1. Project Overview

- **Project Name:** Cerebro AI
- **Project Description:** A multimodal intelligence system that acts as a foundational prototype for a digital "second brain". It ingests audio, documents, and web content to provide a conversational interface with perfect recall and temporal reasoning.
- **Stakeholders:** Individual users seeking personal knowledge management, system architects, and developers.
- **Assumptions:** A 48-hour development window, reliance on high-performance LLM APIs (OpenAI/Anthropic), and a single-user-first architecture that can scale horizontally later.
- **Use-cases:** Recalling specific meeting concerns from past audio transcripts.
    - Summarizing long-form technical articles (e.g., quantum computing).
    - Answering time-sensitive queries like "What did I work on last month?"

2. Requirements

Functional Requirements

- **Multimodal Ingestion:** The system must process audio (.mp3, .m4a), documents (.pdf, .md), scraped web content, and raw text notes.
- **Intelligent Retrieval:** Users must be able to query their data using natural language to receive synthesized, human-like answers.
- **Temporal Awareness:** All data must be searchable by time, supporting queries regarding specific dates or ranges.
- **Image Search:** Images must be stored and made searchable via associated metadata or text descriptions.

Non-Functional Requirements

- **Architectural Rigor:** Deep foresight in the design to ensure the system is more than a simple wrapper around an LLM.
- **Performance:** Responses must be fast and accurate, utilizing asynchronous processing for heavy ingestion tasks.
- **Privacy by Design:** The system must address the trade-offs between local-first and cloud-hosted data residency.
- **User Experience:** A clean, minimal interface featuring streamed responses for a "live" feel.

