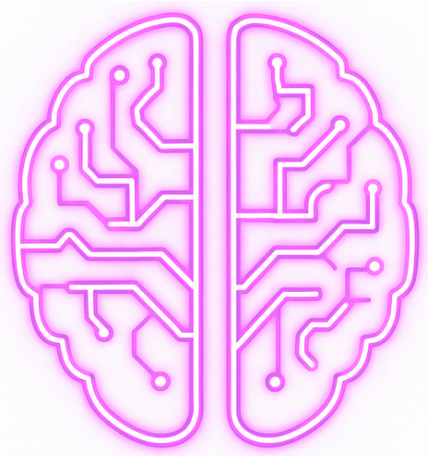
System Design Document

The "Second Brain" AI Companion

Kartikey Sharma ● 16 January 2026



TwinMind AI - 817 209 8158 - kartikeys162@gmail.com - www.kartikeysharma.com

Introduction

The “Second Brain” system is designed to serve as a comprehensive personal knowledge management platform. It enables users to ingest, organize, and query multi-modal data, ranging from voice recordings and documents to web content and images, creating a persistent, searchable digital extension of human memory. The core objective is to provide a highly grounded Q&A experience where every response is synthesized from the user’s unique data and supported by verifiable citations.

System Goals and Core Principles

Design Objectives

The system architecture is guided by the following primary goals:

- **Multi-modal Ingestion:** Seamlessly processing audio, PDFs, Markdown, plain text, web pages, and images.
- **Temporal Awareness:** Ensuring the system understands "when" information was created or discussed to support time-based queries.
- **Grounded Synthesis:** Generating fast, accurate answers derived strictly from stored sources with explicit citations.
- **Scalability & Privacy:** Supporting thousands of documents per user through a multi-tenant architecture that prioritizes data isolation.

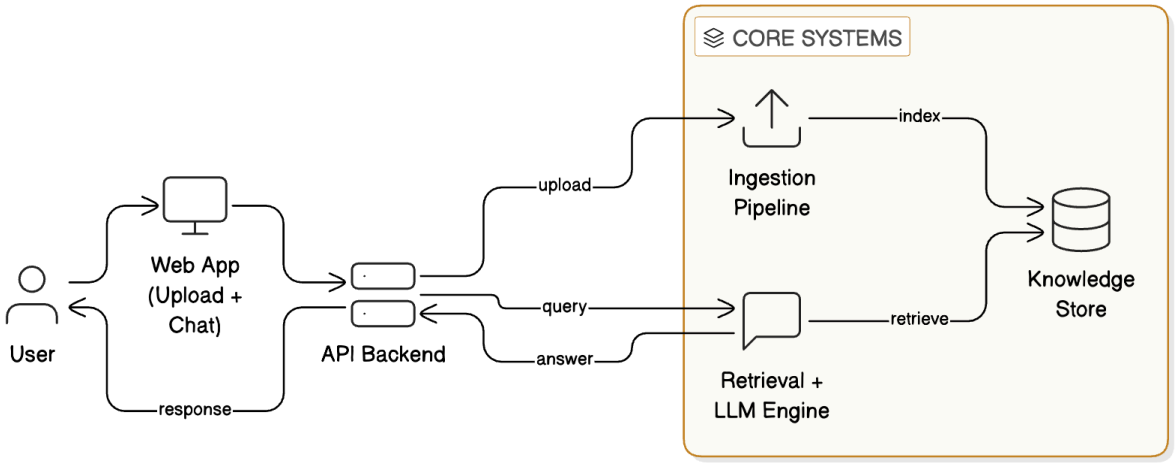
Core Principles

To maintain system integrity, it adheres to:

- **Separation of Concerns:** Distinct boundaries between data ingestion, storage, retrieval, and generation layers.
- **Traceability:** Every LLM-generated response must be mapped back to specific source chunks and timestamps.
- **Hybrid Retrieval:** Combining semantic vector search with lexical (keyword) search and metadata filtering for maximum precision.
- **Privacy by Design:** Implementing encryption at rest and in transit, with strict tenant isolation to prevent data leakage.

