# Smart Travel Guide: A Comparative RAG Performance Analysis
# (GPT-4.1-Mini vs. Gemini-2.5-Flash)

## Project Overview
This project implements a Retrieval-Augmented Generation (RAG) based chatbot designed to provide comprehensive information about European cities (e.g., Rome, Prague). The system covers diverse travel categories, including local food culture, urban transportation, top photography spots, historical landmarks, and major tourist attractions. The primary objective is to evaluate and compare the performance of two different Large Language Models (LLMs)—OpenAI GPT-4.1-Mini and Google Gemini-2.5-Flash—using a multi-faceted local dataset.

## Technical Architecture
The system follows a standard RAG pipeline:

Frontend: Streamlit

Orchestration: LangChain

Vector Database: ChromaDB

Retrieval Strategy:
- Similarity search
- k=6 for interactive chatbot usage
- k=1 for evaluation to enforce strict intent matching

Models:
* Gpt-4.1-Mini with OpenAI Embeddings

* Gemini-2.5-Flash with Google Generative AI Embeddings

## Repository Structure
The project is organized according to the specified directory structure:
```text
chatbot_project/
├── data/
│   └── avrupa_gezi_rehberi_1000.xlsx      # Custom dataset
├── models/
│   ├── openai_model.py                   # OpenAI RAG
implementation (builds vector DB)
│   └── gemini_model.py                   # Gemini RAG
implementation (builds vector DB)
├── evaluation/
│   ├── build_train_vector_db.py          # Vector DB for
evaluation
```

```
│   └── evaluate_models.py                # Model comparison &
metrics
├── app/
│   └── streamlit_app.py                  # Streamlit user
interface
├── video/
│   └── demo_video.mov                     # Demo video
├── README.md                              # Project
documentation
├── Smart Travel Guide.pdf
└── requirements.txt                       # Dependency list
```

## Performance Metrics & Benchmarking
The models were evaluated using intent-level classification metrics rather than token-level text similarity. Each test question was matched against the vector database built from the training set, and the predicted intent was compared with the ground-truth intent label.

Evaluation was performed using the following metrics:

- Precision
- Recall
- F1 Score

A strict evaluation setup was used where:
- The vector database was built **only on the training set**
- The test set was completely unseen during indexing
- Retrieval was performed with k=1 to avoid intent leakage

### Quantitative Results

Model	Precision	Recall	F1 Score
OpenAI (gpt-4.1-mini)	0.96	0.96	0.96
Google (gemini-2.5-flash)	0.94	0.95	0.94

### Response Time Comparison

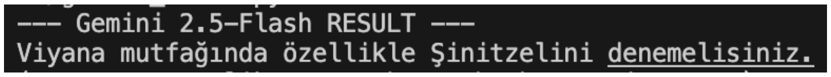
Model	Average Response Time (s)
OpenAI (gpt-4.1-mini)	**1.48**
Google (gemini-2.5-flash)	**1.74**

## How to Run

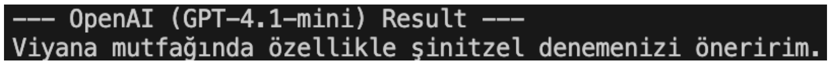
1. Clone the repository
2. Install dependencies:
`pip install -r requirements.txt`
3. Setup Environment Variables:
Create a credentials.env file and add your API keys:
`OPENAI_API_KEY=your_openai_key`
`GOOGLE_API_KEY=your_google_key`
4. Build vector databases and run model demos:
`python models/openai_model.py`
`python models/gemini_model.py`
5. Build training vector database and run model evaluation
`python evaluation/build_train_vector_db.py`
`python evaluation/evaluate_models.py`
6. Launch the App:
`streamlit run app/streamlit_app.py`

## Demo Video
A detailed demonstration of the chatbot, including the "See Raw Data" feature which shows the retrieved context from ChromaDB, can be found [video/demo_video.mp4].