3. System Architecture

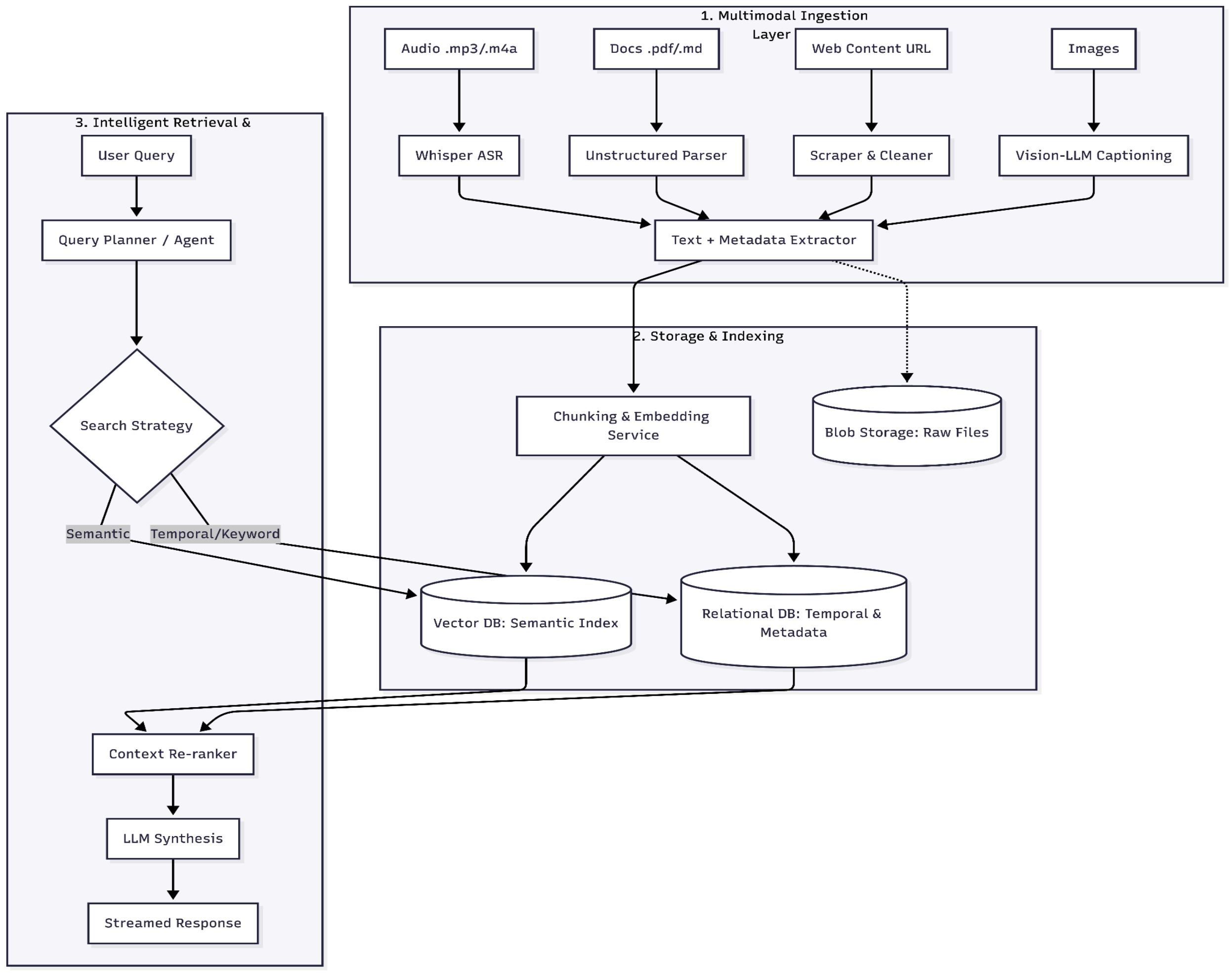
Technology Stack

- **Frontend:** Next.js for a clean, responsive chat interface with token streaming.
- **Backend:** FastAPI (Python) to handle asynchronous data processing.
- **Database:** PostgreSQL with `pgvector` for a hybrid relational and semantic storage model.
- **Task Queue:** Redis and Celery for the asynchronous processing of audio and heavy documents.

System Components

I have structured the system into three major services: the **Ingestion Worker**, the **Knowledge Index**, and the **Reasoning API**. The API handles user queries, while the worker processes incoming files in the background to ensure the UI remains responsive.