High-Level Architecture

The system is composed of several modular components that coordinate the flow from raw data to actionable intelligence.

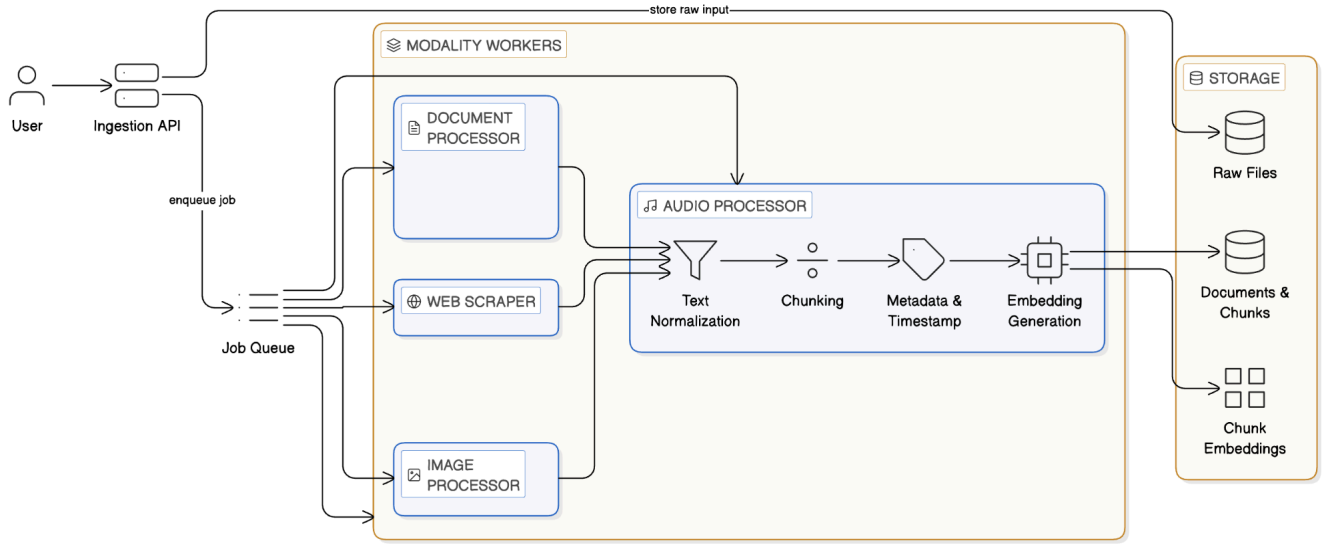


Component Overview

- **Client Interface:** A web application for file uploads, URL submission, and a chat-based query interface.
- **API Gateway & Backend:** Manages authentication, ingestion requests, and the primary chat endpoints.
- **Asynchronous Ingestion Pipeline:** A distributed worker system that handles modality-specific extraction and processing.
- **Knowledge Store:** A multi-layered storage solution involving object storage for raw artifacts, a relational SQL database for metadata, and specialized indexes (Vector and Full-Text) for search.
- **Retriever & Orchestrator:** A logic layer that identifies relevant context, reranks results, and manages the Large Language Model (LLM) interaction to stream answers back to the user.

1. Multi-Modal Data Ingestion Pipeline

The ingestion pipeline is the engine that transforms diverse raw data into a structured, searchable knowledge base. To ensure reliability and a responsive user experience, this entire process operates asynchronously, isolating slow or unreliable tasks like audio transcription and web scraping from the primary user interface.



1.1 Common Pipeline Stages

Every modality, regardless of its original format, passes through a standardized sequence of operations to ensure consistency across the system:

1. **Receive & Persist:** The raw artifact is stored in cloud object storage (S3) or a local store for local-first configurations.
2. **Extraction & Normalization:** Modality-specific tools extract text and metadata. The text is cleaned, boilerplate is removed, and a canonical title is generated.
3. **Chunking & Embedding:** Content is broken into retrieval-friendly segments with stable boundaries. These chunks are then converted into high-dimensional vector embeddings.
4. **Indexing & Quality Control:** Chunks and metadata are upserted into a Vector DB, a full-text index, and SQL metadata tables. Security protocols, including malware scanning and content size limits, are enforced before the job is marked "completed."

