Project Documentation

STAN Conversational AI Agent

Project:	STAN Internship Challenge – Conversational AI
Author:	Zeeshan Malik
Date:	September 6, 2025
Stack:	Node.js, Express.js, Google Gemini, SQLite, LanceDB

1. Executive Summary

1.1. Objective

The primary objective of this project was to design and implement a human-like, empathetic, and context-aware conversational AI agent. The chatbot is intended for integration into consumer-facing applications, such as social or UGC platforms. The agent goes beyond a simple Q&A bot by incorporating personalized long-term memory, emotional adaptability, and a consistent personality, thereby creating a more engaging and authentic user experience.

1.2. Key Features

- **Personalized Long-Term Memory:** The agent remembers key details from past conversations to personalize future interactions.
- **Human-Like Interaction:** Employs a carefully crafted persona and dynamic prompt engineering to avoid robotic responses and adapt its tone to the user.
- **Contextual Awareness:** Leverages both short-term conversational flow and long-term semantic memories to provide relevant and coherent replies.
- **Modular & Scalable Design:** Built with a decoupled architecture, allowing the core logic to be easily integrated into any platform. The use of serverless, local databases ensures low operational overhead and easy scalability.

1.3. Technology Stack

The project was built using a modern Node.js stack, chosen for its performance, extensive ecosystem, and suitability for real-time applications.

Component	Technology	Rationale
Backend Framework	**Node.js / Express.js**	Provides a fast, lightweight, and scalable foundation for building the API server.
Language Model (LLM)	**Google Gemini 1.5 Flash**	Offers an excellent balance of performance, cost-effectiveness, and a large context window.
Structured Memory	**SQLite**	A serverless, file-based SQL database perfect for storing user profiles and chat logs without external dependencies.
Semantic Memory	**LanceDB**	An open-source, serverless vector database that runs locally, enabling powerful semantic search for contextual memory recall.

2. System Architecture

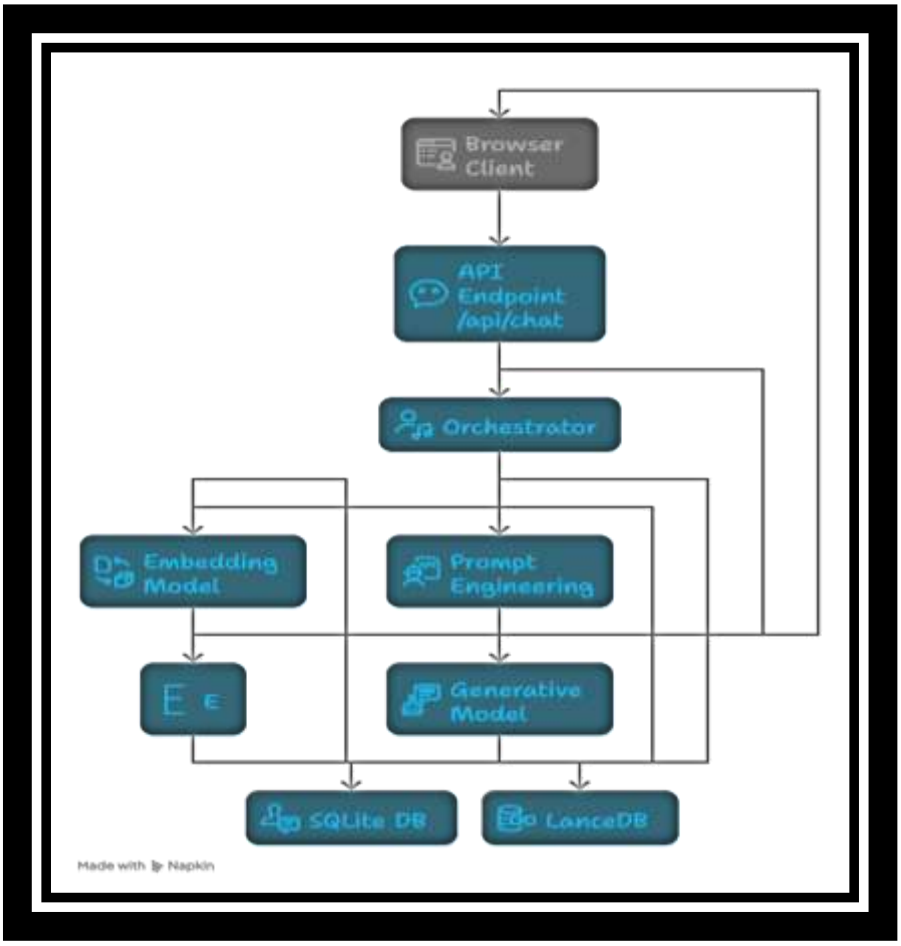
2.1. Core Philosophy: The Dual-Memory System

The chatbot's architecture is centered around a **dual-memory system** to effectively mimic human conversational memory. This approach segregates memory into two distinct types:

1. **Structured Memory (The "Facts"):** Stored in **SQLite**, this memory handles concrete, factual data. It's like a person's explicit memory for things like names, stated preferences ("My favorite color is blue"), and conversation logs. It's fast for direct lookups.
2. **Semantic Memory (The "Vibes"):** Stored in **LanceDB**, this memory handles the conceptual and contextual essence of past conversations. It uses vector embeddings to store the *meaning* behind the words. This enables the chatbot to recall related topics, past feelings, or events, even if the user uses different phrasing (e.g., recalling a conversation about "feeling sad about work" when the user mentions "a tough day at the office").

