System Design & Architecture

1.1 Multi-Modal Data Ingestion Pipeline

Core Principle:

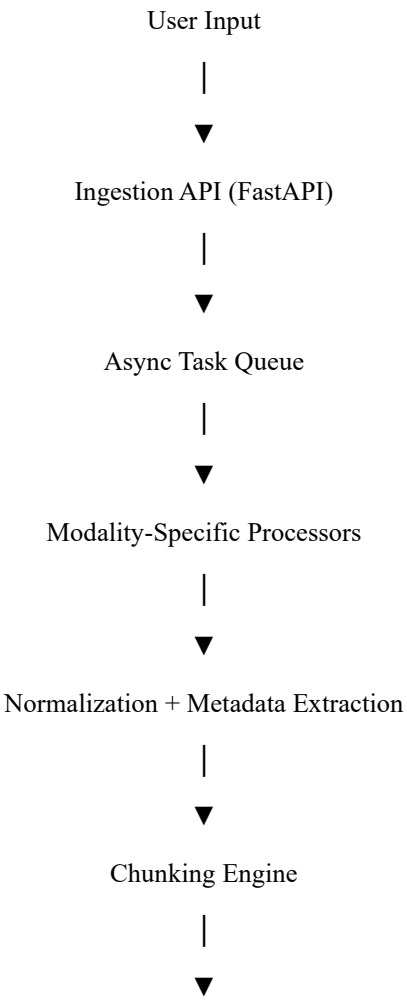
- All data, regardless of modality, is normalized into searchable text chunks with metadata and timestamps.
- This allows a single retrieval strategy across all inputs.

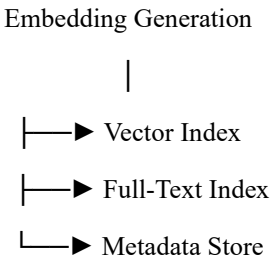
Key design principle:

Every modality ultimately becomes:

- **Normalized text** (or text representations like OCR/captions)
- **Metadata** (source, timestamps, user_id, modality, permissions)
- **Chunks** for indexing
- **Embeddings** for semantic retrieval

High-Level Architecture:





Modality-specific ingestion:

A) Audio ingestion (.mp3, .m4a)

Steps:

1. Upload audio → store raw file in object storage
2. Create job with metadata (user_id, original filename, recorded_at if known)
3. Worker transcribes audio (OpenAI Whisper / other ASR)
4. Normalize transcript (remove timestamps if desired, keep speaker labels if available)
5. Chunk transcript into segments (by time or token length)
6. Generate embeddings per chunk
7. Store chunk + metadata + embedding; update indexes

Why chunk by time?

Audio naturally supports time intervals, which improves retrieval ("what did I say around 2 minutes in?").

B) Documents ingestion (.pdf, .md)

PDF

- Extract text (pdfminer / pymupdf)
- Extract metadata when available:
    - title, author, created date, modified date
- If scanned PDF:
    - OCR fallback (Tesseract / cloud OCR)

Markdown

- Parse raw text
- Preserve headings as structure hints (chunk boundaries)

Steps

1. Store raw doc in object storage
2. Extract text + metadata
3. Normalize (remove boilerplate, fix encoding)
4. Chunk by semantic boundaries (headings/paragraphs) + max token length
5. Embed each chunk; store indexes

C) Web content ingestion (URL)

Steps

1. Save URL + fetch HTML
2. Extract readable text (Boilerpipe/Readability style parsing)
3. Remove nav/ads, keep title, headings, main article text
4. Store raw HTML optionally (for traceability)
5. Chunk + embed + index

Handling dynamic pages

If JS-heavy pages:

- Minimal approach: server-side fetch only (acceptable for assignment)
- Optional: headless browser (Playwright) for full rendering

D) Plain text notes

Steps

1. Accept raw text
2. Normalize whitespace, detect language optionally
3. Chunk + embed + index
4. Store with user-provided timestamp (note created time) or ingestion timestamp

E) Images (storage + searchability strategy)

We won't do full "image semantic search" by pixels in the minimal scope. Instead:

Approach

- Store image file in object storage
- Generate **searchable text** via:
    - OCR (for screenshots/doc images)
    - Captioning (optional: vision-capable model)

    - User tags/alt-text
- Store:
    - caption_text, ocr_text, tags, EXIF metadata (created time, GPS if available—careful for privacy)
- Chunk the text representation and embed it

Result

Images become searchable through their **associated text/metadata**.

