer a user-friendly web interface for upload, query, and navigation of results.

1.2 Objectives

- Enable ingestion of 75+ heterogeneous documents (PDFs, scanned images, text).
- Perform robust text extraction (including OCR) and store content for fast semantic search.
- Support natural-language queries: retrieve, extract, and cite answers per document.
- Cluster and synthesize themes across documents, presenting concise summaries with citations.
- Deliver a full-stack solution: FastAPI backend, React frontend, vector store (Qdrant), LLM integration.

2. Approach & System Overview

2.1 High-Level Architecture

- **Frontend**: React (TypeScript) application with components for document upload, document list & filtering, query input, answer table, theme panel, and document viewer modal.
- **Backend**: FastAPI service exposing endpoints for upload, listing/deletion, query, and theme identification.
- **Database**: PostgreSQL (or SQLite for development) storing document metadata and chunk metadata (page, paragraph index, chunk text, deterministic UUID for mapping to vector store).
- **Vector Store**: Qdrant for semantic search: embeddings of text chunks stored with payload metadata (doc_id, page_num, paragraph_index, optionally sentence offsets or chunk text).
- **OCR Engine**: Tesseract via pytesseract for scanned images or PDF pages lacking native text.

- **LLM Integration**: Abstracted in llm_clients.py, supporting a chosen backend (e.g., OpenAI or Google Gemini) for:
  - **Embedding**: compute embedding vectors for chunks and queries.
  - **Answer Extraction**: given a question and a document chunk, extract a concise snippet and generate a JSON response with text and citation.
  - **Theme Summarization**: given clusters of snippet excerpts, generate theme labels, summarized descriptions, and collect citations.
- **Theme Identification**: Embedding-based clustering of extracted snippet texts (e.g., Agglomerative clustering with heuristic number of clusters), followed by LLM-driven labeling and summary per cluster.
- **Storage Layer**: File system for raw uploads (organized under DATA_DIR/<doc_id>/filename), database for metadata, vector store for embeddings.
- **Services**:
  - **Ingestion Service**: extract text (pdfplumber/Tesseract), chunking service splits text into chunks.
  - **Embedding Index Service**: compute embeddings and upsert to vector store.
  - **Retrieval Service**: semantic search and fetch chunk texts.
  - **LLM Client Service**: abstraction over OpenAI or Google Gemini APIs for embeddings, extraction, and summarization.
  - **Theme Identification Service**: cluster snippets and orchestrate theme generation.

2.2 Data Flow

1. **Upload**: User uploads files → backend saves files → ingestion service extracts text per page (native or OCR) → chunking service splits text into overlapping chunks → store chunk metadata in DB and embeddings in vector store.
2. **Query**: User submits question → backend retrieves list of documents → for each document: vector search in Qdrant returns top-K chunk IDs → fetch chunk texts from DB → call LLM to extract snippet with citation → collect per-document answers.
3. **Theme Identification**: Collect all snippets → compute embeddings of snippet texts → cluster embeddings into themes → for each cluster: call LLM to generate theme name, summary, and collect citations → return to frontend.
4. **Frontend Display**: Show answer table and theme panel; citations clickable to open context via additional endpoint (e.g., GET /docs/{doc_id}/content?page=X&para=Y).

3. Technology Choices

Layer	**Technology**	**Rationale**
Backend Framework	**FastAPI**	Async support, automatic OpenAPI docs, lightweight
Database	**PostgreSQL** + SQLAlchemy	ACID compliance, JSON fields for metadata, ORM productivity
Vector Store	**Qdrant**	Native metadata filtering, Python client, cosine similarity
OCR	**Tesseract**	Mature, open-source, easy integration via pytesseract
Embeddings & LLM	**OpenAI GPT** or **Google Gemini**	Reliable embeddings and generation APIs
Frontend	**React** + **Tailwind CSS**	Component model, responsive UI, rapid styling
Deployment	Docker + Render/Vercel	Container portability, free hosting tiers
Testing	**pytest** + **pytest-asyncio**	Unit & async tests, in-memory SQLite for isolation

4. Methodology & Key Design Decisions

1. **Chunk-Based Retrieval**
    - Splitting documents into moderate-sized, overlapping chunks ensures that each LLM prompt stays within token limits while preserving context across paragraph boundaries.
2. **Per-Document Answer Extraction**
    - Extract answers on a per-document basis (rather than globally) to provide transparent citations back to the source, enhancing trust and auditability.
3. **Clustering-Driven Theme Detection**
    - Compute embeddings of extracted snippets, cluster them (e.g., Agglomerative clustering, 1–4 clusters based on snippet count) to group semantically similar answers, and then synthesize themes per cluster. This reduces prompt complexity and yields coherent themes.
4. **Structured Prompts & JSON Parsing**
    - Prompts explicitly request JSON output (fields: answer, citation; or theme_name, summary, citations). Post-processing with regex extracts JSON object even if extraneous text is included.
5. **Async Concurrency**

    - Use Python's asyncio for parallel LLM calls when extracting answers from multiple chunks across documents, improving throughput while respecting rate limits via controlled concurrency.

6. **Modular, Layered Architecture**
    - Clear separation of concerns (ingestion, embedding index, retrieval, LLM client, theme identification) facilitates testing, debugging, and future extensions (e.g., alternative vector stores or LLMs).

7. **Error Handling & Fallbacks**
    - OCR failures on certain pages: log warnings and skip empty content.
    - LLM extraction failures or malformed JSON: fallback to NO_ANSWER for that chunk.
    - Theme summarization failures: fallback to minimal theme entries listing citations without summary.