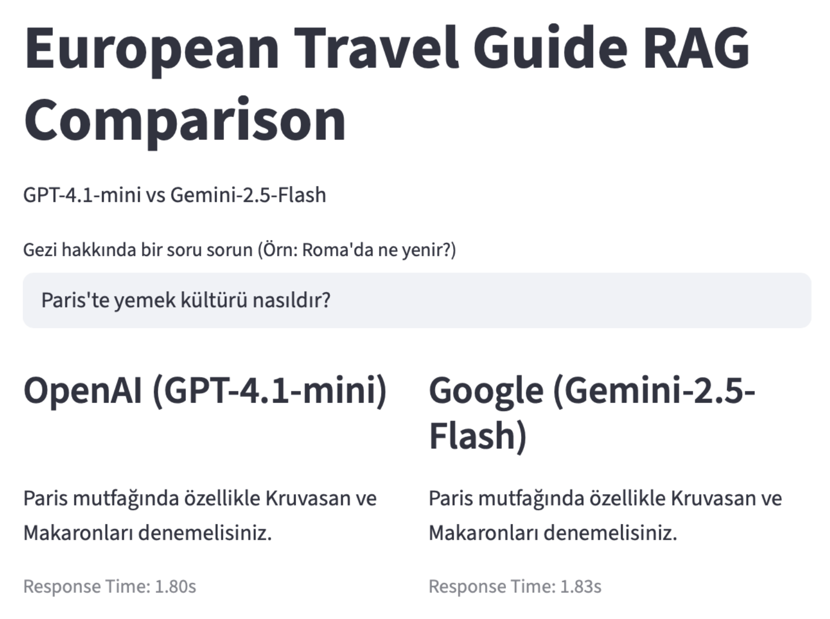
## Example Outputs



```
--- Gemini 2.5-Flash RESULT ---
Viyana mutfağında özellikle Şinitzelini denemelisiniz.
```



```
--- OpenAI (GPT-4.1-mini) Result ---
Viyana mutfağında özellikle şinitzel denemenizi öneririm.
```



European Travel Guide RAG Comparison

GPT-4.1-mini vs Gemini-2.5-Flash

Gezi hakkında bir soru sorun (Örn: Roma'da ne yenir?)

Paris'te yemek kültürü nasıldır?

OpenAI (GPT-4.1-mini)

Paris mutfağında özellikle Kruvasan ve Makaronları denemelisiniz.

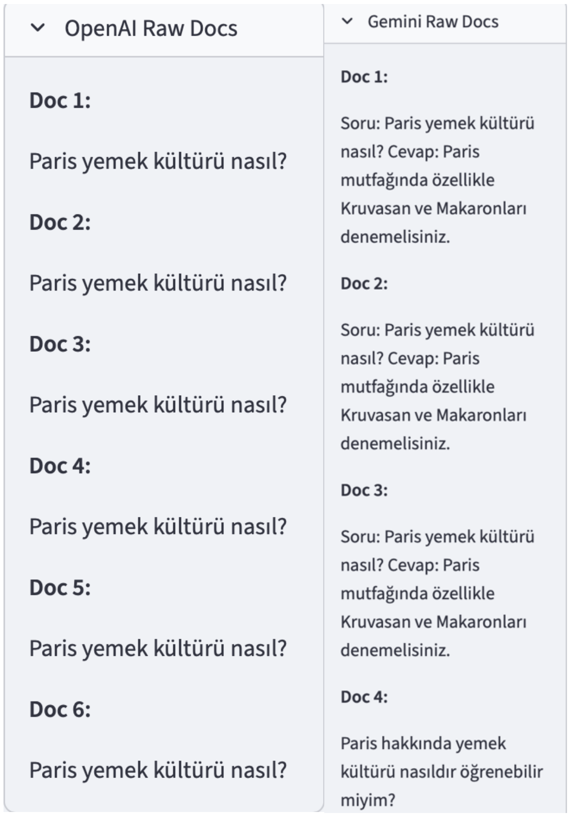
Response Time: 1.80s

Google (Gemini-2.5-Flash)

Paris mutfağında özellikle Kruvasan ve Makaronları denemelisiniz.