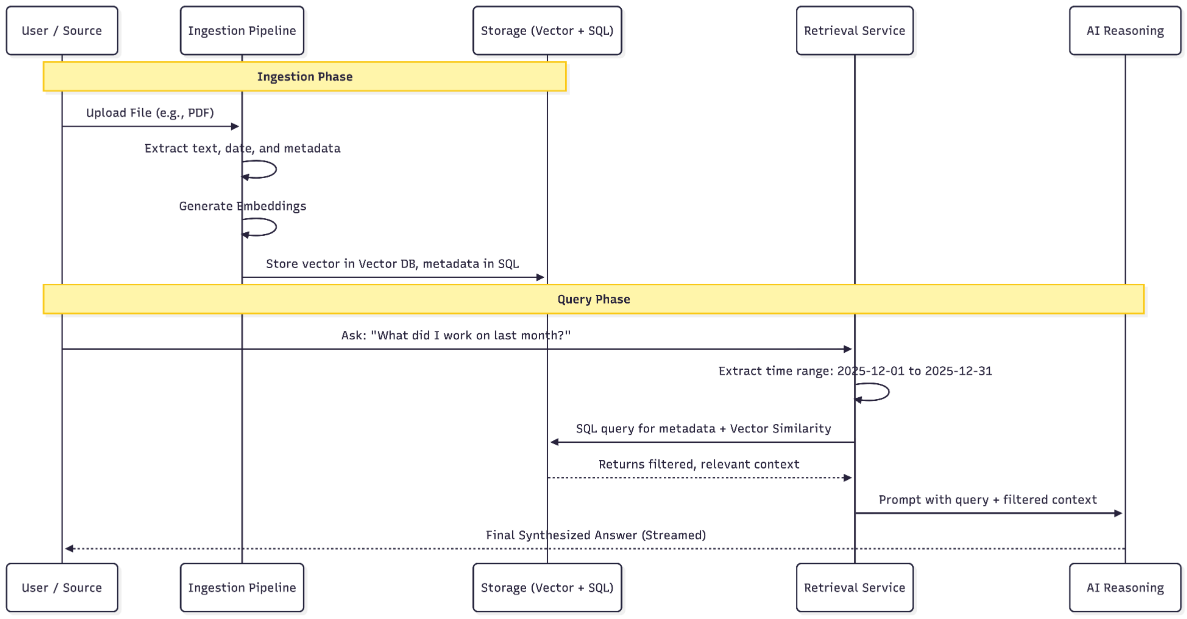
High-Level Diagram



4. Module Design

4.1. Multimodal Ingestion Module

- **Purpose:** To normalize diverse data types into searchable text chunks.
- **Inputs:** Binary files (audio, PDF), URLs, or raw text strings.
- **Outputs:** Standardized text segments, vector embeddings, and structured metadata.
- **Dependencies:** Whisper (Audio transcription), PDFMiner/Unstructured (Document parsing), and an Embedding Model (e.g., text-embedding-3-small).


User / Source
Ingestion Pipeline
Storage (Vector + SQL)
Retrieval Service
AI Reasoning
Ingestion Phase
Upload File (e.g., PDF)
Extract text, date, and metadata
Generate Embeddings
Store vector in Vector DB, metadata in SQL
Query Phase
Ask: "What did I work on last month?"
Extract time range: 2025-12-01 to 2025-12-31
SQL query for metadata + Vector Similarity
Returns filtered, relevant context
Prompt with query + filtered context
Final Synthesized Answer (Streamed)

4.2. Retrieval and Reasoning Module

Purpose The retrieval engine is the core intelligence of Cerebro. My goal here isn't just to find similar text, but to provide the system with the ability to navigate a user's life chronologically and contextually. I designed this module to act as a sophisticated filter that transforms a vague natural language question into a precise, grounded answer.

The Strategy: Multi-Stage Hybrid Retrieval I decided against relying solely on semantic search. While vector embeddings are excellent at capturing the "vibe" of a query, they often struggle with precision—failing to distinguish between two different project meetings or missing specific technical terms. To solve this, I am using a hybrid approach that combines semantic depth with keyword precision and strict temporal logic.

1. Intent Parsing and Temporal Grounding: When I receive a query, the first step is to determine if the user is asking about a specific timeframe. If a user asks, "What did I work on last month?", a standard vector search might return results about "work" from three years ago because they are semantically similar. To prevent this, I use a lightweight LLM call to extract temporal intent and convert relative phrases into absolute date ranges. This generates a metadata filter that I apply directly to the database query.

2. Execution of the Hybrid Search: I execute two search paths in parallel to ensure I don't miss anything:

- **Vector Search:** I use dense embeddings to find chunks that are conceptually related to the query. This handles the "reasoning" part of the search, identifying that a query about "budgeting" should look at spreadsheets and financial notes.
- **Keyword Search:** I run a traditional BM25 or full-text search. This is my safety net for proper nouns, specific project codes, or unique names that might not be well-represented in a general-purpose embedding space.

3. The Re-Ranking Layer: Merging results from two different search types can be messy because their scoring systems don't match. To fix this, I take the top results from both paths and pass them through a Cross-Encoder re-ranker. This model is more computationally expensive but significantly more accurate; it looks at the query and the candidate chunk together to determine a final relevance score. I then take the top ten most relevant chunks to use as context.

4. Synthesis with Grounding: The final stage is the synthesis. I pass the filtered context to a high-reasoning LLM with a specific instruction: "Answer only using the provided context." By forcing the model to cite the specific document and timestamp for every claim, I ensure that the response is verifiable. This effectively eliminates the risk of the AI making up memories and ensures the "Second Brain" remains a reliable source of truth.

