Software Design Document (SDD)

Project: Personal AI Companion – "Second Brain"
Version: 1.0
Date: 26/11/25
Authors: Dhruv Temura

1. INTRODUCTION

1.1 Purpose

This Software Design Document (SDD) describes the architecture, system design, data model, and operational behavior of the Personal AI Companion ("Second Brain"). It serves as the primary reference for developers implementing the backend, frontend, and ingestion pipeline.

1.2 Scope

The Personal AI Companion enables users to store, manage, and retrieve personal knowledge using three primary modalities:

- Normal chat input (text)
- Audio ingestion (via transcription)
- Document ingestion (PDF/text extraction)

It supports context-aware conversational retrieval via semantic search, metadata filtering, and temporal querying.

1.3 Overview

This SDD covers:

- High-level architecture
- Multi-modal ingestion pipeline
- Retrieval and querying strategy
- Temporal reasoning
- Complete data schema
- Component-level behavior

- User interface structure
- Non-functional requirements

1.4 Reference Material

- Project Requirements Document
- IEEE Std 1016 Recommended Practice for Software Design Descriptions
- Vector Database Documentation (pgvector/Qdrant)
- Whisper Transcription Model Docs

1.5 Definitions & Acronyms

- **LLM** – Large Language Model
- **RAG** – Retrieval-Augmented Generation
- **API** – Application Programming Interface
- **DB** – Database
- **Chunk** – Split unit of text for embedding
- **Embedding** – Vector representation of text

---

2. SYSTEM OVERVIEW

The Personal AI Companion is a multi-modal knowledge system that allows users to upload and store various forms of personal information and retrieve insights conversationally.

The system provides:

- Multi-modal ingestion (text, audio, documents)
- Automated transcription/extraction
- Chunking and embedding of processed text
- LLM reasoning to generate responses
- Vector search for semantic retrieval
- Temporal reasoning based on stored timestamps
- Chat interface for natural interaction

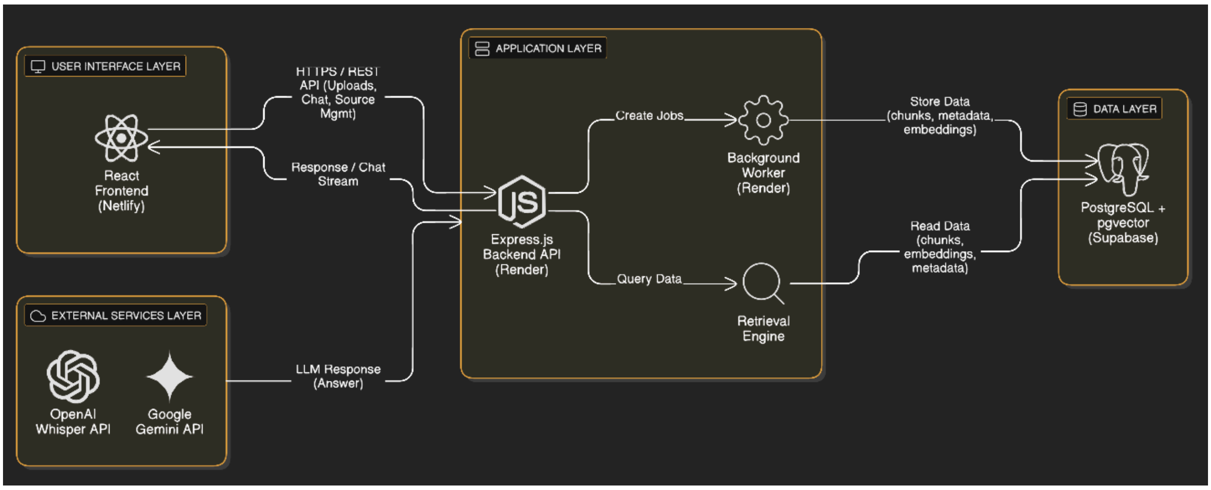
This system functions as a personal memory assistant, allowing users to store information and receive conversational answers to queries via an integrated LLM.

---

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

3.1.1 High-Level Architecture


USER INTERFACE LAYER
React Frontend (Netlify)
HTTPS / REST API (Uploads, Chat, Source Mgmt)
Response / Chat Stream
APPLICATION LAYER
Express.js Backend API (Render)
Create Jobs
Background Worker (Render)
Query Data
Retrieval Engine
Store Data (chunks, metadata, embeddings)
Read Data (chunks, embeddings, metadata)
DATA LAYER
PostgreSQL + pgvector (Supabase)
EXTERNAL SERVICES LAYER
OpenAI Whisper API
Google Gemini API
LLM Response (Answer)

Components:

- **Frontend Web App** (React)
- **Backend API Service** (Node/Express or Python FastAPI)
- **Async Ingestion Worker**
- **Metadata Database** (Postgres/MongoDB)
- **Vector Database** (pgvector/Qdrant)
- **Object Storage** (for audio + raw documents)
- **LLM Integration Layer**

3.1.2 Multi-Modal Ingestion Pipeline

The system supports three ingestion types:

- **Text notes** → stored directly
- **Audio files** → transcribed using Whisper → normalized
- **Documents (PDF/TXT)** → text extracted using a document parser
- **Images (Proposed Method)** → Images (Proposed Implementation) → store image metadata in the database with user-provided text descriptions or auto-generated captions. The description text would be chunked and embedded like other content, making images searchable through their textual context.

- **Web Content (URL)(Proposed Method)** → fetch the page HTML, extract the main readable text using an HTML parsing library (e.g., Cheerio), then pass extracted text through the same chunk → embed → store pipeline

Pipeline Flow:

1. **User uploads/submits file or text -** The user inputs, through the frontend (uploads an audio file, Uploads a PDF/text document, Types a note). This request is sent to the backend through the API. At this point, the system does **not** process the file immediately. It only receives it and prepares it for ingestion.

2. **API creates ingestion job -** The backend, processes the raw file in memory, creates a record in the jobs table, and assigns (job_id, user_id, source_type, status = queued). This allows ingestion to happen asynchronously. This ensures the user doesn't wait for transcription or PDF extraction in real time.

   Files are processed directly in memory during ingestion rather than persisted to separate object storage, simplifying the architecture while maintaining all processing capabilities.

3. **Job added to processing queue -** The ingestion job status is set to 'queued' in the database. A background worker continuously polls the jobs table for pending work, picking up queued jobs for processing. This database-driven queue approach ensures reliable job processing without requiring separate queue infrastructure.

   Why a queue?

