System Documentation: Main Chat Workflow

Workflow Name: Main Chat Workflow for Documentation
Version: As of file provided
Project Lead: Mike Holland
System Architect: Gemini Pro

1. Overall Goal

This n8n workflow serves as the central orchestration layer for the Chat 8 user interface. Its primary purpose is to receive user input, intelligently enrich it with multiple forms of context—long-term memory from a RAG store and short-term conversational history—and generate a context-aware response from a large language model. It also includes a new feature for automatically committing "remember" intents to memory.

2. Key Components & Architecture

The system is comprised of several key components that work in concert:

- **Frontend:** The Chat 8 UI (Chat_8_V7.html) captures user input and displays the final response.
- **Backend Orchestration:** This n8n workflow manages the entire data flow and logic.
- **Long-Term Memory:** A Postgres database containing the rag_store for vectorized knowledge.
- **Short-Term Memory:** The same Postgres database, using the conversation_history table for recent conversational turns.

The workflow's core architecture is a multi-branch parallel processing system. After initial input, it splits into distinct paths to gather different types of context, which are then merged before the final AI call.

3. Step-by-Step Data Flow

The workflow executes in several distinct phases:

Phase 1: Ingestion and Preparation

1. **Webhook1:** The workflow is triggered by a POST request from the UI. It is configured to handle pre-flight OPTIONS requests for CORS compatibility.
2. **Get Session ID & Input:** A Code node parses the incoming request body. It validates that chat_session_id and chatInput are present and transforms all key IDs to a

consistent camelCase format (session_Id, rag_session_Id, project_Id) for use within the workflow.

3. **Save User Message to History:** A Postgres node immediately saves the user's message to the conversation_history table, ensuring a complete and persistent log of the interaction.
4. **Preserve Current Inputs:** A critical Set node creates a stable, preserved copy of the key inputs (session_Id, chatInput, rag_session_Id, etc.). This node acts as a central hub, providing a reliable data source for all subsequent parallel branches.

Phase 2: Auto-Commit Branch (Side Process)

Running in parallel to the main chat logic.

1. **Config: Auto-Commit Enabled:** A Set node acts as a feature flag, enabling the auto-commit functionality.
2. **If Remember Intent:** An If node checks if the user's chatInput starts with the word "remember".
3. **Auto Commit via Webhook:** If the intent is to remember, an HTTPRequest node triggers the separate "Commit to Memory" workflow, passing the necessary IDs to save the new fact in the background.

Phase 3: Main Logic Branching

1. **If Rag is Active:** This is the primary traffic controller. It checks if a rag_session_Id was provided.
    - **If True:** The workflow proceeds down the full RAG path to retrieve long-term memory.
    - **If False:** The workflow bypasses the RAG steps and proceeds directly to retrieve only the short-term conversational history.

Phase 4: The RAG Path (Dual-Retrieval)

This path executes if RAG is active.

1. **Create Query Embedding:** An HTTPRequest node takes the user's chatInput and calls the OpenAI API to convert it into a vector embedding.
2. **Format Data for Vector Search:** A Code node prepares the data for the database search, formatting the embedding vector into the required string format for pgvector.

3. **Parallel Retrieval:** The workflow splits again to perform two simultaneous database lookups:
    - **Retrieve Committed Memory:** A Postgres node searches the rag_store for memories from the live conversation.
    - **Retrieve RAG Chunks:** A Postgres node searches the rag_store for memories from the initial bootstrapped knowledge base.
4. **Merge1:** A Merge node combines the results from both retrieval steps into a single list.
5. **RAG Context consolidator:** A Code node takes the merged list, removes any duplicate memories, and formats the unique results into a single, clean block of text.
6. **Format History for AI:** A Code node takes the consolidated text and wraps it in a standard { role: 'system', content: '...' } object, ready for the final prompt.

Phase 5: Final Prompt Assembly & AI Call

1. **Get Recent history:** (Runs in parallel to the RAG path) A Postgres node queries the conversation_history table for the last 6 turns of the conversation.
2. **Format Recent History:** A Code node formats these turns into the standard OpenAI message format.
3. **Format Current Input:** A Code node formats the user's current message into the standard format.
4. **Merge:** This is the final assembly point. A Merge node combines the three streams of context:
    - **Input 1:** The RAG context (from Format History for AI).
    - **Input 2:** The recent conversational history (from Format Recent History).
    - **Input 3:** The user's current message (from Format Current Input).
5. **Build OpenAI Payload1:** A sophisticated Code node takes the fully merged data. It intelligently selects the correct system prompt (persona), assembles the final message array in the correct order (system prompt, RAG context, history, user message), and builds the complete JSON payload for the AI.
6. **HTTP Request1:** Sends the final payload to the OpenAI Chat Completions API.

Phase 6: Response and Finalization

1. **Code1:** A Code node parses the response from OpenAI, extracts the AI's reply, and performs cleaning and truncation. It also checks for any status messages from the background Auto-Commit process and appends them to the reply.
2. **Postgres2 (Save AI Reply):** Saves the AI's generated response back to the conversation_history table.
3. **Respond to Webhook2:** Sends the final, clean reply back to the Chat 8 UI, completing the cycle.