System Design Document: Memora

Version 1.0

November 16, 2025

Contents

1 Introduction

1.1 System Overview

Memora is a secure, multi-modal personal knowledge base designed to ingest, understand, and retrieve a user's entire digital history. It supports a wide variety of content types, including documents, images, audio, web pages, and code.

The system's core functionality is built on a **privacy-first, encryption-at-rest** architecture, ensuring that all user data is cryptographically isolated. Information retrieval is powered by a **hybrid search strategy** that combines semantic understanding, temporal filtering, and recency boosting to provide highly relevant, context-aware answers to natural language queries.

1.2 Design Goals & Principles

The architecture is guided by the following core principles:

- **Privacy & Security First:** All user content is encrypted with a per-user key *before* being stored. Privacy is non-negotiable and prioritized over search convenience.
- **Unified Ingestion:** A single, modular pipeline (`Extract` → `Encrypt` → `Chunk` → `Embed` → `Store`) is used for all content types, simplifying the codebase and enhancing extensibility.
- **Hybrid Retrieval:** Combine the strengths of semantic (vector) search, temporal filtering, and recency scoring to deliver nuanced and accurate results that keyword-only systems miss.
- **Fast User Experience:** Asynchronous processing for ingestion ensures the UI remains fast and responsive, even when uploading large files.
- **Optimized Cost-Performance:** Use high-performance, open-source libraries for common tasks (e.g., PDF/DOCX parsing) and leverage powerful AI models (Gemini, OpenAI) only for complex, unstructured data (e.g., images, audio).

2 System Architecture

2.1 High-Level Architecture

The system is a cloud-hosted web application built on a serverless architecture.

1. **Client (Frontend):** A Next.js web application handles user authentication (Google OAuth) and file uploads.
2. **Application Layer (Backend):** Vercel Serverless Functions (Next.js API routes) manage business logic, including ingestion, encryption, and querying.
3. **Database:** A Neon serverless PostgreSQL database stores all user data, including encrypted content, metadata, and vector embeddings.
4. **External Services:**
   - **Google Gemini:** Used for LLM-based answer generation (2.5 Pro) and complex extraction (Vision, Audio).
   - **OpenAI:** Used to generate high-quality text embeddings (`text-embedding-3-small`).

2.2 Technology Stack

3 Data Ingestion Pipeline

3.1 Unified Ingestion Flow

All content follows a five-step, asynchronous pipeline:

Table 1: Technology Stack

Component	Technology	Rationale
Framework	Next.js (on Vercel)	Integrated frontend/backend, serverless scaling, fast UI.
Database	PostgreSQL (Neon Serverless)	ACID compliance, JSONB support, future-proof (pgvector
ORM	Drizzle	Lightweight, fast, and type-safe SQL query builder.
Authentication	Google OAuth	Secure, standard, and integrates with `users` table.
Embeddings	OpenAI `text-embedding-3-small`	1536-dim, industry-leading quality, fast cosine similarity.
LLM (Generation)	Google Gemini 2.5 Pro	State-of-the-art answer synthesis from context.
LLM (Extraction)	Google Gemini (Vision, Audio)	High-quality OCR, handwriting/chart recognition, transcri
Parsing Libs	`pdf2json`, `mammoth`, `xlsx`, `cheerio`	Fast, native JS parsers for common formats (cost/performa
Date Logic	`date-fns`	Reliable and robust temporal parsing.
Encryption	`crypto` (Node.js)	AES-256-GCM for strong, authenticated encryption.

1. **Extract:** The raw file is converted into plain text using a format-specific extractor. Content-based dates are also extracted via regex.
2. **Encrypt:** The full text and all future chunks are encrypted using a per-user AES-256-GCM key.
3. **Store (Document):** The original encrypted document and its metadata (file name, type, content dates) are saved to the `documents` table.
4. **Chunk:** The plaintext is divided into 1000-character chunks with a 200-character overlap to preserve semantic context across boundaries.
5. **Embed & Store (Chunks):** Chunks are encrypted, sent in batches to the OpenAI API for embedding, and stored in the `chunks` and `embeddings` tables.

3.2 Format-Specific Extraction

A dedicated extractor is used for each content type to optimize for speed, cost, and reliability.

Table 2: Format-Specific Extraction Methods

Format Category	Extraction Method	Rationale
PDF Documents	`pdf2json` library	Lightweight, Next.js compatible, handles complex layouts.
Microsoft Office	`mammoth`, `xlsx`, `pptx-parser`	Native JS libraries, no external dependencies, fast.
OpenOffice	Google Gemini 2.5 Pro	Complex XML structure, Gemini handles natively.
Images	Google Gemini Vision API	State-of-the-art OCR, handles handwriting, charts.
Audio	Google Gemini Audio API	High-quality, multilingual transcription.
Web Pages	`cheerio` HTML parser	Fast DOM manipulation, extracts main content.
Code Files	Direct text read	Preserves syntax, no parsing needed.