1.2 Modality-Specific Logic

Audio (.mp3, .m4a): The system uses OpenAI Whisper model to generate transcripts with word-level or segment-level timestamps.

- We store the full transcript alongside detailed segments containing start/end offsets and optional speaker labels.
- Chunks are aligned to speech segments (ideally 30–90 seconds) to ensure that "*What happened at minute 12?*" queries return accurate, context-complete results.

Documents (.pdf, .md, .txt): PDFs are processed via parsers to extract text and structural metadata (author, page count). For scanned documents, an OCR fallback is utilized.

- Structure-Aware Chunking: For Markdown and PDFs, we split data by headings, pages, or paragraphs, maintaining a 10–15% overlap between chunks to preserve context. Code blocks in Markdown are treated as distinct chunk types to improve technical query accuracy.

Web Content (URL): The system fetches HTML while respecting robots policies and timeouts. A readability extractor isolates the primary content, stripping away ads and navigation menus.

- Reproducibility: While storing raw HTML snapshots increases storage costs, it allows the system to verify "what the page said then" versus its current state. At a minimum, extracted text and the fetch timestamp are indexed.

Images & Plain Text Notes: Text notes are the simplest modality, using paragraph-based chunking for long entries. For images, the system focuses on searchability through enrichment. We generate captions via Vision LLMs and extract OCR text from screenshots. This text, combined with EXIF data and filenames, is embedded to allow users to find images using natural language (e.g., "*Find that screenshot about the project budget*").

1.3 Pipeline Orchestration and Reliability

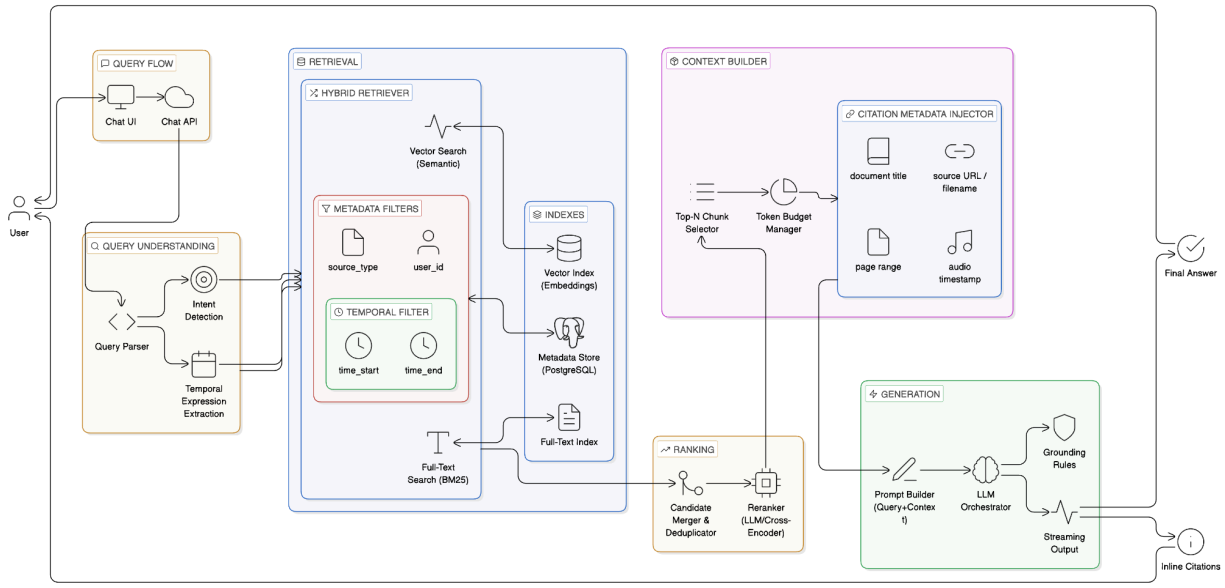
To manage these complex workflows, an orchestrator manages `ingestion_job` rows and task queues.

