RAGWithGraphStore - Complete Data Flow & Architecture

> This document explains how data flows through the system, which databases store what, and answers key architectural questions.

Table of Contents

---

1. Database Overview - What Goes Where

```
+-----------------+-----------------------------------------+---------------------
|    DATABASE     |         WHAT IT STORES                  |         WHY THIS DB?
+-----------------+-----------------------------------------+---------------------
|  Neo4j (Aura)   | Users, Documents, Chunks, Entities      | Graph relationships
|                 | + relationships (OWNS, CONTAINS, etc.)  | documents, chunks, a
+-----------------+-----------------------------------------+---------------------
|  Qdrant         | Vector embeddings of document chunks    | Fast similarity sear
|  (Docker)       | + payload metadata (text, user_id)      | finding relevant con
+-----------------+-----------------------------------------+---------------------
|  PostgreSQL     | LangGraph workflow checkpoints          | Persistent state for
|  (Docker)       | (comparison workflow state)              | LangGraph workflows
+-----------------+-----------------------------------------+---------------------
|  Redis          | JWT token blocklist (logout)            | Fast key-value looku
|  (Docker)       | + refresh token hashes                  | auth token managemen
+-----------------+-----------------------------------------+---------------------
|  Mem0 SDK       | User memories, conversation facts,      | Unified memory layer
|  (uses Qdrant   | shared company knowledge                | Qdrant "memory" coll
|   + Neo4j)      |                                         | Neo4j for entity gra
+-----------------+-----------------------------------------+---------------------
```

2. How The System Knows Which Chat Belongs to Which User & Document

User Identification

Every API request goes through authentication middleware that produces a `UserContext`:

```
Request comes in
    |
    v
get_current_user_optional()
    |
    +---> Has valid JWT token?
    |         YES --> UserContext(id="c7b17fae-...", email="user@email.com", is_an
    |
    +---> No token?
              YES --> UserContext(id="anon_abc123xyz", is_anonymous=True)
                      (anonymous session ID stored in HTTP-only cookie)
```

Multi-Tenant Isolation (How user_id flows everywhere)

```
User (id: "user-123") uploads a document
    |
    v
Document created in Neo4j:      (User {id:"user-123"}) -[:OWNS]-> (Document {id:"d
    |
    v
Chunks created in Neo4j:        (Document {id:"doc-1"}) -[:CONTAINS]-> (Chunk {id:
    |
    v
Vectors stored in Qdrant:       Point { vector: [0.1, 0.2, ...], payload: { user_i
    |
    v
When user-123 asks a question:  Qdrant filter = { user_id == "user-123" }  <-- Onl
When user-456 asks a question:  Qdrant filter = { user_id == "user-456" }  <-- See
```

Key Point: `user_id` is stored in EVERY layer

Layer	Where user_id is stored	How it filters
Neo4j	`Document.user_id` + `(User)-[:OWNS]->(Document)` relationship	Cypher WHERE clause
Qdrant	`payload.user_id` on every vector point	FieldCondition filter
Mem0	`user_id` parameter on every add/search call	Internal filtering

Mem0	`user_id` parameter on every add/search call	Internal filtering
PostgreSQL	Thread ID format: `{user_id}:doc_compare:{session_id}`	Thread ID prefix
Redis	Key format: `refresh:{user_id}:{jti}`	Key pattern match

---

3. Where Chat History Is Stored

> **There is NO traditional `chat_history` table.** The system uses **Mem0** for conversation tracking.

How it works:

```
User sends message: "What is the revenue?"
    |
    v
1. System processes the query and generates an answer
    |
    v
2. Both the question and answer are stored via Mem0:

   mem0.add(
       messages = "User asked: What is the revenue? Assistant answered: Based on..
       user_id = "user-123",
       metadata = {
           "type": "conversation",
           "session_id": "session-abc",
           "role": "user",
           "timestamp": "2024-01-15T10:30:00Z"
       }
   )
```

Where Mem0 actually stores this:

```
Mem0 SDK
    |
    +---> Qdrant "memory" collection (SEPARATE from "documents" collection)
    |     - Stores semantic embeddings of conversation facts
    |     - Enables: "Find memories related to revenue questions"
    |
    +---> Neo4j (entity graph)
          - Stores entity relationships extracted from conversations
          - Enables: "User talked about Company X which relates to Topic Y"
```