5. Testing Strategy

- **Unit Tests**
    - Test chunking logic (chunk_text_into_paragraphs) and deterministic UUID generation.
    - Validate text extraction on small sample inputs (mocking OCR where needed).
- **Service Tests (Mocks)**
    - Monkeypatch embedding and LLM client methods to return deterministic vectors or JSON responses; test clustering logic and parsing routines.
    - Mock Qdrant client to test indexing and retrieval service behaviors.
- **Integration Tests**
    - Use FastAPI's TestClient with an in-memory SQLite DB and monkeypatched vector store & LLM clients.
    - Test endpoints: upload small text files, query with a dummy question, verify response structures; test theme endpoint with controlled snippet data.

6. Implementation Highlights

- **Ingestion Service** (app.services.ingestion):
    - PDF parsing with pdfplumber; detect missing text and run Tesseract OCR on rendered page images.
    - Chunking into overlapping windows based on word count.
    - Store chunk metadata in ChunkORM with a deterministic UUID for vector store mapping.
- **Embedding Index Service** (app.services.embedding_index):

    - Compute embeddings via LLM client and upsert into Qdrant, tagging each vector with payload metadata (doc_id, page_num, paragraph_index).
    - Ensure consistent vector dimension and flatten nested structures.
- **Retrieval Service** (app.services.retrieval):
    - Given a query and document ID, compute query embedding, perform Qdrant search with metadata filter, fetch matching ChunkORM rows by UUID.
- **LLM Client Service** (app.services.llm_clients):
    - Abstract over chosen backend (OpenAI GPT or Google Gemini). Functions include get_embedding_vector, extract_answer_from_chunk, and generate_theme_summary. Prompts request strict JSON output.
- **Theme Identification** (app.services.theme_identification):
    - Embed snippet texts, cluster embeddings into 1–4 clusters, then for each cluster call LLM to produce theme_name, summary, and list of citations.
- **FastAPI Endpoints** (app.api.v1):
    - **Upload**: validate files, generate unique doc_id, save files under DATA_DIR/<doc_id>/, create DocumentORM, run ingestion and indexing.
    - **List/Delete Documents**: support filtering by author, type, date; deletion removes DB row, file folder, and Qdrant embeddings.
    - **Query**: for selected or all documents, retrieve top-K chunks, extract answers concurrently, return per-document snippets.
    - **Theme**: collect all snippets across documents, run cluster+summarization, return theme outputs.
- **Frontend** (React + Tailwind):
    - Components: DocumentUploader, DocumentList, QueryInput, AnswerTable, ThemeSummary, DocumentViewerModal.
    - State management: React state; API calls to backend endpoints.
    - UX: Show progress spinners during upload/indexing and query processing; modals to view full context of citations.

7. Challenges & Solutions

- **OCR Accuracy & Performance**: Some scanned pages yield noisy OCR. We combine native PDF text when available; for pure images, accept limitations and log failures. Preprocess images (grayscale, thresholding) if needed.
- **Token Limits**: Large chunks may exceed LLM context. Chunk size chosen (~300 words) to fit common model limits. For extremely long paragraphs, consider further splitting or summarization in future improvements.
- **LLM Response Parsing**: Models may include extra text around JSON. We use regex to extract the JSON object and fallback to safe defaults on parsing errors.

- **Rate Limits & Latency**: Parallel asyncio calls for embedding and extraction; control concurrency (e.g., semaphores) if needed. For large datasets (75+ docs), ingestion runs in synchronous flow; consider background tasks for production scale.
- **Vector Store Mapping**: Mapping Qdrant point IDs back to DB chunk records requires storing a deterministic UUID in both. We generate UUID based on doc_id, page_num, paragraph_index for consistency.
- **Testing External Dependencies**: Mocking QdrantClient and LLM client functions allows deterministic unit tests; integration tests use in-memory DB and stub clients.

8. Future Enhancements

- **Sentence-Level Citations**: Extend chunk metadata to include sentence offsets; refine extraction prompt to return sentence-level location.
- **Advanced Filtering & Metadata**: Allow users to tag documents with authors, categories, dates; filter the query scope in UI.
- **Background Workers & Queues**: Offload ingestion and heavy LLM tasks to Celery or RQ for scalability; provide status tracking of ingestion jobs.
- **Caching & Reuse**: Cache embeddings and LLM responses for repeated queries; implement debounce or result-store to reduce API costs.
- **User Authentication & Multi-Tenant**: Secure API with JWT auth; isolate user data and documents.
- **Visualization**: Graph-based view linking themes and documents; timeline or heatmap of citation frequency.
- **Alternative Models**: Integrate open-source embedding models (e.g., SentenceTransformers) and local LLMs for cost efficiency.
- **Feedback Loop**: Collect user feedback on snippet relevance; refine prompts or retrain domain-specific models in future.
- **Monitoring & Logging**: Integrate observability (Prometheus, Grafana), error tracking (Sentry), and cost monitoring for LLM usage.

9. Conclusion

This project implements a robust, explainable pipeline for deep document research: precise snippet extraction per document and clustering-driven thematic synthesis, exposed via an intuitive web UI. By combining open-source tools (Tesseract, Qdrant, FastAPI, React) with advanced generative AI backends (OpenAI GPT or Google Gemini), we deliver a scalable solution that balances granular transparency with high-level insights—meeting the rigorous requirements of the internship qualification task.