Idempotency & Retries: Every stage is designed to be idempotent; stable `chunk_id` generation ensures that repeated runs do not create duplicate data. The system automatically retries transient network or API errors, while persistent failures are moved to a dead-letter queue.

Transparency: Each job tracks its progress through granular statuses (e.g., "Transcribing," "Indexing"), allowing the UI to provide real-time updates to the user.

2. Information Retrieval and Querying Strategy

To deliver precise answers from a vast personal knowledge base, the system utilizes a Hybrid Retrieval architecture. This multi-layered approach combines the "meaning-based" strengths of semantic search with the "exact-match" precision of keyword indexing, all while enforcing strict temporal and privacy filters.



The Retrieval Pipeline

The system processes a user's question through a structured pipeline designed to maximize relevance and minimize "hallucinations."

Query Understanding and Candidate Generation: When a query is received, the system first determines if it is time-scoped (e.g., "last Tuesday" or "during Q3") and extracts key entities or rare tokens. From there, it initiates two parallel searches:

- Vector Search: Identifies the top 50 candidates based on embedding similarity, capturing conceptual matches even if the wording differs.
- Full-Text Search: Identifies the top 50 candidates using BM25 or lexical matching, ensuring exact names, IDs, or technical terms are not missed.

Filtering, Reranking, and Assembly: Once candidates are merged and de-duplicated by chunk_id, the system applies a metadata layer. This ensures strict tenant isolation (filtering by user_id) and applies temporal constraints if the user specified a time window. To ensure the highest quality results, a Cross-Encoder reranker evaluates the top 30 remaining candidates, selecting the most relevant segments to fit within the LLM's token budget.

The Answering Phase: The final response is generated by an LLM provided with the user's query and the curated context chunks. To maintain high factual integrity, the system operates under a strict "groundedness" instruction: it must answer only using the provided context, include structured citations (source titles, URLs, or time ranges), and explicitly state if information is missing.

Why Not "Graph-Only"?

While Knowledge Graphs (KG) are powerful for mapping complex relationships, building them reliably from fragmented personal data is computationally expensive and prone to noise.

We have opted for a Vector-First approach for the MVP core because it is more resilient to arbitrary data formats. However, our architecture remains extensible; we store extracted entities and relationships as optional enrichments. This allows us to use Graph-based retrieval as a "secondary boost" in later iterations (for example, to surface "all people connected to Project X") without the high overhead of a pure graph-reliant system.

3. Data Indexing and Storage Model

The system's data architecture is designed to transition raw, immutable artifacts into high-performance, searchable indices. This process ensures that every piece of information is trackable from its original source to its vector representation.

3. 1 Data Lifecycle and Storage Strategy

The data journey begins with raw files stored in object storage (S3 or GCS). Once text is extracted, it is structured into documents, which are then broken down into chunks. These chunks serve as the foundational unit for both vector and full-text indexing.

Choosing the Right Database Stack: We utilize PostgreSQL as the primary engine. This "single-database" approach ensures strong transactional integrity and simplifies joins between metadata and vector data.

- **Relational Metadata:** Postgres handles complex time queries and user-specific filtering with high reliability.
- **Vector Search:** We use `pgvector` for its operational simplicity, though the architecture is designed to migrate to dedicated vector databases (like Pinecone) if extreme scale is required.
- **Full-Text Search:** Native GIN indices on `tsvector` columns provide robust keyword search capabilities without the need for an external Elasticsearch cluster.

3. 2 Core Schema Definitions

To ensure clarity and scalability, the schema is organized into three primary functional areas.

- `users`: Tracks identity via `id (UUID)`, `email`, and `created_at`.
- `documents`: The primary metadata container.
    - Source Details: Tracks `source_type` (audio, pdf, etc.), `source_uri`, `original_filename`, and a `content_hash` for idempotency.