   - Audio transcription may take 5–40 seconds
   - PDF extraction may take time
   - Embedding generation may also take time

   A queue ensures:

   - Jobs are processed in the background
   - API stays fast and responsive
   - Multiple jobs can be handled by workers

4. **Worker processes job based on modality** - A background **Ingestion Worker** continuously listens to the queue. When it picks up a job,
   - If audio → it sends the file to Whisper for transcription
   - If document → extracts clean text from PDF/TXT
   - If text → normalizes the text

   The worker acts like a “factory machine” that runs the heavy processing.

5. **Text is chunked (500–1000 tokens)** - Once the worker obtains usable text, It splits the text into small pieces called **chunks** (~500–1000 tokens).

   Why chunking?

   - LLM embedding models have token limits
   - Small chunks make retrieval more accurate
   - Prevents mixing unrelated text together

   Each chunk gets:
   - `chunk_id`
   - `source_id`
   - `index`
   - `chunk_timestamp`
   - `user_id`

6. **Embeddings generated for each chunk -**
   For each chunk:
   - The worker sends the chunk text to the **embedding API**
   - Receives a **vector representation (embedding)**

   This allows the system to:
   - Perform **semantic search**
   - Retrieve relevant text based on meaning, not keywords

7. **Metadata + vectors stored -**

   All data is stored in a single PostgreSQL database:

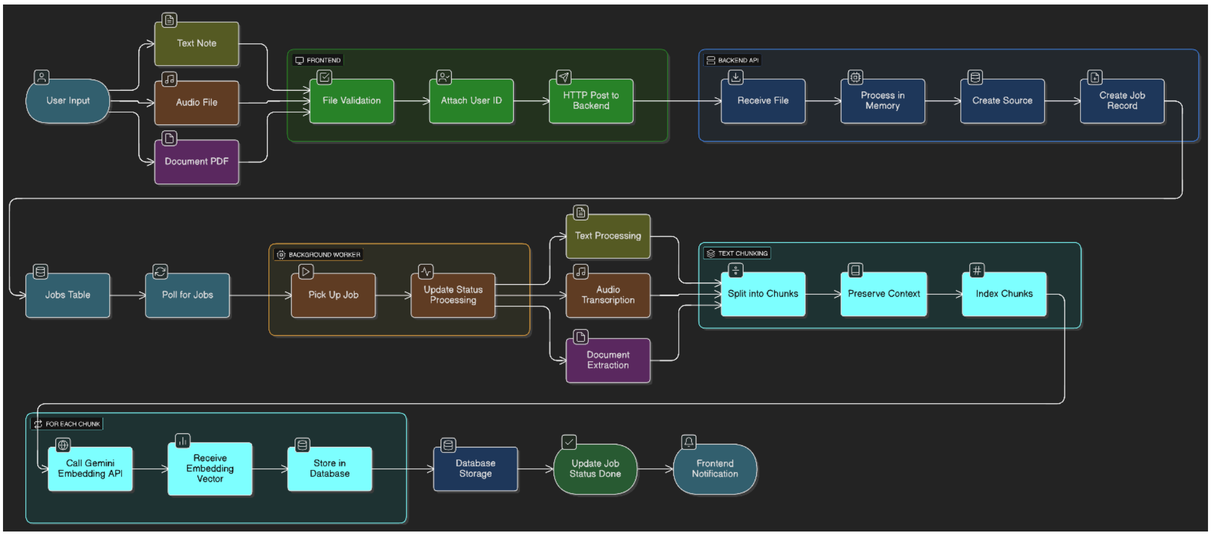
   Chunks table stores:

     - chunk text
     - timestamps
     - user_id
     - source information
     - embedding vector (using pgvector VECTOR type)
   - The unified database approach:
     i. Simplifies deployment
     ii. Maintains atomic transactions
     iii. Enables efficient joins between metadata and vectors

8. **Job marked completed** - After all chunks are stored:
   - The job status becomes `done`
   - The frontend updates the user that ingestion is complete
   - The content becomes **immediately searchable** in chat

   If an error occurred, status becomes `failed`.

Multi-Modal Ingestion Pipeline



3.1.3 Chunking & Embedding Architecture

- Text is divided into manageable chunks (typically 500–1000 tokens) to stay within embedding model limits
- A small overlap is applied between chunks to preserve sentence continuity across boundaries
- Each chunk is embedded using the LLM embedding model and stored for semantic retrieval

Fields stored:

- `chunk_id`
- `source_id`
- `index`
- `chunk_timestamp`
- `user_id`
- `text`
- `embedding`

Chunking ensures long documents and transcripts become searchable at a fine-grained level, improves retrieval accuracy, and prevents loss of meaning at chunk boundaries.

3.1.4 Vector Indexing & Storage Layer

Single PostgreSQL database with pgvector extension handles both:

- **Metadata storage** – user info, sources, timestamps, job records
- **Vector embeddings** – stored using pgvector for semantic search

This unified approach simplifies deployment while maintaining full functionality for semantic retrieval and structured queries.

Metadata filters ensure:

- Only results from the correct user are retrieved
- Temporal constraints applied before vector search

The separation of metadata storage and vector embeddings ensures faster semantic search while keeping structured information easily queryable and maintainable.

3.1.5 Retrieval & Query Reasoning Engine

The retrieval engine is responsible for finding the most relevant stored information for a given user query and preparing it as context for the LLM.

Retrieval components:

