SECOND BRAIN – SYSTEM DESIGN DOCUMENT

Introduction

The main idea behind this project is to build a small prototype that behaves like a "second brain." I want an AI system that can take in different kinds of information, remember it, understand it, and help me later when I ask questions. This could be meeting notes, text I paste, audio recordings, or web pages. Once the system has seen the information, I should be able to ask something like "What were the issues discussed in the meeting?" and get an appropriate answer based on the stored data.

This document explains how I designed the system, how it works internally, and how each part contributes to the final behaviour.

1.Data Ingestion Pipeline

The first part of the system is the ingestion pipeline. This is where the system accepts information in different formats and turns everything into clean text that can later be searched.

Plain text:

When the user sends plain text, the backend simply takes it, breaks it into smaller chunks, creates embeddings for each chunk, and stores them. This is the simplest and fastest path.

Audio files:

For audio files like mp3 or m4a, the system uses an AI transcription model to turn spoken words into text. After we have text, it goes through the same process as any other note.

Documents:

If the user uploads a PDF or markdown file, the backend extracts the text from the document. The text is then split into meaningful sections, such as paragraphs, and each section is processed and stored like any other text input.

Web content:

If the user gives a URL, the backend fetches the HTML, removes unnecessary parts like menus and ads, and extracts the main article content. That text is then chunked, embedded, and stored.

Images:

For images, the approach is to attach a text description to the image. This could be provided by the user or generated by a model. That description text is what gets embedded and stored, making the image searchable by meaning.

In all cases, the ingestion pipeline transforms different kinds of inputs into a unified format: text chunks with metadata.

2. Retrieval and Querying Strategy

This part determines how the system finds relevant information when the user asks a question. I decided to use semantic search because it understands meaning rather than just matching exact words.

Here is the basic idea:

1. Every chunk of stored data is converted into a vector (embedding).

2. When the user asks a question, the question is also converted into a vector.

3. The system compares the question vector with all stored vectors.

4. The chunks that are closest in meaning are considered the most relevant.

5. Those chunks are returned to answer the user's question.

This method allows the system to find the right information even if the wording of the question is different from the wording in the stored notes. It makes the system more flexible and natural.

3. Data Indexing and Storage Model

I designed the storage layer around two concepts: **Sources and Chunks**.

A Source represents the original input. This could be a PDF file, a piece of text, a web page, or an audio transcript. A Source contains metadata such as the type of content, the title, and the timestamps.

Chunks are smaller parts of the content. Since large documents cannot be embedded as one big piece, I split them into smaller sections. Each chunk contains the actual text, an embedding vector, metadata, and a link that tells the system which Source it came from.

This structure makes searching much more accurate and efficient.

In the final version, a real vector database or Postgres with pgvector would be used. For the prototype, I stored everything in memory to keep the implementation lightweight and fast.

4. Temporal Querying Support

The system supports time-based questions such as "What did I do last week?" or "What were the notes from last Tuesday's meeting?" To make this possible, every Source and every Chunk has timestamps.

There are two important timestamps:

CreatedAt, which records when the data was added to the system.

ContentTime, which represents when the information actually happened. For example, the date of a meeting.

When a user asks a question that mentions a time period, the system identifies the time range and filters the stored chunks based on their timestamps before doing semantic search. This allows the user to search their “memory” by time.

5. Scalability and Privacy Considerations

Scalability:

The prototype stores everything in memory, but the design supports scaling up to thousands of documents. A larger version of the system would use a real vector database, background processing for large files, and techniques like approximate nearest neighbour search for faster similarity matching.

Privacy:

There are two possible ways to deploy the system. A cloud version stores everything on a server, which makes updates easy but requires strong security. A local-first version stores everything directly on the user’s device, providing maximum privacy. The architecture supports either approach.

Part 2 – Backend Implementation

The backend is built using Node.js and Express. I implemented two major endpoints: one for ingestion and one for querying.

Ingestion endpoint:

The ingestion endpoint takes text as input, splits it into chunks, generates embeddings, and stores the chunks with metadata.

Query endpoint:

When the user asks a question, the backend embeds the question and compares it with all stored chunks. It selects the most relevant chunks and returns them as the answer. The system also has a fallback method in case an AI model is not available, ensuring that the user always receives something meaningful.

Part 3 – Frontend Implementation

The frontend is a simple React application created with Vite. It has a basic chat interface where the user can load example data or type their own questions.

When the user types a question, the frontend sends it to the backend and displays the response. The interface is intentionally simple so that the focus of the assignment stays on the system design and the retrieval flow.

Conclusion

This prototype demonstrates the core idea of a second brain: take in information from different sources, process it, store it in a structured way, understand relationships through embeddings, and answer questions naturally.

While this version focuses on text ingestion and semantic retrieval, the overall design makes it easy to add more features such as audio transcription, PDF parsing, or more advanced LLM answers. The system is scalable, flexible, and built to grow into a more complete product in the future.

ADDITIONAL NOTE ON IMPLEMENTATION LIMITS

For this prototype, I focused on building the core foundation: ingestion of text, chunking, embedding, storage, semantic search, and question answering. These pieces are the most important parts of the assignment and they represent how the complete system would behave.

I did not implement every single modality described in the design, such as full PDF parsing, audio transcription, or automated web scraping. These features are fully accounted for in the architecture and can be added without changing the overall design. The system is already built in a way that makes it straightforward to plug in additional processors for audio files, documents, or images when needed.

The goal of this assignment was to demonstrate that the design is capable of supporting all these features and that the ingestion and retrieval pipeline is strong enough to handle them. The working prototype shows the core logic in action, and the additional modalities can be implemented as natural extensions of the same pipeline.

GITHUB REPOSITORY:

https://github.com/pavankandulalmn-ctrl/TwinMind2

Railway And Vercel sites:

Live Backend API:

https://twinmind2-production.up.railway.app/api/health

Live Frontend Demo:

https://twin-mind2.vercel.app/

Demo Walkthrough

https://www.loom.com/share/1b07dcefed1f4999bd5108ad731986d6