TwinMind "Second Brain" Prototype: System Design & Architecture

Name: Prerana Sathyabodha Kumsi
Email: pkumsi@asu.edu
Github: https://github.com/pkumsi
LinkedIn: https://www.linkedin.com/in/preranakumsi/

0. Executive Summary

This system is a foundational prototype for a "second brain": a personal AI companion that ingests multi-modal user data (audio, documents, web pages, notes, images), indexes it with time-aware metadata, and answers natural-language questions by retrieving relevant context and synthesizing an accurate response using an LLM.This prototype supports end-to-end ingestion and Q&A for Web URLs, PDFs, and Audio.

The architecture follows a clear lifecycle:

Artifact (raw input) → Document (extracted text) → Chunk (retrieval unit) → Embedding/FTS indexes → Retrieval + Rerank → LLM answer with citations

Key design choices:

- **Hybrid retrieval** (dense semantic embeddings + sparse keyword search) to maximize recall.
- **Lightweight reranking** to improve top-k precision.
- **Temporal-first metadata** on every record to support time-based queries.
- **Asynchronous ingestion pipeline** with job tracking to scale cleanly.

Why this matters (Second Brain framing)
A second brain must answer *across* sources, not just "find a document." A question like "What were Kraig's concerns last week?" may require correlating audio transcript mentions, a saved article, and a note. Hybrid retrieval + time-aware filtering is what makes this feel like "perfect memory," not search.

1. Part 1: System Design & Architecture

1.1 Multi-Modal Data Ingestion Pipeline

Design goals

1. Normalize different modalities into a consistent searchable representation.

2. Maintain provenance (where each chunk came from).
3. Capture time accurately for temporal questions.
4. Keep ingestion asynchronous so UI remains responsive.
5. Make modality support extensible: adding a new parser should not require changing retrieval.

Canonical internal data model

Artifact

- Represents the raw user-provided input: file upload, URL, pasted note, image.
- Stored as a pointer to blob storage + metadata.

Document

- Text content extracted from the Artifact (e.g., transcript, PDF extracted text, article body).
- Carries normalized metadata and the best-available "captured_at" timestamp.

Chunk

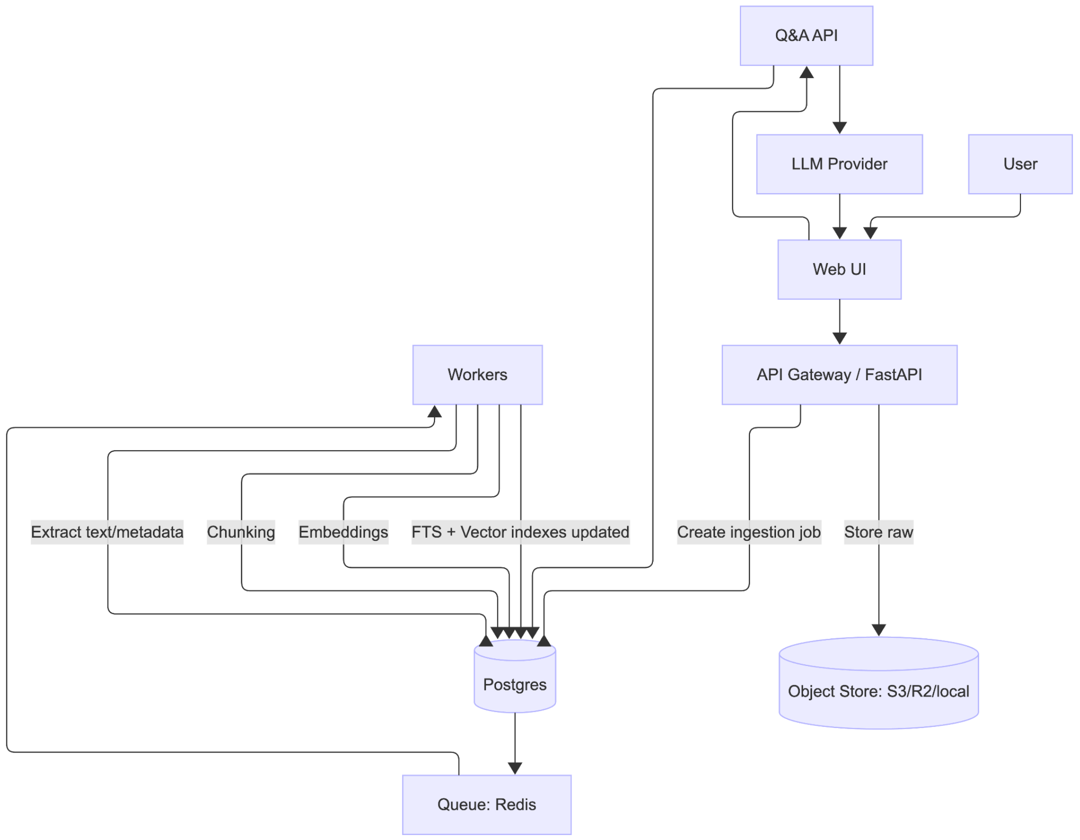
- Fixed-size retrieval unit derived from Document.
- Each chunk inherits timestamps and stores offsets for traceability.

