Task 2 – Report: Star Rating Prediction System Using LLMs

Candidate: Dhairya Dharmesh Muni
Role: AI Engineering Intern – Take-Home Assignment
Model Used: Gemini 2.5 Flash-Lite

Project Report: Two-Dashboard AI Feedback System

1. Executive Summary

FeedbackAI is a full-stack web application designed to close the loop between customer feedback and administrative action. It features two distinct interfaces: a User Dashboard for submitting reviews and receiving instant, empathetic AI responses, and an Admin Dashboard for visualizing sentiment trends and actionable business insights. The system is powered by Google Gemini 2.5 Flash.

2. Architectural Approach

The system follows a MERN Stack (MongoDB, Express, React, Node.js) architecture with a decoupled frontend and backend to ensure scalability and security.

- Frontend: Built with React 19 & Vite for high performance. Uses Tailwind CSS for responsive design and dark mode support, and Recharts for data visualization.
- Backend: A Node.js/Express server acts as the orchestrator. It manages database connections and serves as a secure proxy for the Gemini API.
- Database: MongoDB (Atlas) stores feedback submissions, allowing for flexible schema evolution (e.g., adding new AI analysis fields without breaking the app).
- Deployment: A "Headless" approach where the same codebase is deployed to two different URLs (via Vercel environment variables) to serve distinct user roles, while connecting to a centralized Render backend.

3. Key Design Decisions

- Server-Side AI Processing:
    - *Initial Design:* Client-side AI calls.
    - *Pivot:* Moved logic to server.js.
    - *Reasoning:* To securely hide the API_KEY, resolve CORS issues, and implement robust error handling that the client browser cannot manage effectively.
- Resiliency & Retry Logic:
    - API Quota limits (HTTP 429) are common with GenAI. I implemented an exponential backoff strategy in the backend. If the AI is busy, the server waits (1s, 2s, 4s) and retries before failing, significantly reducing crash rates.
- Graceful Degradation:

    - If the AI service fails completely after retries, the system falls back to pre-written static responses. This ensures the user *always* gets a confirmation, and data is *always* saved, preserving the user experience.

- JSON Schema Enforcement:
    - Instead of parsing raw text, we utilized Gemini's responseSchema and responseMimeType: "application/json". This guarantees that the AI output (Sentiment, Summary, Actions) is always machine-readable and strictly typed.

4. Prompt Engineering & Iterations

The prompt underwent three stages of iteration to maximize utility:

1. Iteration 1 (Basic): "Write a response to this review."
    - *Result:* Good text, but no data for the admin dashboard.
2. Iteration 2 (Multi-task): "Write a response, give me a summary, and tell me the sentiment."
    - *Result:* Hard to parse programmatically (Regex required).
3. Iteration 3 (Structured & Role-Based - *Final*):
    - *Role:* "You are an AI feedback assistant..."
    - *Input:* Rating + Review Text.
    - *Output constraints:* Strictly enforced JSON object with specific keys (userResponse, summary, recommendedActions, sentiment).
    - *Model:* Switched from gemini-2.0-flash-lite to gemini-2.5-flash for higher reasoning capabilities regarding sentiment analysis.

5. System Behavior

1. Input: A user submits a rating (e.g., 2 stars) and text ("Service was slow").
2. Processing:
    - The frontend shows a skeleton loader ("Analyzing...").
    - The backend receives the payload and prompts Gemini.
    - Gemini generates a polite apology for the user and extracts "Negative" sentiment and specific actions (e.g., "Hire more staff") for the admin.
3. Storage: The combined user data + AI analysis is saved to MongoDB.
4. Output (User): The user immediately sees the empathetic response.
5. Output (Admin): The Admin Dashboard polls the database, updating the Live Sentiment Charts and adding the new entry to the actionable list without a page refresh.

6. Evaluation

The final system successfully meets the requirement of a "Two-URL" deployment using free-tier services. It balances User Experience (fast, responsive, dark mode) with Technical Robustness (error handling, type safety, secure API usage). The separation of the "Admin" and "User" views via build-time environment variables (VITE_APP_MODE) ensures a clean separation of concerns within a single codebase.