

Task 2 Report: AI-Powered Feedback Management System

Objective

The objective of Task 2 was to design and deploy an AI-powered feedback management system that:

- Collects user feedback (rating + review)
 - Generates an AI-driven response to the user
 - Produces an internal summary and recommended action for administrators
 - Provides separate User and Admin dashboards
 - Is deployable as a web application
-

System Overview

The system was implemented as a Streamlit web application with two main components:

1. User Dashboard

- Allows users to submit:
 - A rating (1–5)
 - A textual review
- Displays an AI-generated, empathetic response immediately after submission

2. Admin Dashboard

- Displays all submitted feedback
 - Shows:
 - Rating
 - Review
 - AI-generated user response
 - Internal summary
 - Recommended action
 - Calculates and displays the average rating
-

Technology Stack

Component	Technology
Frontend & UI	Streamlit
Backend Logic	Python

AI Model	Google Gemini (gemini-flash-lite-latest)
Data Storage	CSV file
Secrets Management	Environment variables
Deployment	Streamlit Community Cloud

AI Integration Design

Single-Call LLM Strategy

Initially, separate prompts were considered for:

- User response
- Admin summary
- Recommended action

However, this approach required multiple API calls per submission, which is inefficient and incompatible with free-tier rate limits.

Final Optimized Approach

All AI outputs are generated using a single Gemini API call per user submission, returning structured JSON:

```
{  
  "user_response": "...",  
  "summary": "...",  
  "recommended_action": "..."  
}
```

This design:

- Reduces API usage
 - Improves performance
 - Aligns with production best practices
-

Prompt Design

The combined prompt instructs the model to:

- Analyze customer sentiment
- Generate a polite user-facing response
- Produce an internal summary
- Suggest a clear business action
- Output only valid JSON to ensure safe parsing

This structured approach minimizes parsing errors and ensures consistent outputs.

Error Handling & Reliability

Graceful Degradation

If the Gemini API fails due to:

- Quota exhaustion
- Network issues
- Invalid responses

The system falls back to a safe default response, ensuring:

- The app never crashes
- Feedback is still stored
- Admins can review entries manually

Fallback Example

```
{  
  "user_response": "Thank you for your feedback.",  
  "summary": "Error generating summary.",  
  "recommended_action": "Review manually."  
}
```

API Quota Management

Challenge

The Gemini free tier has strict daily request limits.

Solution

- Consolidated AI generation into one API call per submission
- Implemented in-memory caching to prevent repeated calls on UI reruns
- Limited testing submissions during deployment

This ensured stable operation without exceeding quota limits.

Data Storage Strategy

- Feedback is stored in a CSV file with the following fields:
 - Timestamp
 - Rating

- Review
 - AI-generated response
 - Summary
 - Recommended action
- Atomic file writes were used to prevent file-locking issues, especially on Windows environments.
 - CSV storage was chosen for simplicity and transparency, suitable for a prototype-scale system.
-

Deployment

The application was deployed using Streamlit Community Cloud:

- API keys are securely stored using Streamlit Secrets
 - No sensitive credentials are committed to version control
 - The deployed app provides a public URL for easy access and demonstration
-

Limitations

- Gemini free-tier quota limits restrict the number of daily submissions
 - CSV storage is suitable for small-scale usage but not ideal for large production workloads
 - Admin authentication was not implemented (out of scope)
-

Future Improvements

- Replace CSV storage with a database (PostgreSQL / Firestore)
 - Add admin authentication and role-based access
 - Introduce sentiment analytics and visual dashboards
 - Implement async request handling for higher scalability
 - Add feedback categorization and tagging
-

Conclusion

Task 2 successfully demonstrates:

- Practical LLM integration
- Efficient prompt engineering
- API quota-aware system design
- Robust error handling

- End-to-end deployment of an AI-powered application

The system reflects real-world engineering considerations and balances functionality with reliability under constrained resources.