1.2. Information Retrieval & Querying Strategy:

Chosen strategy: Hybrid Retrieval

Why hybrid?

- Semantic search finds meaning (“worked on pipeline”) even if exact words differ.
- Keyword/FTS is best for exact terms (“PR#3052”, “Bytebridge”, “MVVM”).
- Metadata filtering is essential for time-based queries (“last month”) and scoping (“only audio”, “only PDFs”).

Retrieval pipeline:

flowchart TD

Q [User Query] --> A [Query Understanding]

A --> T [Temporal parsing + filters]

A --> K [Keyword extraction]

A --> EM [Generate query embedding]

EM --> VS [Vector Search: Top K chunks]

K --> FTS [Full Text Search: Top K chunks]

T --> FILT [Metadata filtering: user_id, time range, modality]

VS --> MERGE [Merge + dedupe + score]

FTS --> MERGE

FILT --> MERGE

MERGE --> RERANK [Re rank (optional)]

RERANK --> CTX [Context Pack]

CTX --> LLM [LLM Answer Synthesis]

LLM --> OUT [Final Answer + citations]

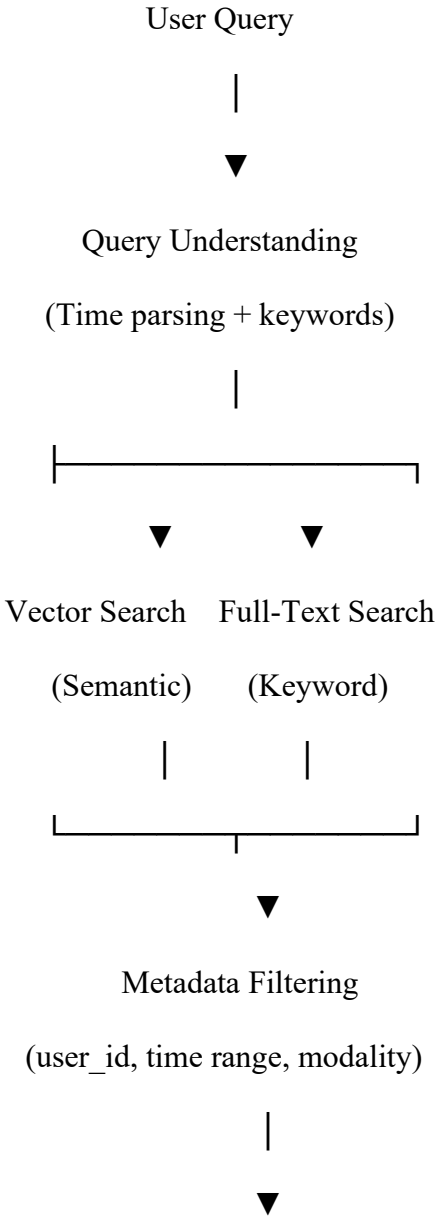
Scoring / ranking

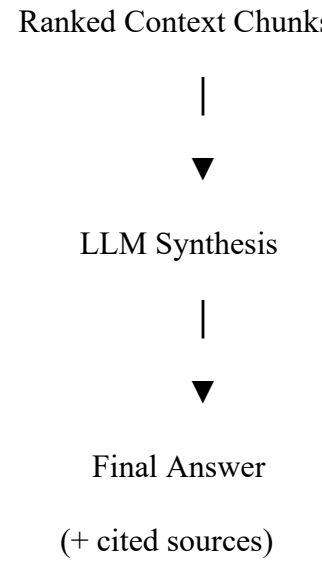
Final relevance score can be:

- score = α * vector_similarity + β * text_rank + γ * recency_boost

Recency boost helps “recent work” questions without hard filtering.

Query & Retrieval (Core “Brain”):





Why not graph-only?

Graphs are great when relationships are explicit (people, projects, tasks). But extracting reliable graph edges from arbitrary text is complex under 48 hours.
Hybrid RAG is the fastest high-quality solution.