Retrieving chat history:

```
# Get conversation history for a session
memories = mem0.get_all(user_id="user-123", limit=100)
# Filter by session_id and type="conversation"
# Sort by timestamp
# Return last N turns
```

---

4. Neo4j - Chunk-to-User Ownership

Yes! Neo4j knows exactly which chunk belongs to which user through a chain of relationships:

```
                    OWNS                          CONTAINS
(User {id: "user-123"}) -------> (Document {id: "doc-1"}) -------> (Chunk {id: "ch
                                  user_id: "user-123"               document_id: "
                                  filename: "report.pdf"            text: "The rev
                                  chunk_count: 15                   position: 0
```

Neo4j Graph Schema:

```
NODE TYPES:
===========

(:User)
  - id: UUID (unique)
  - email: string
  - hashed_password: string (Argon2)
  - role: "user" | "admin"
  - created_at: datetime

(:Document)
  - id: UUID (unique)
  - filename: string
  - user_id: string (owner's ID, or "__shared__" for shared docs)
  - upload_date: datetime
  - chunk_count: integer
  - summary: string (LLM-generated)
  - file_type: string (pdf, docx)
  - file_size: integer (bytes)

(:Chunk)
  - id: UUID (unique, SAME ID used in Qdrant)
  - document_id: string
  - text: string (chunk content)
  - position: integer (0-based index in document)
  - embedding_id: string (same as id)
```

```
(:Entity)  [Infrastructure ready, not fully used yet]
  - name: string
  - type: string (Person, Organization, etc.)


RELATIONSHIPS:
==============

(User) -[:OWNS]-> (Document)
    "This user uploaded/owns this document"

(Document) -[:CONTAINS]-> (Chunk)
    "This document has these chunks"

(Entity) -[:APPEARS_IN]-> (Chunk)        [Future - pending NER integration]
    "This entity is mentioned in this chunk"

(Entity) -[:RELATES_TO]-> (Entity)       [Future - pending NER integration]
    "These entities are related"
```

Example Cypher queries used:

```
-- Get all documents for a user
MATCH (u:User {id: $user_id})-[:OWNS]->(d:Document)
RETURN d ORDER BY d.upload_date DESC

-- Get chunk's parent document
MATCH (c:Chunk {id: $chunk_id})<-[:CONTAINS]-(d:Document)
RETURN d.filename, d.id

-- Delete a document and all its chunks
MATCH (u:User {id: $user_id})-[:OWNS]->(d:Document {id: $doc_id})
OPTIONAL MATCH (d)-[:CONTAINS]->(c:Chunk)
DETACH DELETE d, c
```

---

5. How Shared Memory Works

The Concept

Shared memory = **company-wide knowledge** that ALL authenticated users can access when asking questions.

How it's stored

Shared data uses a special sentinel user_id: `"__shared__"`

```
Admin uploads shared document:
    Document.user_id = "__shared__"
    Qdrant payload.user_id = "__shared__"

Admin adds shared memory fact:
    Mem0.add(content, user_id="__shared__")
```

How it's queried

When an authenticated user asks a question:

```
Step 1: Search user's OWN documents
        Qdrant filter: user_id == "user-123"

Step 2: ALSO search shared documents
        Qdrant filter: user_id == "__shared__"

Combined filter (actual implementation):
        Qdrant filter: user_id IN ["user-123", "__shared__"]
```

```
# From qdrant_client.py
if include_shared and user_id != "__shared__":
    user_filter = Filter(
        must=[
            FieldCondition(
                key="user_id",
                match=MatchAny(any=[user_id, settings.SHARED_MEMORY_USER_ID]),  #
            )
        ]
    )
```

Same pattern for Mem0 memories:

```
# Search user's private memories
user_results = memory.search(query=query, user_id="user-123")

# ALSO search shared company memory
shared_results = memory.search(query=query, user_id="__shared__")

# Combine both, mark shared results
for mem in shared_results:
    mem["is_shared"] = True
```

Who can do what with shared memory:

Action	Admin	Authenticated User	Anonymous

Add shared document	Yes	No	No
Add shared memory fact	Yes	No	No
Read shared documents in queries	Yes	Yes	No
Read shared memory in queries	Yes	Yes	No

Endpoints:

- `POST /api/v1/admin/documents/upload` - Admin uploads shared document
- `POST /api/v1/memory/shared` - Admin adds shared memory fact
- `POST /api/v1/memory/search` - Search includes shared memory (for authenticated users)

---

6. How Document Comparison Works

Does it use a package or the LLM?