Embedding

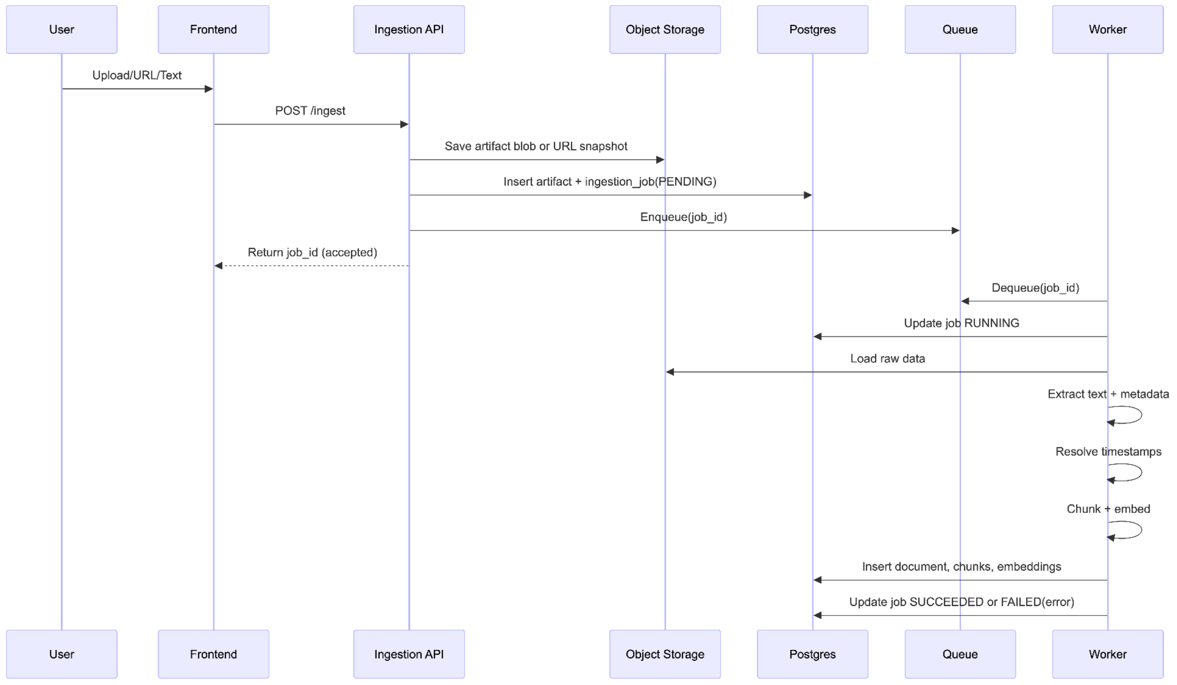
- Vector representation per chunk used for semantic search.