Justification of Choices: I chose this specific architecture because it addresses the "time-blindness" inherent in most RAG systems. By integrating SQL-based temporal filtering with hybrid search, I can guarantee that Cerebro understands not just *what* happened, but *when* it happened. This level of rigor is what separates a simple document chatbot from a true personal companion.

5. Database Design

Schema Design

I chose a relational schema in PostgreSQL because it allows for powerful metadata filtering, which is essential for temporal queries.

- **Documents Table:** Stores `id`, `source_type`, `original_url`, and `owner_id`.
- **Chunks Table:** Stores `id`, `document_id`, `content_text`, `vector_embedding`, and `created_at`.
- **Metadata Table:** A key-value store linked to chunks for extensible attributes like "page number" or "speaker".

Storage Trade-offs

I chose **SQL (PostgreSQL + pgvector)** over a dedicated Vector DB or NoSQL. While a dedicated Vector DB scales well for pure similarity, it often struggles with the complex relational filtering (like time ranges) required by this project. PostgreSQL provides the best balance of scalability, cost, and query flexibility for a single-user "Second Brain".

6. API Design

- **POST /v1/ingest:** Accepts a file or URL. It returns a job ID and processes data asynchronously.
- **POST /v1/chat:** Accepts a text query. It executes the retrieval strategy and returns a streamed LLM response.
- **Authentication:** I will use JWT-based authentication to ensure private data remains accessible only to the owner.

7. Security and Privacy Design

I have adopted a **Privacy by Design** philosophy.

- **Data Encryption:** I will use TLS for data in transit and AES-256 for data at rest in the database.
- **Audio Data Minimization:** I have made the architectural decision to never record or store raw audio files. The ingestion pipeline processes spoken words into text in real-time. Once the transcription is complete, the original audio buffer is immediately purged from memory. This eliminates the risk of sensitive raw audio being leaked or subpoenaed.
- **Identity-Linked Data Isolation:** I use strict row-level security (RLS) in my PostgreSQL database to ensure that a user’s "memories" are logically and physically isolated from others, preventing any possibility of cross-context leakage.
- **Local-first Consideration:** I designed the ingestion and embedding modules to be "swappable." While I am using cloud APIs for this prototype, the architecture allows for local embedding models to be used in the future, enabling sensitive text to be kept on-device.

8. Deployment Architecture

I will deploy this system using **Docker Compose** to manage the API, worker, and database containers. For the demo, I will host the application on a cloud provider such as AWS or Fly.io to ensure a working URL for evaluation.

9. Testing Strategy

- **Unit Testing:** I will use Pytest to verify the accuracy of the document parsing and chunking logic.
- **Integration Testing:** I will test the end-to-end flow from file upload to successful retrieval.

- **Functional Correctness:** I will perform manual checks to ensure the LLM synthesis is grounded in the provided context and does not hallucinate.

10. Maintenance and Monitoring

I will implement basic logging using Python's logging module to track ingestion failures. I plan to monitor LLM API usage and costs to ensure the system remains sustainable during the testing phase.

11. Backup and Recovery

Since this is a prototype, I will rely on PostgreSQL's standard backup tools (pg_dump) to create periodic snapshots of the knowledge base.

12. Risks and Mitigation

- **Risk:** Audio transcription latency.
    - **Mitigation:** I use an asynchronous pipeline so the user doesn't have to wait for the transcription to finish before using other features.
- **Risk:** LLM context limits.
    - **Mitigation:** I will implement a re-ranking strategy to ensure only the most relevant chunks are sent to the LLM.

13. Future Enhancements

- **Graph-based Search:** In the future, I want to add a graph database layer to track relationships between entities (e.g., connecting a person mentioned in an email to a project mentioned in a meeting).
- **Full Local-First:** Transitioning to a local Llama 3 instance to provide 100% privacy for the user's "Second Brain".