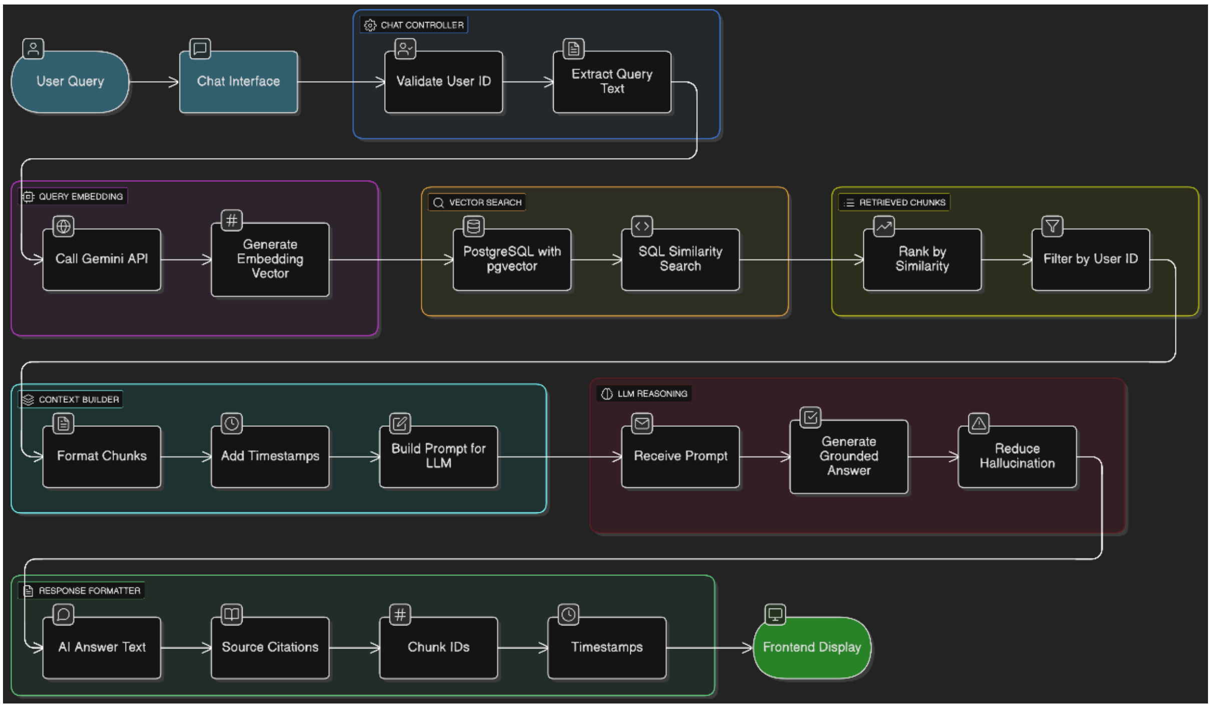
- **Semantic Search** → uses vector similarity over chunk embeddings to find meaningfully related text.
- **Metadata Filtering** → restricts results to the current `user_id` so users never see each other's data.
- **Temporal Filtering** → optionally narrows results based on parsed date expressions in the query (e.g., "yesterday", "last week").

Retrieval Steps:

1. Parse the user query.
2. Detect and resolve any temporal constraints into a date range.
3. Generate an embedding for the query.
4. Perform vector search in the vector DB with `user_id` and optional time filters.
5. Select the top-K relevant chunks.
6. Pass these chunks plus the original query to the LLM, which generates the final answer.

If no sufficiently relevant chunks are found, the LLM is instructed to respond transparently (e.g., indicating that it has no stored information related to the query) instead of hallucinating.

Retrieval & Query Reasoning Engine



3.1.6 LLM Orchestration

The LLM orchestration layer prepares the inputs required for generating an accurate and context-aware response.

The LLM receives:

- The user query
- The top-retrieved chunks
- Associated timestamps and metadata

The prompt builder formats these components into a structured prompt.

Prompt structure ensures:

- The LLM answers only using the provided context
- Hallucinations are minimized
- Temporal constraints are respected
- The final answer is grounded in retrieved user data

If the retrieved chunks do not sufficiently match the query, the LLM is instructed to respond transparently rather than infer unsupported information.

3.1.7 Temporal Querying Support

The system supports natural language time references so users can retrieve information based on when it was added or when an event occurred.

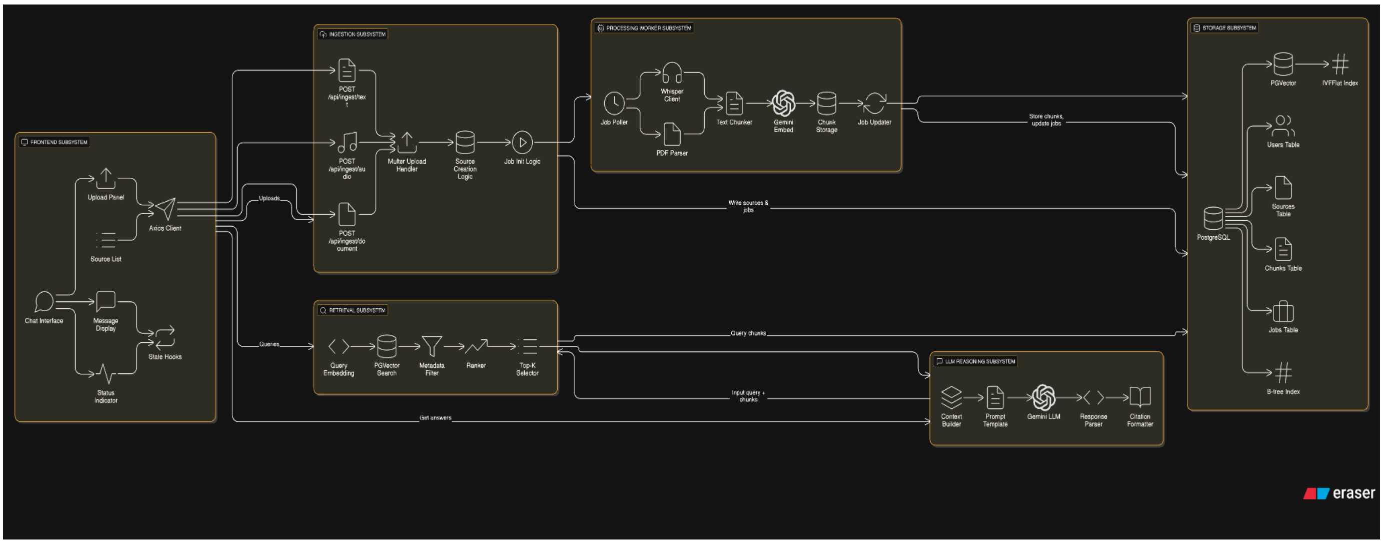
System detects expressions like:

- "yesterday"
- "last week"
- "on 5th March"