This "universal representation" allows the retrieval system to treat all modalities consistently.

High-level architecture diagram



Ingestion sequence diagram



1.1.1 Modality processing details

A) Audio (.mp3, .m4a)

Purpose: Convert spoken content into searchable text with time linkage.

Steps

1. Store raw audio in object storage.
2. Transcribe using Whisper (or provider API).
3. Create a Document with transcript + metadata.
4. Chunk transcript by time-segments (preferred) or token boundaries.
5. Store optional segment offsets (time_start_ms, time_end_ms) to support "jump to moment" in future UX.

Metadata

- captured_at: from user input or file metadata (fallback to ingestion time).
- duration_ms, codec, original filename.

Why this matters
Audio is often where "real work" happens (meetings, voice notes). Turning it into searchable, time-stamped chunks enables questions like "what were the concerns in the meeting last Tuesday?" with high fidelity.

B) Documents (.pdf, .md)

PDF

- Extract text via PDF parser.
- Capture page references (optional) for citations.
- Chunk by paragraphs/headings (fallback to token chunking).

Markdown

- Parse headings.
- Chunk by section with overlap.

C) Web Content (URL)

- Fetch HTML.
- Extract main readable content (readability algorithm).
- Extract metadata: title, site, author, published date if detectable.
- Chunk article by sections.

D) Plain Text / Notes

- Directly store as Document.
- Chunk by paragraphs + token limits.

E) Images (proposal)

Storage

- Store image blob in object storage.

Make searchable

- Extract EXIF (timestamp).
- Generate text via:
    - OCR for text-heavy images (screenshots)
    - Captioning (vision model) for photos
    - User tags (always available fallback)

Index

- Treat OCR/caption output as a Document and chunk it.

Prototype implementation can limit image support to: storage + user tags + EXIF time (and propose OCR/captioning as extension).

1.2 Information Retrieval & Querying Strategy

Design goals

- High recall across paraphrases and exact strings.
- High precision for top results (the chunks you feed to the LLM).
- Time-aware retrieval.
- Explainable answers via citations.

Strategy overview: Hybrid + Rerank (recommended)

Stage 1: Candidate generation (high recall)

- Dense retrieval: vector similarity search over chunk embeddings.
- Sparse retrieval: keyword search using Postgres full-text search (BM25-like behavior is approximated via ranking; true BM25 could be swapped in later).

Stage 2: Fusion

- Use **Reciprocal Rank Fusion (RRF)** to combine dense + sparse results robustly (rank-based, no score calibration needed).
    - Alternative: union + dedupe with weighted scoring (simpler implementation).

Stage 3: Lightweight reranking (precision upgrade) — Prototype decision
To be decisive and maximize relevance under mixed signals (semantic + time + metadata), this prototype uses:

Option B: LLM-based reranking (bounded to top N=30–50 candidates)

Why LLM reranking for the prototype

- Can consider **temporal relevance** and **metadata** (source authority/modality) in a single relevance judgment.
- Avoids adding another dedicated reranker dependency under tight timeline.
- Cost/latency is controlled by limiting candidate count.

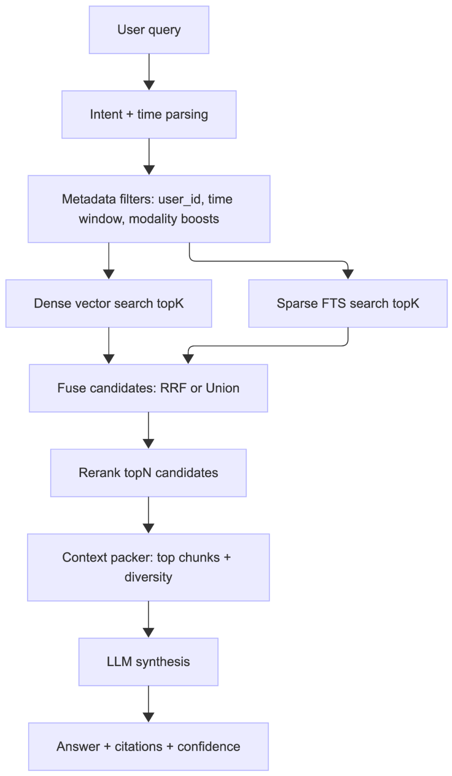
Rerank prompt (prototype)

> Given query: {query}. Rank these {N} passages by relevance.
> Consider: semantic match, temporal relevance, source authority/modality.
> Output: JSON array of passage IDs in ranked order.

