Trip Recommendation Project: System Architecture and RAG Pipeline Write-up

Cloud and Data Architecture Choices

When building this project, the goal was to make travel recommendations feel more personal and helpful, like talking to a real travel planner who knows your tastes and budget. To do this, I used a combination of **traditional databases (BigQuery)**, **semantic search techniques (RAG)**, and cloud infrastructure to keep things fast and accessible.

Google Cloud was chosen mainly because I had free credits available, and it made it easy to set up the pieces I needed. **BigQuery became my main data warehouse** for structured data, it holds details about destinations, weather, costs, and tags. It's great for SQL-based filtering, such as "show me affordable places with good weather for a luxury traveler in July." BigQuery is reliable, fast, and scales well even with large datasets.

For hosting, I deployed the app on Google App Engine. This lets me serve the Streamlit app without worrying about managing servers or infrastructure. It's simple, integrates well with the rest of Google Cloud, and supports authentication and logs.

RAG Pipeline: How It Works

For deeper, AI-based recommendations, I added a RAG pipeline. Instead of just filtering with SQL, this part uses natural language to understand what someone wants and recommends matches based on meaning, not just keywords.

Here's how it works:

1. I took datasets like destinations, user trip histories, and weather summaries, and converted each row into a **vector** (a math-based representation of its meaning).
2. I used the model all-MiniLM-L6-v2 from **Hugging Face Transformers** for this. It's a small but powerful model that can convert sentences into meaningful embeddings (vectors). I chose this because:
    - It's accurate enough for semantic search
    - It's lightweight and works well on a normal laptop
    - It doesn't require paid API access or GPU hosting
3. Once I had these vectors, I stored them in a **FAISS index** (from Facebook), which lets me quickly find the most similar rows when a user gives a new query like:

    "Find affordable cultural trips with great weather in January"

4. This query is also converted into a vector, and FAISS returns the closest matches. Those matches are then shown as recommendations with insights like cost, safety, and why it was picked.

I built different pipelines like:

- Matching destinations based on tags and travel month
- Finding similar travelers from historical data
- Suggesting destinations with good weather and matching preferences

Each of these pipelines uses the same RAG technique with a slightly different dataset and filtering logic.

The RAG setup has two major parts:

1. **Embedding the data** - I used the all-MiniLM-L6-v2 model from Hugging Face Transformers. This is a lightweight, fast sentence-transformer that works well even with limited resources. I chose it because it balances accuracy and speed, and I could run it locally without needing a powerful GPU.

2. **Storing and searching** - After converting destination descriptions, trip histories, and weather data into embeddings (vectors), I stored them in FAISS - a vector search library built by Facebook. It allows me to quickly find the closest matches when a user gives a new query.

So whenever a user describes their travel style, budget, or month, I convert that input into an embedding and use FAISS to find similar destinations or user trip histories. These matches, plus a bit of logic and filtering (e.g., within budget), become the final recommendations shown in the app.

Why Hugging Face Transformers?

Hugging Face models are easy to use, well-documented, and work well with real-world text. I specifically used sentence-transformers from Hugging Face because:

- They work out of the box for text similarity tasks.
- I didn't need huge computing power to use them.
  They are more accurate than simple keyword matching (e.g., "culture" and "historic" are understood as similar concepts).

It made the system feel more human - like the model understood travel preferences instead of just doing keyword matching.

Additional Data Used and Why

To improve the quality of recommendations, I added three types of data:

1. **Weather data per destination by month** - This helped me show "peak season picks" and suggest places with good weather for a user's travel month.
2. **Trip histories of past members** - This gave me a way to match a new traveler with similar travelers and show where those people went.
3. **Activity tags for each destination** - These help personalize results. A person interested in "adventure" gets places tagged with "hiking," "safari," or "diving."

This combination of structured (tables) and unstructured (text + embeddings) data made the system both accurate and flexible.

Trade-Offs and Assumptions

One major assumption was that the tags and weather ratings I assigned were trustworthy and complete. Since this is a synthetic or cleaned dataset, I had to assume that tags like “luxury” or “budget-friendly” were meaningful and consistent across rows.

Another trade-off was not using larger language models (like GPT-4 or Gemini Pro) for embedding due to cost and access limits. Instead, I went with a smaller Hugging Face model that I could run locally. It’s not perfect, but it gives surprisingly good results with fast performance.

Also, for the Gemini chat assistant part, I started with Google’s models but faced persistent access issues. So I added support for OpenAI’s GPT models (like gpt-3.5-turbo) using the OpenAI Python library, with an option for users to upload CSV files and ask questions about their own data.

Future To-Dos

1. **Properly restructure the Git repository**
   Clean up naming conventions, remove unused scripts, and organize folders better for long-term scalability.
2. **Enhance the AI and embedding models**
   Explore better embedding models like bge-m3, use metadata-aware vector search, or apply re-ranking with LLMs.
3. **Deploy to Streamlit Cloud**
   - Push project to a public GitHub repo
   - Sign in to streamlit.io/cloud
   - Deploy app directly with one click
   - Store secrets (API keys and credentials) using their built-in Secrets Manager