1.3 Data Indexing & Storage Model:

Lifecycle of a piece of information:

1. **Raw ingestion**
   - Store original file or source reference (URL)
2. **Extraction**
   - Convert to normalized text + metadata
3. **Chunking**
   - Split into retrieval-friendly pieces
4. **Indexing**
   - Generate embeddings
   - Store chunks, embeddings, and full-text searchable form
5. **Serving**
   - Retrieve relevant chunks per query
   - LLM generates answer with citations

Chunking strategy:

Use a consistent chunk approach across modalities:

- Target chunk size: 300–800 tokens
- Overlap: 50–150 tokens
- Respect structure when available:
  - Markdown headings

- PDF paragraphs
- Web headings
- Audio time segments

Why chunk?

- LLM context windows are limited.
- Retrieval works better on focused, atomic content.
- Updates are easier (re-embed only changed chunks).

Indexing technique

- **Vector embeddings** (semantic): stored in pgvector (or a vector DB)
- **Full-text search** (keyword): Postgres tsvector (or Elasticsearch)
- **Metadata indexes**: B-tree indexes on user_id, modality, created_at, ingested_at

Database Schema (Postgres + pgvector)

Core entities

- users (or assume single-user for assignment)
- sources (each file/url/text/image/audio item)
- chunks (searchable units)
- chunk_embeddings (or embedding column inside chunks)

Example schema (clean, minimal):

```
-- Source = original input item (audio file, pdf, url, text note, image)

CREATE TABLE sources (

id UUID PRIMARY KEY,

user_id UUID NOT NULL,

modality TEXT NOT NULL CHECK (modality IN ('audio','document','web','text','image')),

source_uri TEXT,              -- e.g., s3://bucket/key or file path or URL

title TEXT,

created_at TIMESTAMPTZ,       -- event time (when content was created)

ingested_at TIMESTAMPTZ NOT NULL DEFAULT now(), -- when system ingested it

meta JSONB NOT NULL DEFAULT '{}':: jsonb

);
```

```
-- Chunk = retrieval unit
CREATE TABLE chunks (
id UUID PRIMARY KEY,
source_id UUID NOT NULL REFERENCES sources(id) ON DELETE CASCADE,
user_id UUID NOT NULL,
chunk_index INT NOT NULL,     -- order inside source
content TEXT NOT NULL,        -- normalized chunk text
content_tsv tsvector,         -- for full-text search
start_offset INT,             -- optional: char offset or token offset
end_offset INT,
start_time_ms INT,            -- optional: for audio/video
end_time_ms INT,
created_at TIMESTAMPTZ,       -- inherited from source or chunk-level time
ingested_at TIMESTAMPTZ NOT NULL DEFAULT now(),
modality TEXT NOT NULL,
meta JSONB NOT NULL DEFAULT '{}'::jsonb
);
-- Embeddings in pgvector (example: 1536 dims)
-- Either store in chunks table or separate table.
ALTER TABLE chunks ADD COLUMN embedding vector(1536);
-- Indexes
CREATE INDEX idx_chunks_user_time ON chunks(user_id, created_at);
CREATE INDEX idx_chunks_modality ON chunks(user_id, modality);
CREATE INDEX idx_chunks_fts ON chunks USING GIN(content_tsv);
CREATE INDEX idx_chunks_embedding ON chunks USING ivfflat (embedding
vector_cosine_ops);
```

What metadata is stored?

In sources.meta and chunks.meta

- filename, filetype, url_domain
- doc_author, doc_title, pdf_page_range
- audio_duration, speaker_labels (if available)
- image_exif_created_at, ocr_text, caption_text
- tags (user tags)
- checksum (dedup)
- processing_version (for reprocessing later)

Storage solution trade-offs:

Option chosen (best for a 48-hour build): Postgres + pgvector + Object storage

Pros

- One database for metadata + chunks + FTS + embeddings
- Simple deployment
- Great for MVP and moderate scale
- Strong querying flexibility (SQL)

Cons

- At huge scale, dedicated vector DB (Pinecone/Weaviate/Milvus) may outperform
- Postgres IVFFlat tuning needed for very large corpora

SQL vs NoSQL vs Vector DB

- **SQL (Postgres)**: best for rich filtering, joins, time queries, auditability
- **NoSQL**: flexible schema, but harder ad-hoc querying for time + text + joins
- **Vector DB**: strong similarity search, but you still need metadata store; complexity increases