Stage 4: Answer synthesis

- LLM generates response grounded in retrieved chunks.
- Output citations (document title/url + timestamp) to increase trust.

Retrieval flow diagram


User query
Intent + time parsing
Metadata filters: user_id, time window, modality boosts
Dense vector search topK
Sparse FTS search topK
Fuse candidates: RRF or Union
Rerank topN candidates
Context packer: top chunks + diversity
LLM synthesis
Answer + citations + confidence

Why not dense-only?

Dense-only retrieval is simpler but fails on:

- exact identifiers (“SER321”, “KR”, “invoice 8391”)
- rare names
- quotes and literal phrases

Hybrid retrieval reduces these misses significantly.

Why reranking matters

Hybrid improves recall; reranking improves top-k precision. This reduces hallucinations by ensuring the LLM sees the best evidence first.

Why hybrid retrieval matters for “second brain”
Dense-only might miss “KR” → “Kraig” connection; sparse-only might miss paraphrases like “big concerns” vs “risks.” Hybrid catches both.

1.3 Data Indexing & Storage Model

Lifecycle of information

1. Ingest Artifact (raw storage + metadata)
2. Extract Document text (normalized)
3. Chunk Document into retrieval units
4. Compute embeddings for each chunk
5. Update indexes (vector + FTS)
6. Serve queries via retrieval + rerank + LLM synthesis

Chunking strategy

A robust default:

- Chunk target size: **~600–900 tokens**
- Overlap: **~100 tokens**
- Boundary preference:
    1. section/heading boundaries
    2. paragraph boundaries
    3. fallback to token windows
- Store char_start/char_end offsets for traceability.

Overlap justification (why ~100 tokens)

- Prevents key information from splitting across chunk boundaries.
- ~100 tokens ≈ **1–2 sentences** of continuity, preserving context for retrieval and summarization.

- Trade-off: ~15–20% storage/embedding overhead vs significantly better answer grounding.
- Alternative considered: sentence-window retrieval (more precise boundaries) but adds complexity; deferred.

Embedding model choice (explicit)

Prototype embedding model: text-embedding-3-large (OpenAI)

- Dimensionality: **3072** (option to use 1536 if optimizing cost/speed)
- Justification:
  - strong general retrieval performance and robust semantic matching
  - multilingual-ready (future-proof)
  - straightforward API integration for a 48h build

Storage implication (order of magnitude)

- 1000 chunks × 3072 dims × 4 bytes ≈ **12 MB per user** just for raw float vectors
 (plus indexing overhead; still manageable for prototype)