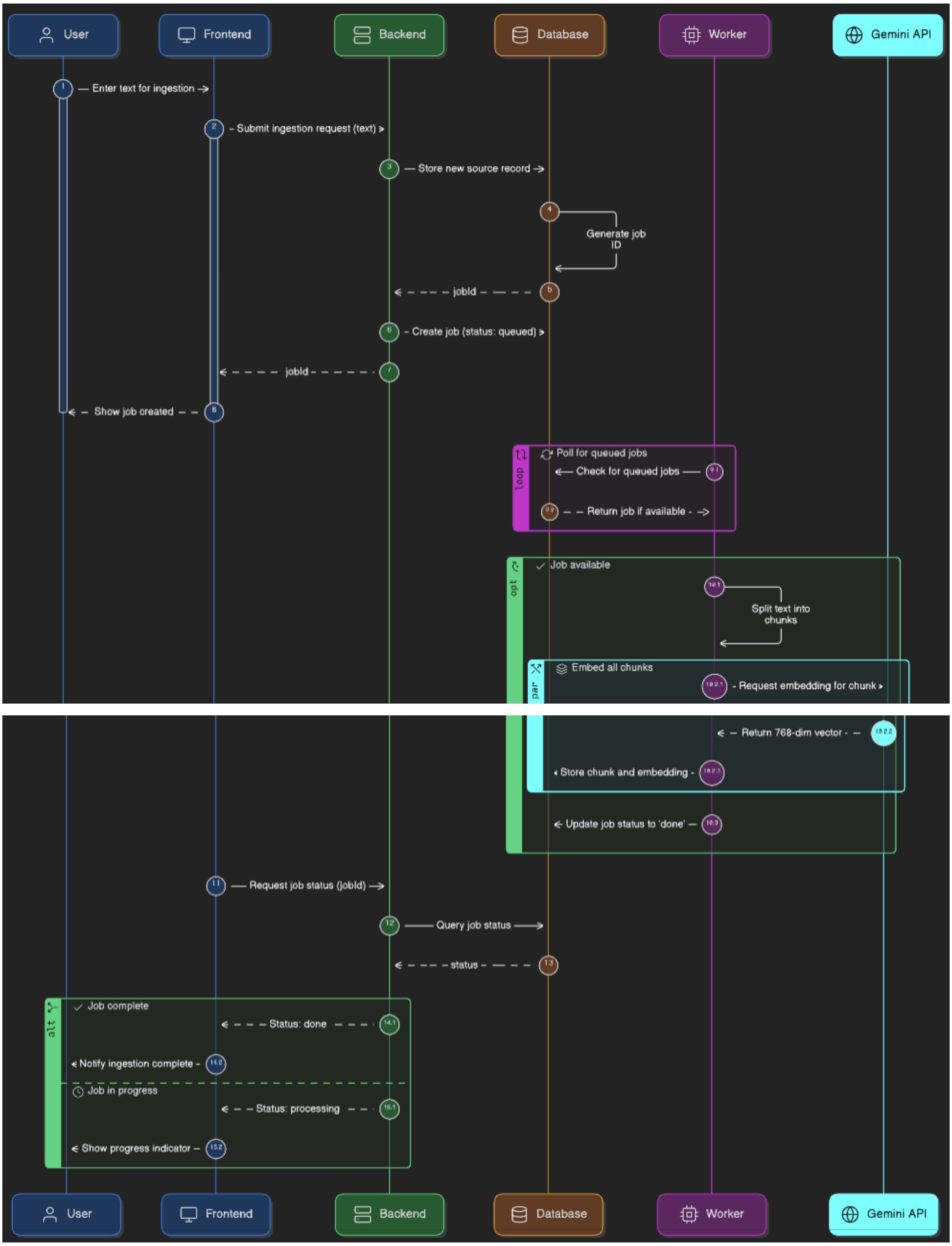
These expressions are converted into concrete date ranges using the temporal parser.
Temporal filtering is applied before vector search, so only chunks whose timestamps fall within the identified range are considered.
If the query contains no temporal expression, the system performs normal semantic retrieval without time filtering.

3.2 Decomposition Description

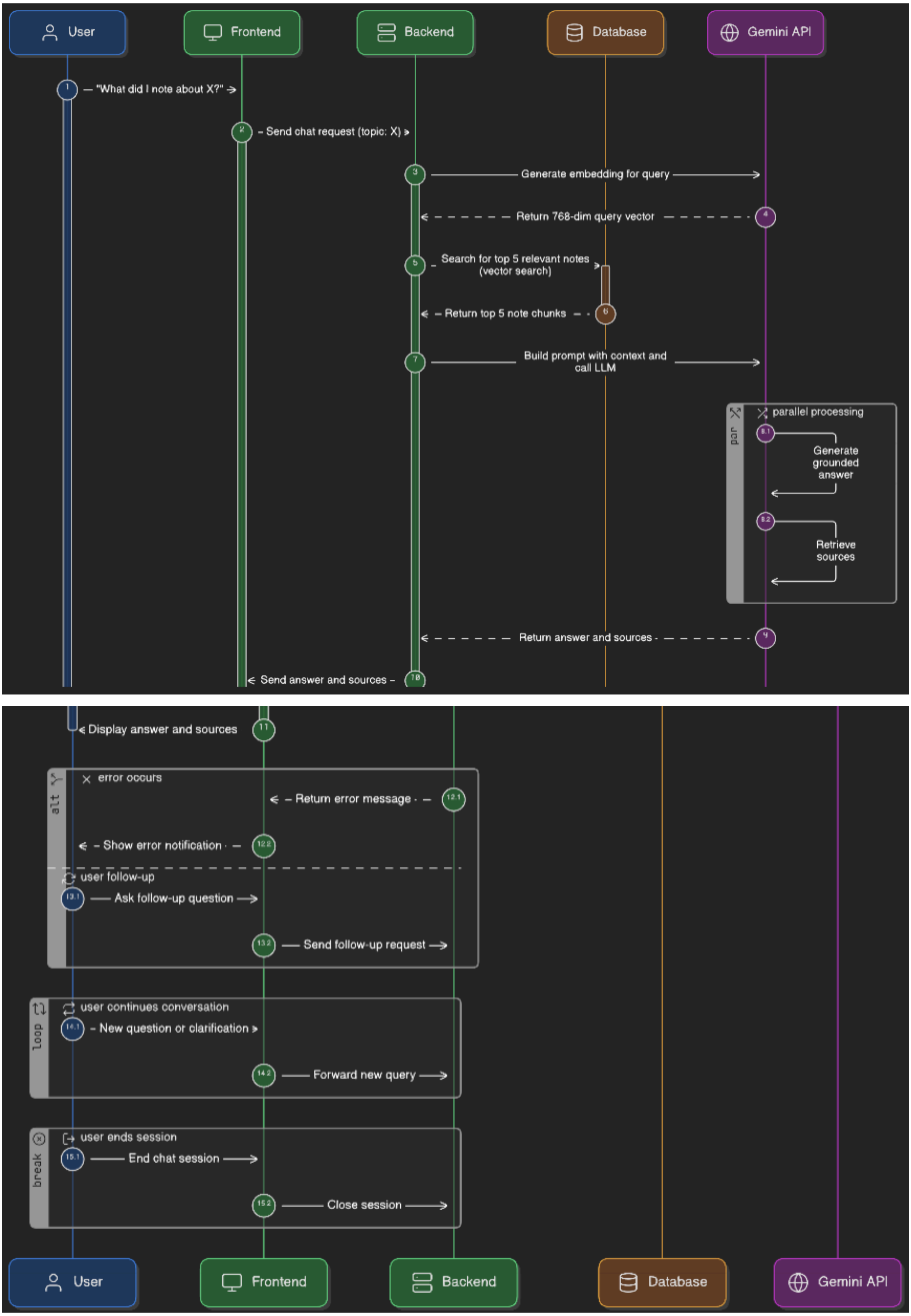
System Decomposition (6 Subsystems)