    - Temporal Fields: Stores `created_at` (event time), `ingested_at` (processing time), and `fetched_at` (web-specific).
    - Governance: Includes a `metadata` JSONB blob and a lifecycle `status` (pending, ready, failed).
- `chunks`: Granular segments of a document.
    - Structural Markers: Includes `chunk_index`, `token_count`, and character/page offsets (`page_start`, `page_end`).
    - Temporal Markers: Stores semantic ranges (`time_start`, `time_end`) and audio alignment offsets (`source_offset_ms`).
    - Filtering: Denormalizes `user_id` for high-speed, secure query filtering.
- `chunk_embeddings`: Maps a `chunk_id` to its high-dimensional `vector`. It also tracks the `embedding_model` used to prevent version mismatches.
- `ingestion_jobs`: An orchestration table that monitors the `stage` of processing (e.g., extracted, chunked, embedded) and captures `error` logs for debugging.

3.3 Search Index Configuration

The system maintains a dual-index strategy within Postgres to support hybrid retrieval:

1. **Lexical Index:** A generated `tsvector` column on the `chunks` table, indexed via a GIN (Generalized Inverted Index), facilitates rapid English-language keyword matching.
2. **Vector Index:** An HNSW or IVF index is applied to the `chunk_embeddings` table, optimized for high-speed Approximate Nearest Neighbor (ANN) semantic search.

3.4 Chunking Methodology

The transformation of raw text into chunks is governed by two goals: maintaining sufficient context and ensuring ID stability.

Rather than arbitrary character counts, we use structure-aware splits. For PDFs, this means respecting page or paragraph boundaries; for Markdown, splitting by headings; and for audio, aligning with segment timestamps. Each chunk maintains a 10–15% overlap with its neighbor to ensure that context isn't lost at the boundary, and we store precise source offsets to allow the AI to provide exact page or timestamp citations.

4. Temporal Data Modeling and Processing

A robust temporal system is essential for accurately addressing user queries such as, "What did I work on last month?" This requires a clear distinction between the actual occurrence of an event and its entry into the system.

4.1 Time Model: Event Time vs. Ingestion Time

To ensure data integrity and retrieval accuracy, we categorize timestamps into two primary types:

- **Event Time:** This represents the actual date or time the content refers to (e.g., a meeting date, the time a recording was captured, or the original document creation date).
- **Ingestion Time:** This represents the specific moment the system processed the data.

Key Schema Fields:

- `documents.created_at`: The best-effort representation of Event Time.
- `documents.ingested_at`: The system-level Processing Time.
- `chunks.time_start` / `chunks.time_end`: Fine-grained temporal markers used for granular data, particularly essential for audio segments.

4.2 Timestamp Assignment by Modality

Temporal extraction logic varies based on the source material to ensure the highest possible precision:

- **Audio Content:** Timestamps for `chunks.time_*` are derived from transcript segment offsets. If the recording start time is known, we calculate absolute timestamps (e.g., `recorded_at` + offset). If unknown, we store relative offsets alongside any user-provided metadata.
- **PDF/Markdown/Text:** We prioritize `documents.created_at` from file metadata. If metadata is unreliable, we use user-provided dates; if neither is available, we fall back to the upload time.
- **Web Content:** The `fetched_at` value serves as the exact ingestion point. Additionally, we attempt to parse `article_published_at` from the page metadata to store as canonical event time.
- **Images:** We prioritize the `EXIF DateTimeOriginal` tag. If absent, we use the upload time. While OCR or captioning may identify dates within the image text, these are treated as metadata enrichments rather than canonical timestamps.

4.3 Query-Time Temporal Handling

The system must dynamically resolve time-based intent during the retrieval and generation phases.

Temporal Parsing: The system detects natural language expressions (e.g., "last Tuesday," "yesterday," "in Nov 2025") and converts them into a standardized [time_min, time_max] range, adjusted for the user's specific timezone.