The LLM does the comparison. There is no diffing package. The system uses a **LangGraph workflow** with 4 steps that retrieves relevant chunks from each document and asks the LLM to analyze similarities and differences.

Complete Comparison Flow:

```
User selects 2+ documents + writes comparison query
    |
    v
POST /api/v1/compare/
    |
    v
+-------------------------------------------------------------------+
|  LANGGRAPH WORKFLOW (4 nodes, linear flow)                        |
|                                                                   |
|  Step 1: RETRIEVE                                                 |
|  +-------------------------------------------------------+    |
|  | For each document:                                      |    |
|  |   1. Generate embedding of the comparison query         |    |
|  |   2. Vector search in Qdrant filtered by document_id    |    |
|  |   3. Get top 5 most relevant chunks per document        |    |
|  |   4. Enrich with Neo4j metadata (filename, etc.)        |    |
|  +-------------------------------------------------------+    |
|                           |                                       |
|                           v                                       |
|  Step 2: EXPAND GRAPH                                             |
|  +-------------------------------------------------------+    |
```

```
|  | For each retrieved chunk:                                    |    |
|  |   1. Query Neo4j for entity relationships                    |    |
|  |      (Entity)-[:RELATES_TO]-(Entity)-[:APPEARS_IN]->(Chunk)  |
|  |   2. Find cross-document connections via entities            |    |
|  |   3. Collect related chunks from other documents             |    |
|  |   Fallback: If no entities, find sibling chunks              |    |
|  +--------------------------------------------------------+    |
|                          |                                      |
|                          v                                      |
|  Step 3: COMPARE (LLM Analysis)                                 |
|  +--------------------------------------------------------+    |
|  | 1. Format all chunks + graph context into a prompt:    |    |
|  |                                                        |    |
|  |    === Document: report.pdf ===                        |    |
|  |    Key Content:                                        |    |
|  |    [Chunk 1] First 500 chars of text...                |    |
|  |    [Chunk 2] First 500 chars of text...                |    |
|  |    Entity Relationships:                               |    |
|  |    - Company A --[PARTNERS_WITH]--> Company B          |    |
|  |                                                        |    |
|  |    === Document: design.pdf ===                        |    |
|  |    Key Content:                                        |    |
|  |    [Chunk 1] First 500 chars of text...                |    |
|  |                                                        |    |
|  | 2. Ask LLM to return JSON:                             |    |
|  |    {                                                   |    |
|  |      "similarities": ["point 1", "point 2"],           |    |
|  |      "differences": ["point 1", "point 2"],            |    |
|  |      "insights": ["insight 1", "insight 2"]            |    |
|  |    }                                                   |    |
|  |                                                        |    |
|  | 3. Parse JSON (fallback: extract from text if no JSON) |    |
|  +--------------------------------------------------------+    |
|                          |                                      |
|                          v                                      |
|  Step 4: GENERATE RESPONSE                                      |
|  +--------------------------------------------------------+    |
|  | 1. Assemble markdown response from analysis            |    |
|  | 2. Extract citations from retrieved chunks             |    |
|  | 3. Return formatted comparison report                  |    |
|  +--------------------------------------------------------+    |
+-----------------------------------------------------------------+
    |
    v
Response:
{
    "similarities": ["Both documents discuss...", ...],
    "differences": ["Document 1 focuses on...", ...],
    "cross_document_insights": ["A connection between...", ...],
    "response": "## Document Comparison Analysis\n...",
    "citations": [{document_id, chunk_id, text, filename}, ...],
    "session_id": "uuid",
    "status": "complete"
```

```
}
```

Key Details:

- **LLM**: Uses `get_llm(temperature=0.3)` - slightly creative but grounded
- **Model**: Whatever is configured (Ollama llama3.2 or OpenAI)
- **State Persistence**: PostgreSQL checkpointer saves state after each step
- **Thread ID**: `{user_id}:doc_compare:{session_id}` (prevents cross-user state leakage)
- **Timeout**: Can take 1-2 minutes with Ollama (local LLM is slow)