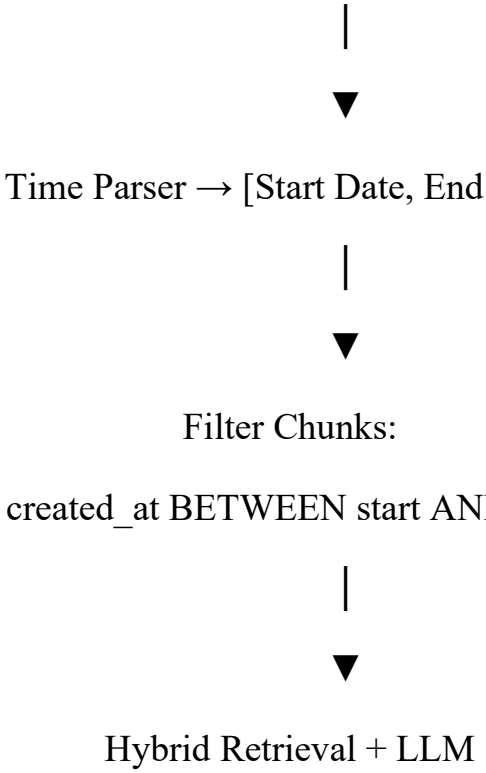
Temporal Query Support:

Data Item

├── created_at ← when work actually happened

└── ingested_at ← when system received it

Query: "What did I work on last month?"



1.4 Temporal Querying Support:

Requirement

Answer questions like:

- “What did I work on last month?”
- “Summarize my last week’s meetings”
- “Show changes over the last 3 days”

Timestamp model

We store two timestamps:

1. created_at = when the content happened / was created (event time)
2. ingested_at = when the system received it (system time)

Why both?

- User might upload old docs today → created_at is old but ingested_at is now.
- For “what did I work on last month,” we care about created_at.

How timestamps are assigned per modality

- Audio: use recording timestamp from metadata if available; else user-provided; else fallback to ingested_at
- PDF/MD: use doc metadata if present; else file modified time; else ingested_at

- Web: if article has publish date → created_at; else ingestion time
- Text notes: user supplies note time; else ingested_at
- Images: EXIF created date; else ingested_at

Temporal query execution

1. Parse time expression (e.g., "last month" → start/end range)
2. Apply metadata filter before/with retrieval:
    - created_at BETWEEN start AND end
3. Run hybrid retrieval only within that time window
4. Rank with relevance + (optional) recency within range
5. Summarize with LLM + cite sources

Example SQL filter (conceptual):

```
SELECT * FROM chunks

WHERE user_id = :uid

  AND created_at >= :start

  AND created_at < :end

ORDER BY created_at DESC

LIMIT 200;
```

1.5 Scalability and Privacy

Scalability to thousands of docs per user

Key scaling points:

- **Asynchronous ingestion**: ingestion jobs don't block the API
- **Chunk-level storage**: retrieval works on small units
- **Indexes**:
    - time indexes on created_at
    - modality and user partitioning
    - vector index (IVFFlat/HNSW depending on stack)
- **Partitioning strategy**
    - Partition chunks by user_id (logical) or time range
- **Caching**
    - Cache embeddings for repeated queries
    - Cache frequent retrieval results (optional)
- **Worker scale-out**
    - Add more ingestion workers as data grows

Privacy by design

Minimum privacy requirements:

- Every record is scoped by **user_id**
- Strong access control at API layer (auth token)
- Encrypt data at rest (db + object storage) in cloud scenario
- Avoid storing sensitive EXIF like GPS unless needed (or strip it)

Cloud-hosted vs Local-first

Cloud-hosted

- Pros: easy multi-device access, scalable, managed backups
- Cons: trust + compliance burden, sensitive data risk

Local-first

- Pros: best privacy, user keeps full control (e.g., SQLite + local vector index)
- Cons: harder sync, limited compute for transcription/LLM unless using local models

Balanced approach

- Local storage for raw personal data + optional cloud for embeddings/metadata (complex)
- For assignment, clearly state which approach you'd implement first and how you'd evolve.