2.2. Architectural Diagram

The following diagram illustrates the flow of information for a single user interaction.



2.3. Request Lifecycle Explained

The system processes each user message through a seven-step lifecycle:

1. **Request Ingestion:** A user sends a message (containing their userId and message text) from the frontend client. This hits a dedicated POST /api/chat endpoint on the Express.js server.
2. **Immediate History Logging:** The user's message is immediately saved to the chat_history table in SQLite. This ensures no data is lost and provides an up-to-date log for context retrieval.
3. **Context Retrieval:** Before calling the LLM, the system gathers a comprehensive context package:
    - **User Profile (Facts):** The users table in SQLite is queried to fetch the user's profile (e.g., name, interests, conversation summaries).
    - **Short-Term Memory (Recency):** The chat_history table is queried for the last 10-20 conversational turns to provide immediate context.
    - **Long-Term Memory (Relevance):** The user's message is converted into a vector embedding. This vector is used to query LanceDB for the top 3-5 semantically similar memories from all past conversations with that user.
4. **Dynamic Prompt Engineering:** All retrieved context (user profile, short-term history, long-term memories) is dynamically compiled into a single, detailed "master prompt." This prompt instructs the Gemini model on its persona (Stan), provides all known information about the user, and presents the current conversation.
5. **LLM Inference:** The master prompt is sent to the **Gemini 1.5 Flash** API. The model generates a response based on its persona and the rich context provided.
6. **Response to Client:** The generated text from Gemini is immediately sent back to the user's interface, ensuring a low-latency experience.
7. **Asynchronous Memory Update:** After the response has been sent, the system performs "fire-and-forget" memory updates in the background:
    - The AI's response is saved to the chat_history table in SQLite.
    - The recent conversation turn is vectorized and stored in LanceDB as a new long-term memory.
    - *(Optional/Advanced)* A background job can be triggered to summarize the conversation and update the user's profile in SQLite with newly learned facts.

3. Component Deep Dive

3.1. API Server (server.js, src/routes/chat.js)

The Express.js server acts as the central orchestrator. Its responsibilities include:

- Serving the static frontend files (public directory).
- Parsing incoming JSON requests.
- Routing API calls (/api/chat, /api/reset) to the appropriate controller logic.
- Managing the request-response lifecycle and error handling.

3.2. Memory Module (src/chatbot/memory.js)

This module abstracts all database interactions, providing a clean interface for the rest of the application to manage memory without needing to know the underlying database implementation.

- **SQLite Functions:** Provide methods like getOrCreateUser, addMessageToHistory, and getChatHistory. The user profile is stored as a flexible JSON blob, allowing new facts to be added without schema migrations.
- **LanceDB Functions:** Provide methods like addMemory and fetchRelevantMemories. It handles the vectorization of text (via the LLM handler) and querying for semantic similarity.

3.3. LLM Handler (src/chatbot/llm_handler.js)

This is the core intelligence layer of the application.

- **Prompt Engineering:** The generateResponse function is the most critical piece of the application's "soul." It meticulously constructs the prompt to ensure the chatbot maintains its persona, uses memory effectively, and adapts its tone.
- **Embedding Generation:** It contains the getEmbedding function, which interfaces with Google's embedding-001 model to create the vector representations required by LanceDB.

4. Setup and Usage Guide

4.1. Prerequisites

- Node.js (v18 or higher)
- npm (Node Package Manager)
- Git

4.2. Installation

1. Clone the repository: git clone <your-repo-url>
2. Navigate to the project directory: cd stan-chatbot-node
3. Install dependencies: npm install

4.3. Configuration

1. Create a .env file in the root directory.
2. Add your Google Generative AI API key to the file: GOOGLE_API_KEY="your_api_key_here"

4.4. Running the Application

1. Start the server: node server.js
2. Open a web browser and navigate to http://localhost:3000.

5. Conclusion

This project successfully demonstrates the feasibility of building a sophisticated, human-like conversational AI using a modern, cost-effective Node.js stack. The dual-memory architecture provides a robust foundation for both factual recall and contextual understanding, meeting all the core requirements of the STAN Internship Challenge. The system is modular, scalable, and provides a genuinely engaging user experience.