---

7. How Different Data Is Stored Across Different Databases

Neo4j (Graph Database) - Relationships & Metadata

```
WHAT:
  - User accounts (id, email, hashed password, role)
  - Document metadata (filename, upload date, chunk count, summary)
  - Chunk metadata (text content, position in document)
  - Entity nodes (planned: person names, organizations, concepts)
  - ALL relationships between these nodes

WHY NEO4J:
  - Graph traversals: "Find all chunks from documents owned by user X"
  - Multi-hop queries: "Find entities in doc A that relate to entities in doc B"
  - Relationship-first data model: OWNS, CONTAINS, APPEARS_IN, RELATES_TO

EXAMPLES:
  - "Get all documents for user-123" -> MATCH (u:User {id: $id})-[:OWNS]->(d:Docum
  - "Get filename for chunk" -> MATCH (c:Chunk {id: $id})<-[:CONTAINS]-(d:Document
  - "Find related chunks via entities" -> Multi-hop traversal query
```

Qdrant (Vector Database) - Embeddings & Similarity Search

```
WHAT:
  TWO COLLECTIONS:

  1. "documents" collection:
     - Vector: Embedding of chunk text (768d for Ollama, 1536d for OpenAI)
     - Payload: { text, document_id, user_id, position }
     - One vector per chunk

  2. "memory" collection (managed by Mem0):
     - Vector: Embedding of memory/conversation fact
     - Payload: { memory text, user_id, metadata }
     - One vector per memory entry
```

```
  - One vector per memory entry

WHY QDRANT:
  - Semantic similarity search: "Find chunks most relevant to this question"
  - Fast approximate nearest neighbor (ANN) search
  - Filtering by user_id and document_id at query time
  - Cosine distance metric for measuring relevance

EXAMPLES:
  - User asks "What is revenue?" -> Generate query embedding -> Find top 3 similar
  - Document comparison -> Filter by specific document_ids + user_id
```