Storage choice

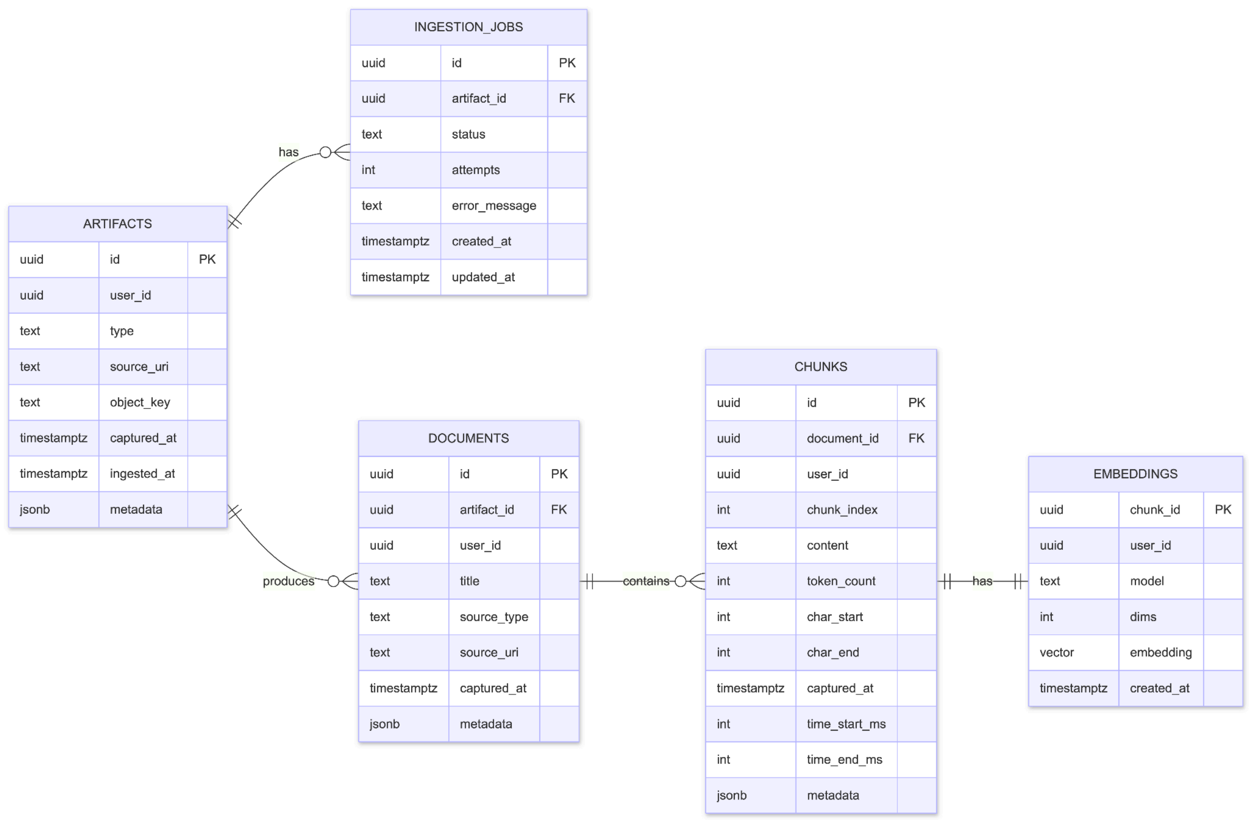
Postgres as the system of record, with:

- pgvector for vector similarity search
- Postgres FTS (tsvector) for keyword search
- Strong metadata querying (time, modality, source)

Raw files stored in object storage.

This minimizes infrastructure while retaining a scalable path forward.

1.3.1 Database schema (core tables)



INGESTION_JOBS		
uuid	id	PK
uuid	artifact_id	FK
text	status	
int	attempts	
text	error_message	
timestamptz	created_at	
timestamptz	updated_at	

ARTIFACTS		
uuid	id	PK
uuid	user_id	
text	type	
text	source_uri	
text	object_key	
timestamptz	captured_at	
timestamptz	ingested_at	
jsonb	metadata	

DOCUMENTS		
uuid	id	PK
uuid	artifact_id	FK
uuid	user_id	
text	title	
text	source_type	
text	source_uri	
timestamptz	captured_at	
jsonb	metadata	

CHUNKS		
uuid	id	PK
uuid	document_id	FK
uuid	user_id	
int	chunk_index	
text	content	
int	token_count	
int	char_start	
int	char_end	
timestamptz	captured_at	
int	time_start_ms	
int	time_end_ms	
jsonb	metadata	

EMBEDDINGS		
uuid	chunk_id	PK
uuid	user_id	
text	model	
int	dims	
vector	embedding	
timestamptz	created_at	

Indexing plan

- chunks.content_tsv (GIN) for full-text
- embeddings.embedding ANN index (HNSW preferred)
- (user_id, captured_at desc) for temporal filtering
- (document_id, chunk_index) for fast retrieval + traceability
- (user_id, type, captured_at) for modality + time queries

1.4 Temporal Querying Support

Temporal querying is a first-class requirement; the architecture treats time as core metadata, not an afterthought.