3.3 Asynchronous Processing

To ensure a fast user experience, document upload is non-blocking.

1. User uploads a file.
2. The API immediately saves the original document, encrypts it, and stores it in the `documents` table.
3. An instant success response (`{ documentId, success: true }`) is returned to the user.
4. In the background, a non-blocking process handles chunking and embedding (`processDocumentEmbeddings(...)`).

This means the document is stored instantly but becomes searchable only after a short delay (typically 2-3 seconds).

4 Data Storage & Schema

4.1 Database Choice: PostgreSQL (Neon Serverless)

PostgreSQL was chosen over NoSQL or dedicated Vector DBs for the following reasons:

- **ACID Compliance:** Guarantees data integrity for relational data (users, documents, chunks).
- **JSONB Support:** Provides a flexible, queryable, and indexed column for `metadata_json`.
- **Referential Integrity:** Foreign keys (e.g., `chunk.document_id`) ensure data consistency.
- **Future-Proof:** Can be upgraded with the `pgvector` extension for native, database-level vector search when scale demands it (Phase 2).
- **Simplicity:** Avoids the cost and complexity of managing two separate databases (a relational DB + a vector DB) in Phase 1.

4.2 Database Schema

The schema is designed to be relational, secure, and multi-tenant.

- `users`: Stores user profile information from Google OAuth.
- `documents`: Stores the original ingested content and high-level metadata.
  - `id` (uuid): Primary key.
  - `userId` (uuid, FK to `users`): Isolates data per user.
  - `contentEncrypted` (text): Base64-encoded AES-256-GCM ciphertext of the *full* document.
  - `metadataJson` (jsonb): Stores file name, size, MIME type, source URL, and extracted `contentDates`.
  - `created_at` (timestamp): The *upload* timestamp.
- `chunks`: Stores encrypted text segments for retrieval.
  - `id` (uuid): Primary key.
  - `document_id` (uuid, FK to `documents`): Links chunk to its source.
  - `chunk_index` (int): The order of the chunk (0, 1, 2...).
  - `content_encrypted` (text): Base64-encoded ciphertext of the *chunk*.
  - `start_char` / `end_char` (int): Location within the original text.
- `embeddings`: Stores vector representations of chunks.
  - `id` (uuid): Primary key.
  - `chunk_id` (uuid, FK to `chunks`): Links embedding to its chunk.
  - `embedding` (text): JSON string representation of the `float[1536]` vector.
  - `model` (text): e.g., "text-embedding-3-small".
- `sessions` / `messages`: Store chat history, with `messages.content` also encrypted.

4.3 Metadata & Content Date Extraction

A critical feature is the `metadataJson.contentDates` array.

- **During ingestion:** A regex parser extracts all identifiable dates (e.g., "January 5, 2025", "2025-01-05", "Nov 5") from the text.

- **Storage:** These are stored in an array and a date range (`earliest`, `latest`) within the `metadataJson` field.
- **Purpose:** This allows the system to differentiate between *when a document was uploaded* (`created_at`) and *what time period the document is about* (`contentDates`).

5 Information Retrieval Strategy

5.1 Overview: Hybrid Multi-Method Search

Retrieval is not based on a single method. It is a weighted, multi-step process that combines three signals for maximum relevance:

1. **Temporal Filtering (Pre-Search):** Narrows the search space using date logic.
2. **Semantic Search (Primary):** Understands the *meaning* and *intent* of the query using vector similarity.
3. **Recency Boosting (Post-Search):** Applies a subtle boost to more recent documents.

5.2 Detailed Retrieval Flow

This 6-step flow executes for every user query:

Step 1: Temporal Parsing The user's raw query (e.g., "What did I work on last week?") is parsed.

- **Temporal Expression:** "last week" is extracted.
- **Date Range:** A date range is calculated (e.g., Nov 8 - Nov 15, 2025).
- **Cleaned Query:** The query is cleaned for embedding ("What did I work on").

Step 2: Query Embedding The **cleaned query** is sent to the OpenAI API to be converted into a `float[1536]` query vector.

Step 3: Temporal Filtering (In-Memory) All embeddings for the user are fetched from the database. The application layer *first* filters this list, dramatically reducing the search space. A document is **INCLUDED** if:

- Its `created_at` (upload date) is within the query's date range.
- **OR**
- Any date in its `metadata_json.contentDates` array falls within the date range.

  Example: A query for "last week" (Nov 8-15) will match a document uploaded in *September* if that document's *content* mentions a meeting on "November 10th".

Step 4: Vector Similarity Search For the **filtered set** of chunks, a Cosine Similarity calculation is performed in-memory between the query vector and each chunk vector.

- **Formula:**
$$\text{similarity}(A, B) = \frac{A \cdot B}{||A|| \times ||B||}$$
- **Why In-Memory?** This is a Phase 1 trade-off. Since embeddings are stored as encrypted text/JSON, the DB cannot run vector operations. The calculation is moved to the application layer, which is fast enough for $<$10,000 documents.