PostgreSQL - LangGraph Workflow State

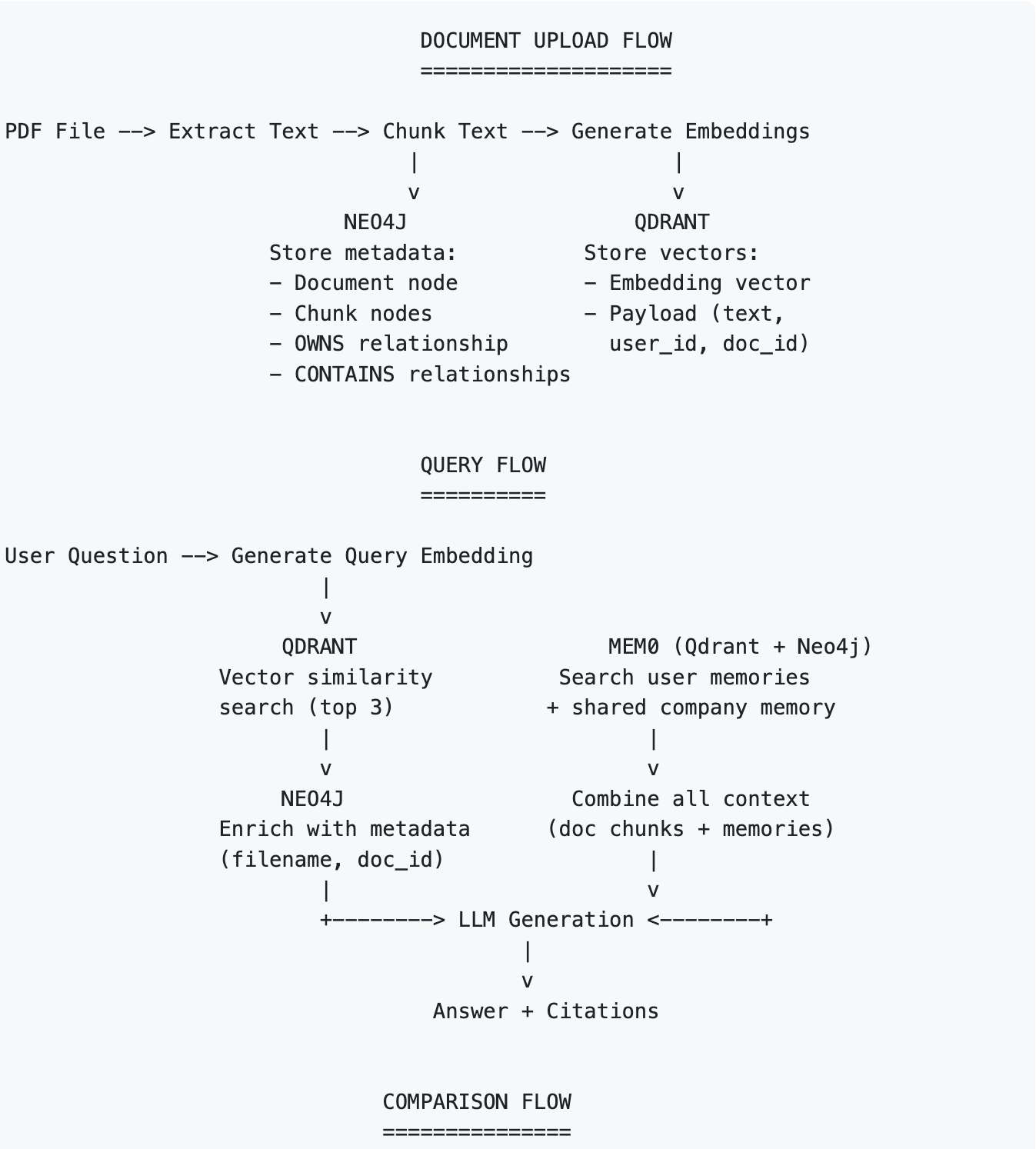
```
WHAT:
  4 TABLES (all for LangGraph checkpointing):

  - checkpoints: Workflow state snapshots (JSONB)
  - checkpoint_blobs: Serialized binary state data
  - checkpoint_writes: Pending write operations
  - checkpoint_migrations: Schema version tracking

WHY POSTGRESQL:
  - Durable state persistence for multi-step workflows
  - Required by LangGraph's AsyncPostgresSaver
  - Enables resuming interrupted comparison workflows
  - Transaction support for state consistency

WHAT IT DOES NOT STORE:
  - No user data
  - No document data
  - No chat history
  - Only workflow execution state
```

Redis - Auth Token Management

```
WHAT:
  TWO KEY PATTERNS:

  1. Token Blocklist:
     Key:   "blocklist:{jti}"
     Value: "1"
     TTL:   7 days
     Purpose: Invalidate tokens on logout

  2. Refresh Token Hashes:
     Key:   "refresh:{user_id}:{jti}"
     Value: SHA-256 hash of refresh token
     TTL:   7 days
     Purpose: Single-use refresh token enforcement
```

```
WHY REDIS:
  - Sub-millisecond lookups for token validation
  - TTL-based auto-expiry (no cleanup jobs needed)
  - In-memory speed for high-frequency auth checks

WHAT IT DOES NOT STORE:
  - No user profiles (those are in Neo4j)
  - No session data
  - No chat history
  - Only token metadata
```

Visual Summary - Data Flow Across All DBs



```
Select 2+ docs + query
         |
         v
      QDRANT                    NEO4J                    LLM
  Vector search per doc    Graph traversal for      Analyze similarities
  (filtered by doc_id)    entity relationships      and differences
         |                        |                        |
         v                        v                        v
    Retrieved chunks  +  Graph context  -->  Formatted prompt  -->  JSON analysis
                                                                          |
                                                                          v
                                                                     POSTGRESQL
                                                                  Save workflow state
                                                                  (checkpointing)
```

---

8. RAG Pipeline - How Relevant Context Is Retrieved

When a user asks a question, here's the complete step-by-step pipeline:

Step 1: Query Embedding

```
User Question: "What is the company's revenue strategy?"
    |
    v
embedding_service.generate_query_embedding(query)
    |
    v
Uses configured embedding model:
  - Ollama: nomic-embed-text (768 dimensions)
  - OpenAI: text-embedding-3-small (1536 dimensions)
    |
    v
Result: [0.023, -0.145, 0.891, ...] (768 or 1536 floats)
```