Filtering Logic: Filters are applied at the chunk level to ensure relevance. The system includes chunks where:

- The `time_end` is null but the `document.created_at` falls within the range.
- The chunk's specific time range overlaps with the query range.

For specific requests like "last Tuesday's meeting," the system applies a relevancy boost to audio chunks with matching dates, documents tagged as "meeting notes," and files with titles containing the target date.

Ranking and Scoring: When a specific time is identified in a query, the ranking algorithm incorporates a temporal scoring term. This favors data chunks that are closest to—or precisely within—the requested interval.

Response Formatting: To provide a clean user experience, the model is instructed to:

- Organize findings into bullet points grouped by day.
- Include explicit dates within the body of the response.
- Provide detailed citations that reference chunk-specific time ranges, page numbers, or source URLs.

5. Scalability and Privacy

As the system expands, it must maintain high performance while ensuring rigorous data protection. This section outlines the architectural strategies for scaling to thousands of documents per user and maintaining a "privacy-first" posture.

5.1 Scaling to Thousands of Documents per User

To handle high-volume data ingestion and retrieval without latency degradation, the following tactics are employed:

- **Asynchronous Ingestion:** Leveraging horizontal worker scaling to process document uploads in the background.
- **Incremental Indexing:** Using `content_hash` gating to ensure only new or modified chunks are embedded, significantly reducing computational overhead.
- **Efficient Retrieval:**
    - ANN (Approximate Nearest Neighbor): Utilizing vector search with a restricted `topK` to maintain speed.
    - Full-Text Indexing: Implementing GIN (Generalized Inverted Index) for high-performance keyword searches.
    - Two-Stage Retrieval: Re-ranking only the top N candidates to balance precision and performance.

Caching Strategies

- Embedding Cache: Storing vector representations of common query phrases.
- Result Cache: Storing retrieval results for identical query and time-window combinations to bypass redundant DB lookups.

Sharding and Partitioning

- Data Partitioning: Segmenting chunks by `user_id` hash or by time intervals for enterprise-level tenants.
- Vector Namespacing: Enforcing strict logical separation within the vector database using per-user namespaces.

Operational Safeguards

- Implementing backpressure mechanisms on ingestion queues to prevent system saturation.
- Enforcing rate limits on third-party model APIs.
- Utilizing batch embedding calls to optimize API throughput and cost.

5.2 Privacy by Design (Cloud-Hosted)

For cloud-based deployments, privacy is integrated at the architectural level rather than treated as an afterthought.

Tenant Isolation

- Schema Enforcement: Every table is strictly keyed by `user_id`.
- API Authorization: Mandatory, granular authorization checks for every data access request.
- Vector Security: Combining per-user namespaces with robust access control lists (ACLs).

Encryption Standards

- In Transit: Mandatory TLS encryption for all data movement.
- At Rest: Full encryption for object storage and database disks.
    - Support for optional per-user encryption keys for advanced security requirements.

Data Minimization and Control

- Retention Policies: User-controlled settings for raw file retention.
- Cascade Deletion: When a user deletes a document, the system automatically purges all associated chunks and embeddings.

Auditability

- Access Logs: Detailed logging of every document or chunk accessed during the generation of an answer.
- Transparency: Providing a "Why did you answer that?" feature that surfaces the specific citations and logic used by the AI.

5.3 Local-First Approach (Trade-offs)

For users requiring the highest level of sovereignty, a local-first deployment option is available, albeit with specific technical trade-offs.

Local-First Architecture

- Storage: Running SQLite and local vector indices directly on the user's device.
- File System: Utilizing local file storage rather than cloud buckets.

Comparison of Approaches

- Pros: Guarantees maximum privacy and enables offline functionality.
- Cons: Limited by local hardware constraints for heavy tasks (e.g., high-fidelity audio transcription or large language model inference); cross-device synchronization introduces significant complexity.

Hybrid Configurations

- Selective Cloud: Storing raw files locally while utilizing the cloud for embeddings and LLM processing only with explicit user consent.
- Encrypted Sync: Uploading data to the cloud in an end-to-end encrypted format where the cloud provider cannot access the content.