Step 5: Recency Boosting A final score is calculated by adding a small boost for recent documents. This helps break ties when semantic similarity is close.

- **Formula:** `recency_boost = max(0, 1 - days_since_creation / 365) * 0.1`
- **Effect:** A document created today gets a `+0.1` boost; one from 6 months ago gets `+0.05`.
- **Final Score:** `final_score = similarity + recency_boost`

Step 6: Context Building & Answer Generation

- The Top K (e.g., K=10) highest-scoring chunks are selected.
- Their encrypted content is decrypted using the user's key.
- The decrypted text is formatted into a "context string" with citations.
- This context and the original query are sent to Google Gemini 2.5 Pro, which synthesizes a final, human-readable answer.

5.3 Rejected Approaches

- **Keyword-Only:** Rejected because it fails to understand context. It would miss a query for "budget meeting" if the text only contained "Q4 financial planning session".
- **Graph-Based (e.g., Neo4j):** Rejected as overkill for a personal knowledge base. It requires a predefined schema, complex entity extraction, and is not suited for highly unstructured, diverse personal data.

6 Security & Privacy

6.1 Encryption-First Design

Privacy is the system's foundational principle.

- **Model:** Per-user, envelope encryption.
- **Flow:**
  1. A single, high-entropy **Master Key** is stored as a server-side environment variable (`ENCRYPTION_KEY`).
  2. When a user acts, a **Per-User Key** is derived via `PBKDF2(masterKey, userId)`. This key is *never* stored, only derived in-memory.
  3. All content (documents, chunks, chat messages) is encrypted using **AES-256-GCM** with this Per-User Key.
  4. The resulting ciphertext (Base64) is all that is ever stored in the database.
- **Benefits:**
  - **Zero-Knowledge (Admin):** Database administrators or anyone with access to the database *cannot* read user content. They only see ciphertext.
  - **Cryptographic Isolation:** A breach of User A's key (theoretically) does not impact User B's data.
  - **Breach Containment:** A database dump exposes only encrypted, unusable data.
- **Trade-off:**
  - This design *intentionally* sacrifices the ability to use database-level full-text search, as the DB cannot index ciphertext.
  - All search and decryption must happen in the application layer. This is an acceptable performance trade-off for the privacy gained.

6.2 Data at Rest & In Transit

- **At Rest:** All sensitive content (`documents.contentEncrypted`, `chunks.contentEncrypted`, `messages.content`) is encrypted with AES-256-GCM.
- **In Transit:** All communication is enforced over HTTPS. Session tokens are stored in secure, HTTP-only cookies.

6.3 User Isolation

All database queries are *mandated* to include a `WHERE` clause filtering by the authenticated user's ID. This is enforced at the ORM/application level to prevent Insecure Direct Object Reference (IDOR) attacks.

```
-- All queries are namespaced
SELECT * FROM documents WHERE user_id = $session_user_id;
```

Listing 1: User-Scoped Query Example

7 Scalability & Performance

7.1 Current Capacity & Bottlenecks

The primary bottleneck in Phase 1 is the **in-memory vector similarity calculation**.

Table 3: Scalability Metrics (Phase 1)

Metric	1,000 Documents	10,000 Documents	Notes
Chunks	~10,000	~100,000	10 chunks/doc avg.
Storage	~500 MB	~5 GB	Includes embeddings.
Search Time	~100 ms	~500 ms	In-memory calculation.

The current architecture is performant up to ~10,000 documents per user. Beyond 100,000 chunks, the 500ms+ search time and memory-loading become a concern.

7.2 Phased Scaling Plan

- **Phase 1 (Current): 0 - 10K Docs/User**
  - **Strategy:** In-memory vector search.
  - **Storage:** Embeddings as JSON strings in PostgreSQL.
  - **Action:** No changes needed.
- **Phase 2: 10K - 100K Docs/User**
  - **Strategy:** Move search to the database.
  - **Action:**
    1. Install the `pgvector` extension on Neon.
    2. Migrate `embeddings.embedding` column from `text` to `vector(1536)`.
    3. Create a `USING ivfflat` index for fast Approximate Nearest Neighbor (ANN) search.
    4. Change retrieval query to use the native `<=>` (cosine distance) operator.
  - **Result:** 10-50x faster queries, minimal application changes.
- **Phase 3: 1M+ Docs/User**

– **Strategy:** Decouple storage and search.

– **Action:**

1. Introduce a dedicated Vector Database (e.g., Pinecone, Weaviate).
2. Ingestion pipeline pushes embeddings to the vector DB.
3. Retrieval queries the vector DB for Top-K `chunk_ids`, then fetches encrypted content from PostgreSQL.

– **Result:** Horizontal, distributed scaling for massive datasets.