Project Report: Vehicle Inspection Chatbot System

This document provides a detailed walkthrough of the Vehicle Inspection Chatbot project. The system is designed to process natural language queries about vehicle inspection data, leveraging both SQL-based and semantic search methods. It integrates with PostgreSQL as a backend database, FAISS for semantic retrieval, and an LLM (Ollama + Qwen model) for query understanding and response generation. Below, we will go through each core component of the project in sequence, explaining their roles and flow in simple terms.

app.py

This script provides a simple command-line interface for interacting with the chatbot. It initializes the chatbot with database configuration and allows the user to type queries. The system runs in a loop until the user types 'exit' or 'quit'. For each query, it delegates processing to the chatbot and prints the formatted response. This is mainly used for local testing.

main.py

The FastAPI-based web server exposing the chatbot as an HTTP API. It defines endpoints: - **/chat**: Accepts a query and returns the chatbot response with a session ID and timestamp. - **/session/{session_id}/clear**: Clears session history for a given session. It uses UUIDs to manage sessions and ensures CORS is enabled for frontend integration. This module makes the system accessible as a web service.

chatbot.py

This is the central logic of the system. The `VehicleChatbot` class orchestrates the query flow: 1. Checks for cached responses to speed up repeat queries. 2. Validates queries to block destructive operations like DELETE or UPDATE. 3. Routes queries either to SQL or semantic search using the LLM interface. 4. Executes SQL against the database via the DatabaseManager. 5. Enriches results with additional insights like top damages or inspector stats. 6. Formats results into a natural language response using the LLM. It also maintains conversation history with a memory system and gracefully handles errors.

database.py

Handles all database connectivity and querying. The `DatabaseManager` establishes a connection to PostgreSQL and provides methods to execute queries. It also defines schema metadata describing the inspection table and its fields (VIN, inspector name, damage comments, etc.). The query results are converted into serializable dictionaries for further processing.

semantic_search.py

Implements semantic search for vague queries. It uses FAISS and sentence-transformers (`nomic-embed-text-v1`) to embed and index damage descriptions. If a query is semantically routed, this component retrieves the most relevant inspection records. The index is built from database records and stored locally for fast retrieval.

llm_interface.py

Provides the connection to Ollama’s local LLM server (Qwen2.5 model). Its responsibilities include: - Preprocessing queries and enforcing SQL formatting rules. - Routing queries to SQL or semantic search. - Generating valid SQL queries based on schema and examples. - Summarizing query results. - Formatting chatbot responses in a professional, structured manner. This ensures that all queries and responses follow strict, predictable rules.

formatter.py

Contains an async function to further format responses using an LLM endpoint. It takes the original query, SQL result, additional context, and sources, then generates a clear and client-friendly output. This is useful for refining the final response.

schemas.py

Defines Pydantic models for request and response validation. `Message` and `Session` schemas capture conversation history and ensure consistent session tracking. These are especially important for maintaining structured state in the FastAPI service.

How to Run the Project 1. Ensure PostgreSQL is running and the `vehicle_inspection_db` is set up with the `inspections` table. 2. Install required Python packages: `pip install -r requirements.txt`. 3. Start the Ollama server locally with: `ollama serve`. 4. To run via CLI, execute: `python app.py` and interact with the chatbot in terminal. 5. To run the web service, start FastAPI with: `uvicorn app.main:app --reload --host 0.0.0.0 --port 8000`. 6. Once running, you can query the chatbot at `http://localhost:8000/chat` using POST requests.