System Documentation: Context Distillation & Indexing

Workflow Name: System: Context Distillation & Indexing for Documentation
Source File: System__Context_Distillation___Indexing_for_Documentation.json
Project Lead: Mike Holland
System Architect: Gemini Pro

1. Overall Goal

This workflow serves as the dedicated "Commit to Memory" function for the Chat 8 UI. Its purpose is to solve the problem of limited context windows in LLMs by creating a persistent, long-term memory store. It is triggered manually by the user, reads the recent, unprocessed conversation history for a given session, uses a powerful AI model to distill that conversation into a set of atomic facts, generates vector embeddings for those facts, and saves them to the rag_store database.

2. Key Components & Architecture

- **Trigger:** A unique n8n **Webhook** (...6c1ce608...) listens for POST requests from the Chat 8 UI's "Commit to Memory" button.
- **Database:** It interacts with three key tables in the Postgres database: conversation_history (to read new messages), memory_commit_log (to track progress with a high-water mark), and rag_store (to save the final indexed memories).
- **External Services:** It makes two distinct calls to the OpenAI API: one to the Chat Completions endpoint for fact distillation and another to the Embeddings endpoint for vectorization.
- **Logic:** The workflow is a sequential data processing pipeline that includes a crucial "high-water mark" system to prevent duplicate processing of messages.

3. Step-by-Step Data Flow

The workflow executes in a clear, linear sequence:

1. **Trigger: Receive Session ID:** The workflow is triggered by a POST request from the UI. The payload contains the chat_session_id and rag_session_id for the current conversation.
2. **Get Project ID:** A Postgres node takes the chat_session_id and queries the project_contexts table to find the associated permanent project_id.

3. **DB: Get High-Water Mark:** A Postgres node queries the memory_commit_log table to find the ID of the last message that was processed for this chat_session_id. It uses a COALESCE function to safely default to 0 if no record exists (i.e., this is the first commit for the session).
4. **Set: Watermark Value:** A Set node takes the high_water_mark from the previous step and prepares it for the next query.
5. **DB: Get Conversation History:** A Postgres node queries the conversation_history table, fetching all messages for the current chat_session_id whose id is greater than the highWaterMark.
6. **Calculate New High-Water Mark:** A Code node iterates through the messages retrieved in the previous step and finds the highest message id. This value will be saved at the end of the workflow.
7. **IF: New Messages Found?:** An If node checks if any new messages were returned. If the count is zero, the workflow stops cleanly. If new messages exist, it proceeds.
8. **Format: Assemble Transcript:** A Code node takes the new messages and formats them into a clean, human-readable transcript string.
9. **Build: OpenAI Request Body:** A Code node constructs the full JSON payload for the fact-distillation call to the OpenAI API, including a detailed system prompt and the conversation transcript.
10. **Execute: OpenAI API Call:** An HTTPRequest node sends the payload to the OpenAI Chat Completions API (gpt-4o).
11. **Set: Extracted Summary:** A Set node parses the AI's response and extracts the JSON string containing the distilled facts.
12. **Chunk: Distilled Summary:** A Code node parses the JSON string of facts and splits them, outputting each individual fact as a separate item to be processed in the following steps.
13. **API: Create Embeddings (HTTP) node:** (Runs for each fact) An HTTPRequest node takes each fact and calls the OpenAI Embeddings API (text-embedding-3-small) to generate its vector embedding.
14. **Merge Embedding with Original Data:** A Code node (not a Merge node) runs to assemble the final, complete payload for the database, combining the project_id, session_id, the fact text, the role (summary), and the new embedding vector.

15. **DB: Insert Embedding1:** (Runs for each fact) A Postgres node executes the INSERT query to save the complete record for each fact into the rag_store table.
16. **DB: Update High-Water Mark:** The final database operation. A Postgres node performs an UPSERT on the memory_commit_log table, saving the new_high_water_mark calculated in Step 6. This ensures the next run will only process messages created after this point.
17. **Count Facts & Respond to Webhook6:** A final Code node counts the number of facts committed, and the Respond to Webhook node sends a detailed success message (including a "toast" for the UI) back to the user, completing the cycle.