# Full-Stack & AI Systems Design Engineer: Take-Home Assignment

## Part 1: System Design & Architecture

### Part 1.1 : Multi-Modal Data Ingestion Pipeline

**Objective :**

The goal of the ingestion pipeline is to reliably accept, process, normalize, and store information from multiple data modalities so that all user knowledge—regardless of source—can be indexed and queried in a unified manner.

The pipeline is designed to be:

- **Asynchronous** (non-blocking ingestion)
- **Extensible** (easy to add new modalities)
- **Fault-tolerant** (errors isolated per modality)
- **Metadata-rich** (to support temporal and semantic queries)

## High-Level Architecture

Each modality follows a **common ingestion contract**, ensuring consistent downstream processing.

Input Source

↓

Modality-Specific Processor

↓

Text Normalization Layer

↓

Semantic Chunking Engine

↓

Embedding Generator

↓

Metadata + Vector Storage

This design decouples ingestion from retrieval, enabling independent scaling and future upgrades.

## Modality-Specific Pipelines

### A. Audio Ingestion (Speech → Knowledge)

**Input:**

- Audio files (.mp3, .m4a, .wav)

**Processing Steps:**

1. Audio is uploaded via API and stored in raw form.
2. A background task triggers speech-to-text transcription.
3. Transcription output is normalized (punctuation, casing).
4. Text is segmented into semantically meaningful chunks.
5. Each chunk is embedded and stored with metadata.

**Key Metadata Stored:**

- source_type: audio
- original_filename
- duration
- transcription_timestamp
- ingestion_time

**Justification:**
Audio captures meetings, voice notes, and spontaneous thoughts. Converting audio into searchable text ensures no knowledge remains inaccessible due to modality constraints.

**B. Document Ingestion (PDF / Markdown)**

**Input:**

- PDF files
- Markdown files

**Processing Steps:**

1. Document text is extracted using format-aware parsers.
2. Structural hints (headings, sections) are preserved.
3. Text is cleaned and normalized.
4. Content is chunked using semantic boundaries rather than fixed size.
5. Chunks are embedded and indexed.

**Key Metadata Stored:**

- source_type: document
- file_type
- document_title
- page_number
- upload_timestamp

**Justification:**
Documents are often long-form and structured. Preserving hierarchy improves retrieval quality and enables more accurate summarization and contextual responses.

## C. Web Content Ingestion (URL-Based)

**Input:**

- Web URLs

**Processing Steps:**

1. Web page content is fetched and cleaned.
2. Boilerplate (ads, navigation) is removed.
3. Main article text is extracted.
4. Text is chunked and embedded.
5. Source URL and fetch timestamp are recorded.

**Key Metadata Stored:**

- source_type: web
- url
- page_title
- fetch_timestamp

**Justification:**
Web content is dynamic and time-sensitive. Storing fetch timestamps ensures accurate temporal reasoning and avoids stale knowledge.

**D. Plain Text Ingestion (Notes)**

**Input:**

- Raw text notes

**Processing Steps:**

1. Text is directly normalized.
2. Chunking and embedding are applied.
3. Notes are indexed with minimal latency.

**Key Metadata Stored:**

- source_type: text
- creation_timestamp

**Justification:**
This enables quick capture of ideas and thoughts without friction, preserving cognitive flow for the user.

**E. Image Ingestion (Searchable via Metadata)**

**Input:**

- Image files (.png, .jpg)

**Processing Steps:**

1. Images are stored in object storage.
2. Captions or OCR-extracted text is generated.
3. Descriptive metadata is created.
4. Associated text is embedded for semantic search.

**Key Metadata Stored:**

- source_type: image
- caption_text
- upload_timestamp

**Justification:**
While images themselves are not directly searchable, associating semantic text enables retrieval and contextual reasoning.

**Unified Data Representation**

After processing, **all modalities are converted into a unified representation**:

Code :

```
{
  "content_chunk": "string",
  "embedding_vector": [...],
  "source_type": "audio | document | web | text | image",
  "source_reference": "file_id | url",
  "timestamp": "ISO-8601",
  "user_id": "uuid"
}
```

This abstraction allows the retrieval system to operate independently of the original data format.