Timestamp fields

- **captured_at**: when the underlying event/content happened (user-intent time)
- **source_published_at**: for web pages (optional)
- **ingested_at**: when processed by system (fallback)

Timestamp resolution policy (robust fallback order)

When ingesting an artifact:

1. User-provided timestamp (best)

2. File metadata created/modified timestamps
3. Web publish date (if extracted)
4. Ingestion time fallback

Every chunk stores captured_at so time filtering can happen at retrieval time.

Time parsing: implementation detail (decisive)

Prototype approach: Option A (deterministic) with fallback

A) Regex + dateparser library

- Extract phrases like: “last Tuesday”, “yesterday”, “last month”, “Q3 2024”, “Jan 15”
- Use dateparser.parse() with PREFER_DATES_FROM='past'
- Resolve into one of:
    - **Hard filter window**: [start_ts, end_ts]
    - **Soft time hint**: boost a period when query is vague (“Q3 earnings call”)

B) LLM fallback for ambiguous cases

- If parsing confidence is low (e.g., “around early spring”), call a small LLM tool prompt:
    - “Extract time constraints; return JSON with start/end or ‘soft’ hints.”

Examples

- “last Tuesday” → [YYYY-MM-DD 00:00, YYYY-MM-DD 23:59]
- “last month” → [first day of last month 00:00, last day 23:59]
- “Q3 earnings call” → soft boost for Jul–Sep (don’t hard filter if uncertain)

Query-time temporal processing

1. Parse time constraints from the query.
2. Apply time window filter:
    - WHERE chunk.captured_at BETWEEN start AND end
3. For soft constraints:
    - apply time-decay or period boost (recent/within-hint favored)

Temporal examples

- “What did I work on last month?”
    - Filter to last month window → retrieve and summarize grouped by source/type/topic.
- “What were key concerns raised in the project meeting last Tuesday?”
    - Filter to Tuesday window + modality boost to audio transcripts.

Why time-first matters
Most “memory” questions are inherently temporal (“last week”, “before the deadline”, “after the

meeting"). Treating time as a first-class index is what makes this a second brain rather than a semantic search box.

1.5 Scalability and Privacy

Scaling to thousands of documents per user

- **Asynchronous ingestion** avoids blocking the UI.
- **Chunk-based indexing** keeps retrieval efficient.
- Indexes chosen for scale:
  - vector ANN index for semantic search
  - GIN full-text index for keyword queries
  - (user_id, captured_at) for time filters
- Incremental updates:
  - only embed new chunks
  - store embedding model version to support re-embedding strategy later
- Controlled latency:
  - rerank only top 30–50 candidates
  - cap context to top 8–12 chunks

Privacy by design

- Per-user isolation: every query and index is scoped by user_id.
- Encrypt at rest:
  - managed Postgres encryption + object storage encryption.
- Minimal logging:
  - log IDs and statuses, not raw content.
- Cloud vs local-first trade-off:
  - **Cloud-hosted**: best for multi-device sync + heavy compute, but requires trust in hosting.
  - **Local-first**: strongest privacy, but heavier on-device requirements and harder sync.
  - **Hybrid future**: local cache + encrypted cloud sync + opt-in redaction.