SEQUENCE DIAGRAM: Text Ingestion Flow

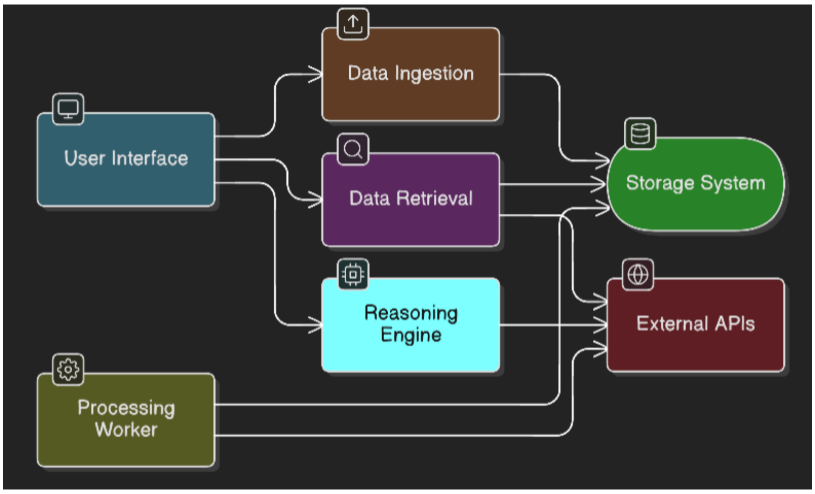


SEQUENCE DIAGRAM: Chat Query Flow



Subsystems:

- **Ingestion Subsystem**
    - Handles file uploads and text input
    - Creates ingestion jobs in database
    - Returns job IDs to frontend for status tracking
- **Processing Worker Subsystem**
    - Polls jobs table for queued work
    - Processes audio files (Whisper transcription)
    - Extracts text from PDFs
    - Chunks text into manageable segments
    - Generates embeddings via Gemini API
    - Stores chunks and embeddings in database
    - Updates job status
- **Retrieval Subsystem**
    - Accepts user queries
    - Generates query embeddings
    - Performs vector similarity search
    - Applies user_id filtering for security
    - Returns ranked relevant chunks
- **LLM Reasoning Subsystem**
    - Constructs prompts with retrieved context
    - Calls Gemini API for response generation
    - Formats and returns conversational responses
- **Frontend Subsystem**
    - Provides chat interface
    - Handles file uploads
    - Displays conversation history
    - Shows upload/processing status
- **Storage Subsystem**
    - PostgreSQL with pgvector extension
    - Stores users, sources, chunks, jobs
    - Provides vector similarity search
    - Maintains data isolation per user



3.3 Design Rationale

The system architecture is designed to balance correctness, performance, scalability, and ease of implementation within the project constraints.

Key Architectural Decisions

- **Asynchronous ingestion** → Separates heavy processing (transcription, document parsing, embedding generation) from user-facing interactions. This prevents slowdowns in the chat interface and ensures consistent user experience.
- **Chunking** → Breaking extracted text into smaller segments improves retrieval accuracy and ensures embedding models stay within token limits. It also increases granularity during semantic search.
- **Vector database** → Enables fast, meaningful retrieval based on semantic similarity rather than keyword matching. This forms the core of the conversational memory system.
- **Temporal metadata** → Allows the system to support natural language time-based queries (e.g., "last week", "yesterday"), enhancing the assistant's ability to reason over the user's timeline of information.

Scope Decisions & Simplifications

Certain capabilities were intentionally excluded to keep the solution focused, stable, and implementable within the assignment timeframe:

- No BM25/keyword search component (semantic retrieval is sufficient for the project's goal).
- No image ingestion or web scraping.
- No re-ranking or advanced hybrid retrieval.

These exclusions allow the system to remain clean and robust while meeting all required functional goals.

Technology & Storage Rationale

- **Node/Express Backend** → Chosen for familiarity (MERN stack), rapid development, and strong support for I/O-heavy workflows such as file uploads and API calls. Its non-blocking architecture aligns naturally with asynchronous ingestion tasks.

- Single **PostgreSQL** database with **pgvector** extension handles both:
  - Metadata storage – user info, sources, timestamps, job records
  - Vector embeddings – stored using pgvector for semantic search

This unified approach simplifies deployment while maintaining full functionality for semantic retrieval and structured queries.

Both databases are essential for the ingestion, retrieval, and temporal reasoning components. The separation maintains clarity between structured metadata and high-dimensional semantic search, improving maintainability and performance.

React Frontend - Provides a modern, component-based UI with efficient state management for real-time chat interactions and file upload handling.

Google Gemini API - Serves dual purpose as both embedding generator (text-embedding-004) and conversational LLM (gemini-1.5-flash), providing high-quality semantic search and response generation.

OpenAI Whisper API - Industry leading speech-to-text transcription with high accuracy across multiple languages and audio qualities.

---

4. DATA DESIGN

4.1 Data Description

The system stores:

- User accounts
- Source records (audio, documents, text)
- Raw files in object storage
- Chunked text + embeddings
- Job records for asynchronous ingestion
- Timestamps for temporal filtering

4.2 Data Dictionary

Users

Field	Type	Description
user_id	UUID	Unique identifier
name	String	User name
email	String	Login credential

Sources

Represents each ingestion item.