## Design Trade-Offs

| Decision | Reasoning |
|---|---|
| Async ingestion | Prevents UI blocking and improves UX |
| Chunk-based indexing | Improves retrieval precision |
| Modality normalization | Enables unified querying |
| Metadata-first design | Supports temporal and filtered queries |

## Summary

The ingestion pipeline transforms heterogeneous user data into a consistent, searchable knowledge representation. By separating modality-specific processing from downstream indexing, the system remains scalable, extensible, and resilient—laying a strong foundation for intelligent retrieval and reasoning.

## Part 1.2 – Information Retrieval & Querying Strategy

**Objective** :

The primary objective of the retrieval system is to accurately identify the most relevant pieces of information from a large, heterogeneous knowledge base in response to a user's natural language query.

Unlike traditional keyword-based systems, a "Second Brain" must:

- Understand **semantic intent**
- Handle **vague or conversational queries**
- Respect **temporal constraints**
- Synthesize information across multiple sources

To achieve this, a **hybrid retrieval strategy** is employed.

### Retrieval Strategy Overview

The system uses a **Hybrid Retrieval Architecture**, combining:

1. **Semantic Search (Vector Embeddings)**
2. **Keyword-Based Filtering**
3. **Metadata & Temporal Constraints**
4. **LLM-Based Reasoning Layer**

This approach balances **precision, recall, interpretability, and performance**.

## Semantic Search(Primary Retrieval Mechanism)

### How It Works

- Each content chunk is converted into a high-dimensional embedding vector.
- User queries are also embedded using the same model.
- Similarity search retrieves the top-K semantically closest chunks.

### Why Semantic Search is Required

Keyword search fails when:

- The user paraphrases earlier content
- Synonyms or abstract concepts are used
- The query is conversational or implicit

**Example:**

“What were the main blockers discussed in last week’s meeting?”

The original text may never contain the word *“blockers”*, but semantic search still retrieves the correct context.

## Keyword-Based Search(Supporting Layer)

While semantic search is powerful, keyword-based search is useful for:

- Exact term matching
- Technical identifiers (IDs, filenames, names)
- Precision filtering

This is implemented using:

- Full-text indexing over normalized content

**Why not keyword-only search?**

- Poor semantic understanding
- Fails on paraphrased queries

**Why not semantic-only search?**

- May miss exact matches or structured data

Hence, **hybrid retrieval** is chosen.

## **Metadata and Temporal Filtering**

Before similarity ranking, retrieved candidates are filtered using metadata:

- source_type (audio, document, web, etc.)
- timestamp
- user_id

This enables queries like:

- *“What did I work on last month?”*
- *“Summarize notes from yesterday’s meeting.”*

Temporal constraints significantly reduce search space and improve relevance.

**Retrieval Pipeline**

User Query

↓

Query Embedding

↓

Temporal & Metadata Filtering

↓

Hybrid Retrieval (Vector + Keyword)

↓

Top-K Context Selection

↓

Context Ranking & Deduplication

↓

LLM Reasoning & Answer Generation

Each stage is independently tunable and observable.

## **Context Window Management**

To avoid overwhelming the LLM:

- Retrieved chunks are ranked by relevance
- Redundant or overlapping chunks are removed
- A token budget is enforced

This ensures:

- Faster responses
- Lower cost
- Higher answer accuracy

LLM Integration

The LLM is **not** used to search data.
Instead, it is responsible for:

- Synthesizing retrieved context
- Resolving ambiguities
- Producing concise, human-like responses

This separation:

- Improves determinism
- Reduces hallucinations
- Makes the system auditable

## Design Trade-Offs

| Approach | Pros | Cons |
|---|---|---|
| Keyword-only | Fast, cheap | Poor Semantic understanding |
| Semantic-only | Context-aware | Misses exact matches |
| Hybrid(Chosen) | Best balance | Slightly more complex |

## Summary

By combining semantic similarity, keyword precision, and metadata filtering, the retrieval system achieves high relevance, robustness, and flexibility. This hybrid strategy forms the cognitive core of the “Second Brain,” enabling accurate reasoning across time, modality, and context.