Step 2: Vector Search in Qdrant

```
Query Vector: [0.023, -0.145, 0.891, ...]
    |
    v
qdrant_client.query_points(
    collection = "documents",
    query = query_vector,
    filter = {
        user_id IN ["user-123", "__shared__"]    <-- User's docs + shared docs
    },
    limit = 3,                                    <-- Top 3 most relevant chunks
```

```
    with_payload = True                              <-- Include text content
)
    |
    v
Results (sorted by cosine similarity):
  1. score=0.92  chunk from "report.pdf"     text: "Revenue grew 25% in Q3..."
  2. score=0.88  chunk from "strategy.pdf"   text: "The revenue strategy focuses..
  3. score=0.85  chunk from "report.pdf"     text: "Key financial metrics show..."
```

Step 3: Enrich with Neo4j Metadata

```
For each chunk, query Neo4j:
    MATCH (c:Chunk {id: $chunk_id})<-[:CONTAINS]-(d:Document)
    RETURN d.filename, d.id
    |
    v
Enriched chunks:
  1. { text: "Revenue grew...", filename: "report.pdf", document_id: "doc-1", scor
  2. { text: "The revenue...", filename: "strategy.pdf", document_id: "doc-2", sco
  3. { text: "Key financial...", filename: "report.pdf", document_id: "doc-1", sco
```

Step 4: Retrieve Memory Context (Authenticated Users Only)

```
If user is authenticated:
    |
    v
  Mem0 search user's private memories:
    memory.search(query="revenue strategy", user_id="user-123")
    --> "User previously discussed Q3 earnings projections"
    |
    v
  Mem0 search shared company memory:
    memory.search(query="revenue strategy", user_id="__shared__")
    --> "Company targets 20% YoY revenue growth"
    |
    v
  Combine as additional context chunks:
    { text: "User previously discussed...", filename: "User Memory", score: 0.5 }
    { text: "Company targets 20%...", filename: "Shared Memory", score: 0.5, is_sh
```

Step 5: Generate Answer with LLM

```
All context combined:
    [Source: report.pdf]
    Revenue grew 25% in Q3...

    [Source: strategy.pdf]
```

```
    The revenue strategy focuses...

    [Source: report.pdf]
    Key financial metrics show...

    [Source: Shared Memory]
    Company targets 20% YoY revenue growth
    |
    v
LLM Prompt:
    System: "Answer questions ONLY based on provided context.
             If context doesn't contain the answer, say 'I don't know.'
             Cite the source document when referencing information."

    User: "Context: {above context}
           Question: What is the company's revenue strategy?
           Answer:"
    |
    v
LLM (Ollama llama3.2 or OpenAI GPT-4):
    "Based on the documents, the company's revenue strategy focuses on
     achieving 25% quarterly growth (report.pdf). The strategy specifically
     targets 20% year-over-year growth (Shared Memory), with key financial
     metrics showing positive trajectory (report.pdf)."
```

Step 6: Build Citations

```
Citations built from the retrieved chunks:
[
  { document_id: "doc-1", filename: "report.pdf", chunk_text: "Revenue grew 25%...
  { document_id: "doc-2", filename: "strategy.pdf", chunk_text: "The revenue strat
  { document_id: "doc-1", filename: "report.pdf", chunk_text: "Key financial...",
]
```

Step 7: Confidence Scoring (Enhanced Endpoint Only)

```
For OpenAI:
    Extract token-level log probabilities from LLM response
    confidence = geometric_mean(exp(logprobs))
    --> score: 0.87, level: "high"

For Ollama (no logprobs):
    Ask LLM: "Rate how well this answer is supported by the context (0-100)"
    LLM responds: "82"
    --> score: 0.82, level: "medium"
```

Step 8: Streaming (SSE) Response to Frontend

```
Server-Sent Events sequence:

Event 1: { event: "status", data: "retrieving" }        <-- Show "Searching..."
Event 2: { event: "citations", data: [citation1, ...] }  <-- Show source cards
Event 3: { event: "status", data: "generating" }         <-- Show "Generating..."
Event 4: { event: "token", data: "Based" }               <-- Stream word by word
Event 5: { event: "token", data: " on" }
Event 6: { event: "token", data: " the" }
...
Event N: { event: "done", data: "" }                      <-- Complete
```