1.6 Robustness & Error Handling (Production Mindset)

Ingestion failures

- Retries: **3 attempts** with exponential backoff (e.g., 5s, 30s, 2m)
- Failed jobs:
  - persisted with error_message and stack trace (internal)
  - UI shows "failed" with human-readable reason and "retry" option
- Partial success:
  - if chunking fails, keep extracted Document text
  - if embedding fails, keep chunks (embedding can retry later)

Retrieval edge cases

- No results found:
    - respond: “I don’t have information about that yet in your saved data.”
    - suggest next step: “Try uploading the file or adding the link.”
- Very recent ingestion:
    - if job still RUNNING, answer: “Still processing X; try again shortly.”
- Malformed/underspecified query:
    - graceful fallback: retrieve broadly and ask a single clarifying follow-up **only if necessary**
    - otherwise provide best-effort summary and cite uncertainty

Rate limiting and provider failures

- LLM transient failure:
    - retry with backoff
    - if still failing, return a friendly error and preserve server logs
- Future: per-user quotas and token accounting (store usage per user)

Data quality guards

- Minimum chunk size: discard fragments under **~50 tokens**
- Maximum chunk size: split anything over **~1500 tokens**
- Deduplication:
    - content hash to avoid re-embedding identical chunks
    - checksum at artifact level to avoid reprocessing duplicates

1.7 Context Packing Strategy (Post-rerank)

Even with perfect retrieval, sending “the wrong 8 chunks” can degrade answer quality. The system uses a context packer after reranking.

Goals

- High-signal evidence
- Avoid redundancy
- Fit token budget
- Preserve temporal coverage when user asks time-range questions

Algorithm

1. **Diversity constraint**
    - group candidates by `document_id`
    - max **3 chunks per document** (unless only one doc is relevant)
2. **Temporal spread (for time-range queries)**
    - if query is range-based (e.g., “last month”), sample across the window:

        - e.g., choose top chunks per week, then rerank within each slice
3. **Token budget management**
    - target context window: **3000–4000 tokens**
    - always include at least top **5** reranked chunks
    - if needed, truncate long chunks (preserve beginning + keep citations)
4. **Metadata enrichment**
    - prepend each chunk with:
        - source title
        - modality (audio/web/note)
        - captured_at
    - enables LLM citations: "According to [Audio: Standup 2024-12-10]…"

1.8 Cost Model (Order of Magnitude)

This prototype's costs scale roughly linearly with the amount of content and number of queries.

Example ballpark per "batch"
Per ~1000 chunks ingested:

- Embeddings (3072 dims): on the order of **cents to low dollars** depending on chunk tokenization
- Storage: Postgres + object store: low monthly cost for this scale
- Audio transcription is the biggest variable (depends on minutes of audio)

Per 100 user queries (RAG responses):

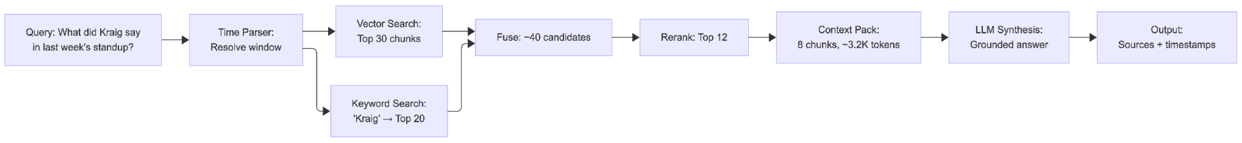
- LLM synthesis cost depends on context size (target ~4K tokens) and model choice.
- Rerank via LLM adds additional bounded cost (top 30–50 candidate snippets).

Optimization opportunities

- Cache repeated queries (Redis)
- Batch embedding requests
- Consider smaller reranker model once stable (move from LLM rerank → dedicated reranker)
- Aggressive dedupe to avoid re-embedding

Why include this section
Startups and product teams care about unit economics. Even a lightweight cost model signals you understand how this system behaves in the real world.

1.9 End-to-End Query Flow Example (Clarity Diagram)


Query: What did Kraig say in last week's standup?
Time Parser: Resolve window
Vector Search: Top 30 chunks
Keyword Search: 'Kraig' → Top 20
Fuse: ~40 candidates
Rerank: Top 12
Context Pack: 8 chunks, ~3.2K tokens
LLM Synthesis: Grounded answer
Output: Sources + timestamps