## Part 1.3 : Data Indexing & Storage Model

**Objective** :

The storage and indexing layer is responsible for persisting raw data, processed content, embeddings, and metadata in a manner that supports:

- Efficient semantic retrieval
- Temporal queries
- Scalability to thousands of documents per user
- Flexibility across multiple data modalities

To meet these goals, a **multi-store architecture** is adopted.

**Data Lifecycle Overview :** This ensures consistent treatment of all modalities.

Raw Input

↓

Modality Processing

↓

Text Normalization

↓

Semantic Chunking

↓

Embedding Generation

↓

Indexed Storage (Metadata + Vectors)

Chunking Strategy

**Why Chunking is Required**

Large documents or transcripts cannot be indexed or reasoned about effectively as a single unit. Chunking:

- Improves retrieval granularity
- Reduces noise
- Optimizes LLM context usage

**Chunking Method**

- **Semantic chunking** using paragraph and section boundaries
- Chunk size: ~300–500 tokens
- Overlap: 10–20% between adjacent chunks

This balances context preservation with retrieval precision.

Storage Components

## A. Relational Database (Metadata & Control Plane)

**Technology:** PostgreSQL

**Stored Data:**

- Document metadata
- Chunk references
- Timestamps
- User associations

**Example Schema:**

**`Documents`**

- `id (uuid)`
- `user_id (uuid)`
- `source_type (enum)`
- `source_reference (string)`
- `created_at (timestamp)`

**`content_chunks`**

- `id (uuid)`
- `document_id (uuid)`
- `chunk_text (text)`
- `chunk_index (int)`
- `timestamp (timestamp)`

**B. Vector Database (Semantic Index)**

**Technology Options:**

- FAISS (local)
- Chroma (developer-friendly)

**Stored Data:**

- Embedding vectors
- Chunk IDs
- Lightweight metadata

This enables fast similarity search over high-dimensional vectors.

**C. Optional Full-Text Index**

**Purpose:**

- Keyword filtering
- Exact term matching

Implemented using:

- PostgreSQL Full-Text Search

## Metadata Design

Each chunk is enriched with metadata:

```
{
  "user_id": "uuid",
  "source_type": "audio | document | web | text | image",
  "source_reference": "file_id | url",
  "created_at": "timestamp",
  "chunk_index": 3
}
```

This metadata enables:

- Temporal filtering
- Source-specific queries
- User-level isolation

## Indexing Strategy

| Index Type | Purpose |
|---|---|
| Vector index | Semantic similarity |
| Timestamp index | Time-based queries |
| Full-text index | Keyword search |

All indices are updated asynchronously to ensure ingestion performance.

## Trade-Off Analysis

### SQL vs NoSQL vs Vector DB

| Storage | Pros | Cons |
|---|---|---|
| SQL(Postgres) | Strong consistency, time queries | Not ideal for embeddings |
| NoSQL | Flexible Schema | Weak relational guarantees |
| Vector DB | Fast Semantic search | Limited transactional support |

**Chosen Approach:**

- SQL for metadata & control
- Vector DB for semantic retrieval

This separation ensures performance without sacrificing query flexibility.

Scalability Considerations

- Chunk-based indexing scales linearly
- Vector search supports ANN (Approximate Nearest Neighbor)

- User data is logically isolated by user_id
- Horizontal scaling possible by sharding vector indices

**Summary**

The hybrid storage model enables efficient semantic retrieval, accurate temporal reasoning, and scalable growth. By separating metadata management from vector indexing, the system remains flexible, performant, and maintainable.

## Part 1.4 : Temporal Querying Support

**Objective :**

A defining capability of the “Second Brain” system is its ability to reason over time. Users must be able to query their knowledge base using temporal constraints such as:

- *“What did I work on last month?”*
- *“Summarize notes from last Tuesday’s meeting.”*
- *“What were the key concerns raised earlier this year?”*

To support this, temporal metadata is treated as a **first-class concern** throughout the ingestion, storage, and retrieval pipeline.