Field	Type	Description
source_id	UUID	Unique identifier
user_id	UUID	Owner
source_type	Enum(text, audio, document)	Modality
title	String	Extracted or user-provided
raw_location	String	File path or stored object key
created_at	DateTime	Upload timestamp
source_timestamp	DateTime	Time referenced in content (if available)

Chunks

Field	Type	Description
chunk_id	UUID	Unique identifier
source_id	UUID	Link to source
user_id	UUID	Owner
text	Text	Chunk content
embedding	Vector	Embedding vector
chunk_timestamp	DateTime	Derived timestamp
index	Int	Order of chunk

Jobs

Field	Type	Description
job_id	UUID	Job identifier
user_id	UUID	Owner
status	Enum(queued, processing, done, failed)	Job state
error_message	String	Error info if any
created_at	DateTime	Creation time
updated_at	DateTime	Update time

4.3 Database Schema

Implemented Schema:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  name VARCHAR(255),
  created_at TIMESTAMP DEFAULT NOW()
);


CREATE TABLE sources (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id),
  source_type VARCHAR(50) NOT NULL,
  title TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE chunks (
  id SERIAL PRIMARY KEY,
  source_id INTEGER REFERENCES sources(id),
  user_id INTEGER REFERENCES users(id),
  chunk_index INTEGER NOT NULL,
  text TEXT NOT NULL,
  embedding VECTOR(768),
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX ON chunks USING ivfflat (embedding vector_cosine_ops);