Complete Pipeline Diagram

```
  User: "What is the revenue strategy?"
         |
         v
  [1] EMBED QUERY -----> Embedding Model (Ollama/OpenAI)
         |                      |
         v                      v
  [2] VECTOR SEARCH <---- Query Vector [0.023, -0.145, ...]
         |
         v
  [3] QDRANT ------> Top 3 chunks (filtered by user_id)
         |
         v
  [4] NEO4J -------> Enrich with filenames & metadata
         |
         v
  [5] MEM0 --------> User memories + shared company knowledge
         |
         v
  [6] COMBINE -----> Document chunks + Memory chunks = Full Context
         |
         v
  [7] LLM ---------> Generate answer with strict citation rules
         |
         v
  [8] CONFIDENCE ---> Score via logprobs (OpenAI) or self-assessment (Ollama)
         |
         v
  [9] STREAM ------> SSE events: status -> citations -> tokens -> done
         |
         v
  Frontend displays answer with citations and confidence badge
```

Summary: Quick Reference Table

Question	Answer
Which chat belongs to which user?	Every DB operation uses `user_id` parameter. Authenticated users get UUID, anonymous get `anon_xxx`
Where is chat history?	Mem0 SDK stores it in Qdrant "memory" collection + Neo4j entity graph
Does Neo4j know chunk ownership?	Yes: `(User)-[:OWNS]->(Document)-[:CONTAINS]->(Chunk)` + `user_id` on Document node
How does shared memory work?	Uses sentinel `user_id = "__shared__"`. Queries include both user's data AND shared data
How does comparison work?	LangGraph 4-step workflow: Retrieve chunks -> Graph expand -> LLM analyze -> Format response. **Uses LLM, not a diff package**
What's in each database?	Neo4j=metadata+relationships, Qdrant=vectors, PostgreSQL=workflow state, Redis=auth tokens
How is relevant context found?	Query is embedded -> Qdrant cosine similarity search -> Neo4j metadata enrichment -> Mem0 memory search -> Combined context to LLM

Key Files Reference

File	What it does
`app/db/neo4j_client.py`	Neo4j connection, schema initialization
`app/db/qdrant_client.py`	Qdrant connection, vector CRUD, similarity search
`app/db/mem0_client.py`	Mem0 SDK configuration (uses Qdrant + Neo4j)
`app/db/redis_client.py`	Redis connection, token blocklist operations
`app/db/postgres_client.py`	PostgreSQL connection pool
`app/db/checkpoint_store.py`	LangGraph checkpoint table setup
`app/services/retrieval_service.py`	RAG retrieval pipeline (embed -> search -> enrich)
`app/services/generation_service.py`	LLM answer generation + streaming
`app/services/embedding_service.py`	Embedding generation via LangChain

`app/services/indexing_service.py`	Document processing: store in Neo4j + Qdrant
`app/services/graphrag_service.py`	Neo4j graph traversal for context expansion
`app/services/confidence_service.py`	Confidence scoring (logprobs or self-assessment)
`app/services/memory_service.py`	Memory CRUD (add/search/delete via Mem0)
`app/services/memory_summarizer.py`	Auto-compress old memories when threshold exceeded
`app/services/document_processor.py`	PDF/DOCX extraction, chunking, embedding pipeline
`app/workflows/document_comparison.py`	LangGraph comparison workflow orchestration
`app/workflows/nodes/retrieval.py`	Workflow node: retrieve + graph expand
`app/workflows/nodes/comparison.py`	Workflow node: LLM-powered analysis
`app/workflows/nodes/generation.py`	Workflow node: format response + citations
`app/api/queries.py`	Query endpoints (standard, stream, enhanced)
`app/api/comparisons.py`	Comparison endpoint
`app/api/documents.py`	Document upload/list/delete endpoints
`app/api/memory.py`	Memory endpoints (add/search/delete/shared)
`app/core/security.py`	Authentication middleware (JWT + anonymous sessions)
`app/core/auth.py`	JWT creation, token pairs, password hashing
`app/services/llm_provider.py`	LLM factory: get_llm(), get_embedding_model()