### Timestamp Association

Every ingested data item is associated with **one or more timestamps**, depending on modality:

| Modality | Timestamp Source |
|---|---|
| Audio | Recording time / ingestion time |
| Documents | Upload time / document creation time |
| Web Content | Fetch time |

| | | |
|---|---|---|
| Plain Text | Creation time | |
| Images | | Upload time |

These timestamps are stored at:

- **Document level**
- **Chunk level**

This enables both coarse and fine-grained temporal filtering.

<u>**Temporal Indexing**</u>

Timestamps are indexed in the relational database using standard time-based indices.

This allows efficient filtering such as:

- Last N days
- Specific date ranges
- Relative periods (week, month, year)

## Query-Time Temporal Reasoning

### Step 1: Temporal Constraint Extraction

The user’s natural language query is analyzed to detect temporal intent:

- Absolute (e.g., *“March 2024”*)
- Relative (e.g., *“last week”*)

This can be implemented using:

- Rule-based parsing
- LLM-assisted query understanding

### Step 2: Candidate Filtering

Before semantic retrieval:

- Content outside the temporal range is excluded
- Search space is significantly reduced

### Step 3: Context Ranking

Among temporally valid chunks:

- Semantic similarity determines relevance
- More recent information can be weighted higher

**Example Flow**

**Query:**

*"What did I work on last month?"*

**System Flow:**

1. Detect temporal intent → last 30 days
2. Filter chunks by timestamp
3. Perform semantic retrieval
4. Aggregate and summarize using LLM

**<u>Design Trade-Offs</u>**

| Approach | Pros | Cons |
|---|---|---|
| Time filtering before retrieval | Faster, precise | Requires good metadata |
| Time filtering after retrieval | Flexible | Less efficient |

**Chosen Approach:**

- Temporal filtering **before** semantic retrieval

Summary

By attaching timestamps at ingestion time and enforcing temporal constraints during retrieval, the system enables accurate time-aware reasoning. This capability is essential for transforming raw memory into meaningful, contextual knowledge.

## Part 1.5 Scalability and Privacy

### Scalability Considerations

The system is designed to scale efficiently as a user’s knowledge base grows from a handful of items to **thousands of documents, audio files, and web sources**.

### Horizontal Scalability

- **Asynchronous ingestion** prevents blocking operations.
- Ingestion and retrieval services can scale independently.
- Vector indices support approximate nearest neighbor (ANN) search for large datasets.

### Data Growth Management

- Chunk-based indexing ensures linear growth.
- Metadata-driven filtering reduces retrieval scope.
- Old or low-priority data can be archived without impacting retrieval.

### Retrieval Performance

- Temporal filtering significantly reduces candidate set size.
- Hybrid retrieval balances speed and relevance.
- Top-K retrieval limits LLM context size.

## Privacy by Design

Privacy is treated as a **core architectural principle**, not an afterthought.

**User Data Isolation**

- All data is logically isolated using user_id.
- Retrieval queries are always scoped to a single user.
- No cross-user embedding access is permitted.

Cloud-Hosted vs Local-First Trade-Offs

**Cloud-Hosted Approach**

**Pros:**

- High availability
- Easy scalability
- Managed LLM services

**Cons:**

- User data leaves local device
- Higher privacy risk

**Local-First / Hybrid Approach**

**Pros:**

- Strong privacy guarantees

- Offline access possible
- Full user control

**Cons:**

- Limited compute
- Slower embedding & transcription
- Harder to maintain

**Chosen Design Philosophy**

The system is designed to support:

- **Cloud-first deployment** for rapid development and scalability
- **Local-first compatibility** for privacy-sensitive users

This hybrid-friendly design enables future migration without architectural changes.

## <u>Security Measures</u>

- storage at rest
- Secure API access
- Minimal data retention outside user scope
- No training of global models on user data

## Summary

- By combining asynchronous processing, chunk-based indexing, and strict user-level isolation, the system scales efficiently while preserving privacy. The architecture balances performance, flexibility, and trust—key requirements for a personal AI companion.