CREATE TABLE jobs (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id),
  source_id INTEGER REFERENCES sources(id),
  status VARCHAR(50) DEFAULT 'queued',
  error_message TEXT,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);
```

Notes:

- Uses SERIAL (auto-increment) for IDs instead of UUIDs for simplicity
- pgvector's VECTOR type stores embeddings directly in PostgreSQL
- ivfflat index on embeddings enables fast approximate nearest neighbor search
- All timestamp fields use TIMESTAMP type with automatic NOW() defaults

---

5. COMPONENT DESIGN

5.1 Ingestion Workers

The ingestion worker is a separate Node.js process that:

- Polls the jobs table at regular intervals
- Picks up jobs with status='queued'
- Updates status to 'processing' during execution
- Marks jobs as 'done' or 'failed' upon completion
- Runs independently from the API server for true async processing

Worker Process Flow:

1. Query database for jobs WHERE status = 'queued' ORDER BY created_at LIMIT 1
2. If job found, update status to 'processing'
3. Retrieve source details from sources table
4. Based on source_type:
    a. Audio: Call Whisper API for transcription

        b. Document: Use pdf-parse to extract text
        c. Text: Use text directly
5. Split text into chunks (~500-1000 tokens each)
6. For each chunk:
    a. Call Gemini embedding API
    b. Insert chunk with embedding into chunks table
7. Update job status to 'done'
8. On error: update status to 'failed', set error_message
9. Loop back to step 1

Error Handling:

- Network failures are caught and logged
- Failed jobs retain error_message for debugging
- Worker continues processing other jobs
- Jobs can be manually reset for retry

5.2 Retrieval Engine

The retrieval engine handles semantic search over stored chunks.

Process:

1. Receive user query and user_id
2. Generate query embedding using Gemini API
3. Execute vector similarity search: ```sql SELECT id, text, created_at, 1 - (embedding <=> $queryEmbedding) AS similarity FROM chunks WHERE user_id = $userId ORDER BY embedding <=> $queryEmbedding LIMIT 5; ```
4. Return top 5 most similar chunks
5. Filter results with similarity threshold (e.g., > 0.3)

Key Features:

- Uses pgvector's <=> operator for cosine distance
- Applies user_id filter for security
- Limits results to top-K for context window management
- Includes timestamps for temporal context

5.3 Temporal Parser

Current implementation stores timestamps but does not parse natural language temporal queries.

Stored temporal data:

- created_at for chunks (when content was added)

- created_at for sources (when file was uploaded)

Basic temporal filtering capability:

- Filter by date range in SQL: WHERE created_at BETWEEN $start AND $end
- Display timestamps in retrieved context

Future enhancement potential:

- Parse "yesterday" → today - 1 day
- Parse "last week" → last 7 days
- Parse explicit dates → concrete date range

5.4 Chat Controller

The chat controller orchestrates the query-response flow.
Endpoint: POST /api/chat Request Body: ```json { "message": "user query text", "userId": 1 } ``` Process: 1. Validate user authentication 2. Generate embedding for user query 3. Call retrieval engine to get relevant chunks 4. Build LLM prompt with retrieved context 5. Call Gemini API for response generation 6. Return formatted response Response: ```json { "response": "LLM generated answer", "sources": [{ "chunkId": 123, "text": "relevant chunk text", "timestamp": "2024-11-26T10:30:00Z" }] } ```

5.5 LLM Context Builder

The context builder formats retrieved chunks into an effective LLM prompt.

Template Structure:

You are a personal AI assistant with access to the user's stored information.

Context from user's knowledge base:

[Timestamp: 2024-11-26 10:30] Content: [chunk 1 text]

[Timestamp: 2024-11-25 15:45] Content: [chunk 2 text]

User question: [user query]

Instructions:

- Answer based ONLY on the provided context

- If the context doesn't contain relevant information, clearly state that
- Do not make up or infer information not present in the context
- You may reference the timestamps if relevant to the query

Key Principles:

- Clear separation between context and query
- Explicit instruction to avoid hallucination
- Timestamps included for temporal awareness
- Limited context to stay within token limits

5.6 API Layer

Backend API Endpoints:

Authentication & Users:

POST /api/register

- Create new user account
- Request: { email, name, password }
- Response: { userId, token }

POST /api/login

- Authenticate user
- Request: { email, password }
- Response: { userId, token }

Ingestion:

POST /api/ingest/text

- Submit text note
- Request: { userId, title, text }
- Response: { jobId, sourceId }

POST /api/ingest/audio

- Upload audio file
- Request: FormData with userId, title, audioFile
- Response: { jobId, sourceId }

POST /api/ingest/document

- Upload PDF/text document
- Request: FormData with userId, title, documentFile

- Response: { jobId, sourceId }

Job Tracking:

GET /api/jobs/:jobId

- Check ingestion job status
- Response: { jobId, status, errorMessage }

GET /api/jobs/user/:userId

- List all jobs for user
- Response: [{ jobId, status, createdAt, title }]

Chat:

POST /api/chat

- Submit query and get response
- Request: { userId, message }
- Response: { response, sources: [{ chunkId, text, timestamp }] }

Sources:

GET /api/sources/:userId

- List all sources for user
- Response: [{ sourceId, title, sourceType, createdAt }]

DELETE /api/sources/:sourceId

- Delete source and associated chunks
- Response: { success: true }

Middleware:

- CORS enabled for frontend access
- JSON body parsing
- File upload handling (multer)
- Error handling with proper status codes
- Request logging

---

6. HUMAN INTERFACE DESIGN

6.1 Overview of User Interface

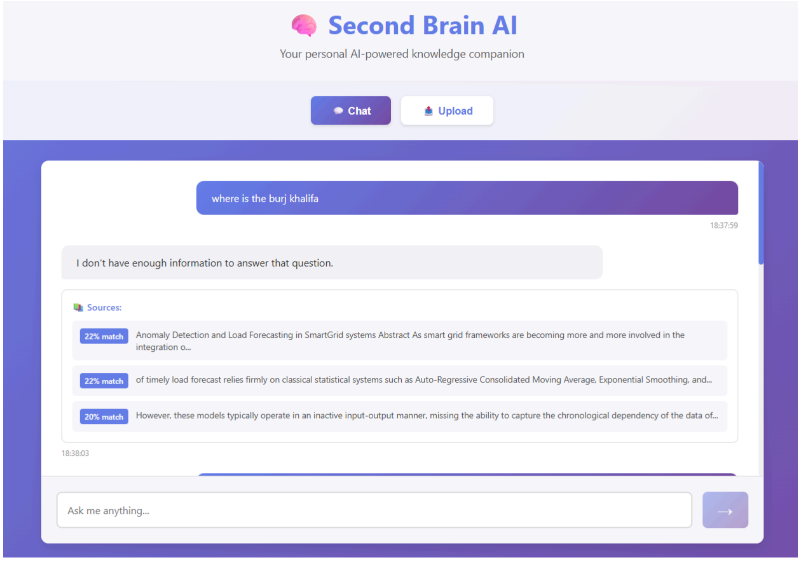
Layout:

- Top navigation bar with app title and user info
- Left sidebar for upload controls and source management
- Central chat interface for conversations
- Bottom input area for typing messages

Key Features:

- Real-time chat interface with message history
- File upload panel supporting audio and documents
- Text input for direct note entry
- Visual indicators for processing status
- Source list showing uploaded content
- Responsive design for different screen sizes

6.2 Screen Images



where is the burj khalifa

18:37:59

I don't have enough information to answer that question.

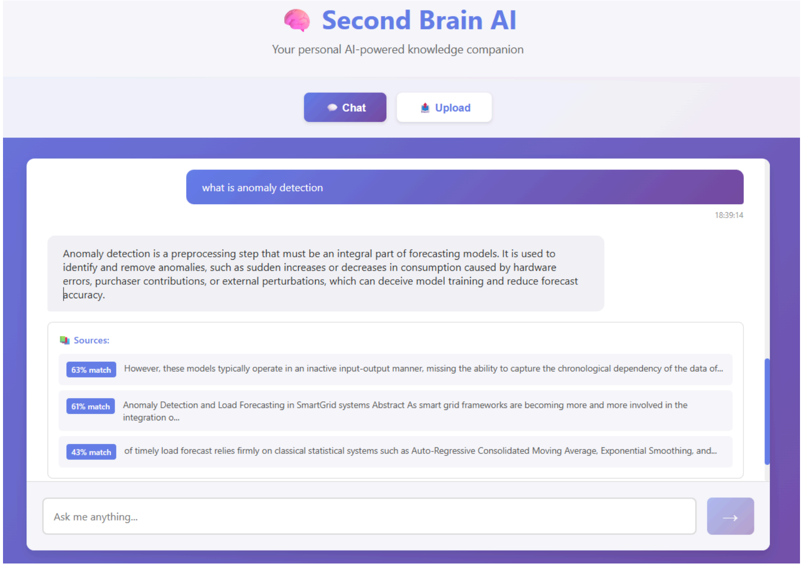
Sources:

22% match Anomaly Detection and Load Forecasting in SmartGrid systems Abstract As smart grid frameworks are becoming more and more involved in the integration o...

22% match of timely load forecast relies firmly on classical statistical systems such as Auto-Regressive Consolidated Moving Average, Exponential Smoothing, and...

20% match However, these models typically operate in an inactive input-output manner, missing the ability to capture the chronological dependency of the data of...

18:38:03



what is anomaly detection

18:39:14

Anomaly detection is a preprocessing step that must be an integral part of forecasting models. It is used to identify and remove anomalies, such as sudden increases or decreases in consumption caused by hardware errors, purchaser contributions, or external perturbations, which can deceive model training and reduce forecast accuracy.

Sources:

63% match However, these models typically operate in an inactive input-output manner, missing the ability to capture the chronological dependency of the data of...

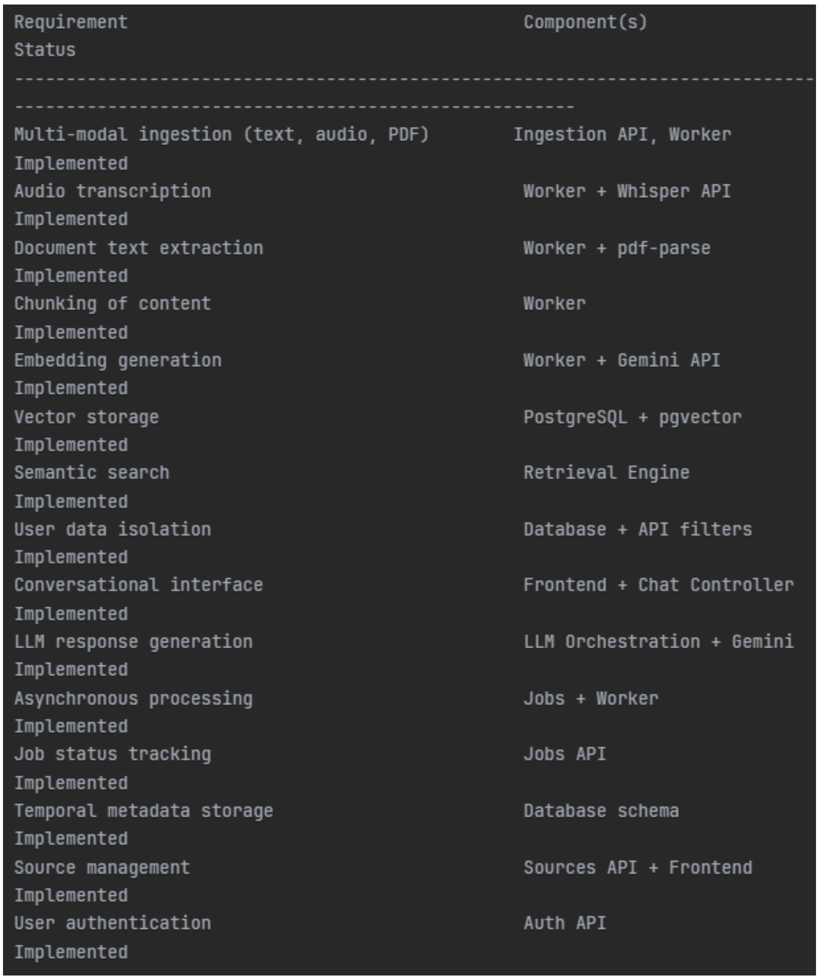
61% match Anomaly Detection and Load Forecasting in SmartGrid systems Abstract As smart grid frameworks are becoming more and more involved in the integration o...

43% match of timely load forecast relies firmly on classical statistical systems such as Auto-Regressive Consolidated Moving Average, Exponential Smoothing, and...

6.3 Screen Objects & Actions

- Upload button → triggers ingestion
- Text box → direct text input
- Chat window → LLM conversation
- Loader/status marker → job progress

7. REQUIREMENTS MATRIX



```
Requirement                                      Component(s)
Status
-----------------------------------------------------------------------------
------------------------------------------------------
Multi-modal ingestion (text, audio, PDF)        Ingestion API, Worker
Implemented
Audio transcription                              Worker + Whisper API
Implemented
Document text extraction                         Worker + pdf-parse
Implemented
Chunking of content                              Worker
Implemented
Embedding generation                             Worker + Gemini API
Implemented
Vector storage                                   PostgreSQL + pgvector
Implemented
Semantic search                                  Retrieval Engine
Implemented
User data isolation                              Database + API filters
Implemented
Conversational interface                         Frontend + Chat Controller
Implemented
LLM response generation                          LLM Orchestration + Gemini
Implemented
Asynchronous processing                          Jobs + Worker
Implemented
Job status tracking                              Jobs API
Implemented
Temporal metadata storage                        Database schema
Implemented
Source management                                Sources API + Frontend
Implemented
User authentication                              Auth API
Implemented
```

8. APPENDICES

A. Technology Stack Summary

Frontend:

- React 18.x
- Axios for API calls
- CSS for styling
- Deployed on Netlify

Backend:

- Node.js 18.x
- Express.js
- Multer for file uploads
- CORS middleware
- Deployed on Render

Worker:

- Node.js 18.x
- Standalone process
- Deployed on Render

Database:

- PostgreSQL 15.x
- pgvector extension
- Hosted on Supabase

External APIs:

- Google Gemini API (text-embedding-004, gemini-1.5-flash)
- OpenAI Whisper API (whisper-1)

Libraries:

- pg (PostgreSQL client)
- pdf-parse (PDF text extraction)
- dotenv (environment configuration)

B. Deployment Architecture

Production Environment:

- Frontend: Netlify (CDN distribution, auto-deploy from Git)
- Backend API: Render (web service, auto-scaling)
- Worker: Render (background worker, separate service)
- Database: Supabase (managed PostgreSQL with pgvector)

Environment Variables:

- DATABASE_URL (PostgreSQL connection string)
- GEMINI_API_KEY (Google AI API key)
- OPENAI_API_KEY (Whisper API key)
- PORT (server port)
- FRONTEND_URL (for CORS configuration)

C. API Response Formats

Success Response:

```json
{
  "success": true,
  "data": { ... }
}
```

Error Response:

```json
{
  "success": false,
  "error": "Error message description"
}
```

D. Database Connection Configuration

Connection String Format:

postgresql://user:password@host:port/database?sslmode=require

Connection Pool Settings:

- Max connections: 20
- Idle timeout: 30 seconds
- Connection timeout: 10 seconds

E. Embedding Model Specifications

Gemini text-embedding-004:

- Dimension: 768
- Max input tokens: 2048

- Use case: Semantic search and retrieval
- Distance metric: Cosine similarity

F. Chunking Strategy Details

Chunk Size: 500-1000 tokens (approximately 375-750 words)

Overlap: None (sequential chunking)

Splitting Method: Sentence-boundary aware

Edge Cases:

- Very short sources (< 500 tokens): Stored as single chunk
- Code blocks: Preserved without splitting mid-block
- Lists: Kept together when possible

---

9. NON-FUNCTIONAL REQUIREMENTS

9.1 Scalability

Stateless API design:

- Backend API servers are stateless
- Session data stored in database
- Enables horizontal scaling by adding more API instances

Worker scalability:

- Workers can run as multiple independent processes
- Database-based queue supports concurrent workers
- Each worker polls for available jobs independently

Database scalability:

- PostgreSQL with connection pooling
- pgvector supports efficient vector indexing
- IVFFlat index provides approximate nearest neighbor search at scale

- Can shard by user_id if needed for horizontal scaling

9.2 Privacy & Security

Data Isolation:

- All queries filter by user_id at database level
- Users cannot access other users' data
- Foreign key constraints enforce referential integrity

Communication Security:

- HTTPS for all API communication
- Environment variables for sensitive credentials
- No API keys exposed to frontend

Data Privacy:

- No training on user data
- User content never leaves the system except for API calls
- Embeddings generated via API, not stored by provider
- Transcriptions processed via API, not retained by provider

Authentication:

- User authentication required for all operations
- Session tokens for maintaining user state
- Password hashing (if implementing password auth)

9.3 Reliability

Job Retry Mechanism:

- Failed jobs marked with status='failed'
- Error messages logged for debugging
- Jobs can be manually reset and retried
- Worker continues processing other jobs on failure

Robust Processing:

- Try-catch blocks around all external API calls
- Graceful handling of malformed inputs
- Transaction rollback on partial failures

Service Separation:

- Worker runs independently from API
- API remains responsive even if worker is slow
- Database acts as reliable queue

Monitoring:

- Job status tracking enables monitoring
- Error messages provide debugging information
- Logs for troubleshooting

9.4 Performance

Asynchronous Ingestion:

- Heavy processing offloaded to background worker
- API responds immediately with job ID
- User can continue using chat while ingestion processes

Vector Search Optimization:

- IVFFlat index reduces search time from O(n) to O(log n)
- Cosine similarity operator optimized by pgvector
- Limit results to top-5 reduces response size

Efficient Retrieval:

- Database indexes on user_id and status
- Single query retrieves all needed chunks
- Minimal data transfer between services

Caching Opportunities (Future):

- Query embeddings could be cached
- Frequently accessed chunks could be cached
- LLM responses could be cached for identical queries

Response Time Targets:

- Chat query response: < 3 seconds
- Job creation: < 500ms
- Job status check: < 200ms
- Source listing: < 500ms

END OF DOCUMENT