Response Time: 1.83s

## Raw Data Columns



Doc 1:

Paris yemek kültürü nasıl?

Doc 2:

Paris yemek kültürü nasıl?

Doc 3:

Paris yemek kültürü nasıl?

Doc 4:

Paris yemek kültürü nasıl?

Doc 5:

Paris yemek kültürü nasıl?

Doc 6:

Paris yemek kültürü nasıl?

Doc 1:

Soru: Paris yemek kültürü nasıl? Cevap: Paris mutfağında özellikle Kruvasan ve Makaronları denemelisiniz.

Doc 2:

Soru: Paris yemek kültürü nasıl? Cevap: Paris mutfağında özellikle Kruvasan ve Makaronları denemelisiniz.

Doc 3:

Soru: Paris yemek kültürü nasıl? Cevap: Paris mutfağında özellikle Kruvasan ve Makaronları denemelisiniz.

Doc 4:

Paris hakkında yemek kültürü nasıldır öğrenebilir miyim?

## Implementation Methodology
This project implements a Retrieval-Augmented Generation (RAG) architecture to provide accurate travel information based on a curated European Travel Guide dataset. The development process is divided into four main stages:

1. Data Ingestion & Preprocessing

Source Dataset: A structured Excel file containing 1,000 question-answer pairs regarding European destinations, cuisines, and logistics.

Chunking Strategy: Utilizing LangChain's DataFrameLoader, each row is transformed into a discrete Document object. The "Question" column serves as the page_content (primary searchable index), while the "Answer" and "Intent" columns are preserved as metadata.

Data Splitting: To ensure unbiased evaluation, the dataset was split into 80% Training and 20% Testing sets using a stratified split to maintain consistent intent distribution across both sets.

2. Vectorization & Vector Store (Embedding)

Two parallel embedding pipelines were established for comparative analysis:

OpenAI Pipeline: Utilizing the text-embedding-3-small model for generating high-dimensional dense vectors stored in a ChromaDB instance.

Google Pipeline: Utilizing the models/text-embedding-004 (Gemini) model for vectorization and storage in a separate ChromaDB instance.

Search Mechanism: The system utilizes a Semantic Search approach by configuring the retriever with search_type="similarity". Instead of relying on traditional keyword matching, this mechanism compares the vector embedding of the user's query against the stored document embeddings. By retrieving the top-k most relevant matches based on vector proximity, the system can understand the underlying context and provide accurate information even when the user's phrasing differs from the source text.

3. RAG Chain Architecture

The core logic is built using LCEL (LangChain Expression Language), creating a modular pipeline:

Retriever: Fetches the top 6 most relevant documents (k=6) from the vector store based on the user's query.

Prompt Template: Orchestrates a specialized system prompt that instructs the LLM to remain faithful to the retrieved context and avoid hallucinations.

LLM Processing: gpt-4.1-mini and gemini-2.5-flash act as the reasoning engines, synthesizing the retrieved information into a natural language response.

Output Parser: Standardizes the raw model output into a clean string format for the UI.

4. Evaluation Framework

Performance was quantified using a rigorous classification-based approach:

Intent Classification Test: The test set was processed through the retriever with k=1. The predicted intent (from the retrieved document's metadata) was compared against the ground truth.

Quantitative Metrics: Performance was measured using Precision, Recall, and F1-Score.

Efficiency Metrics: Average Response Time (Latency) was tracked for both models to evaluate real-world usability.

## Interactive UI & Real-Time Comparison (Streamlit)

The project features a web-based dashboard built with Streamlit, designed for side-by-side model comparison and real-time interaction.

Dual-Chain Execution: The application runs both OpenAI and Google RAG chains simultaneously for every user query, allowing for an immediate comparative analysis of the outputs.

Performance Tracking: Each response is accompanied by a "Response Time" metric, providing data on the latency of each model ecosystem.

State Management & Optimization: Using the @st.cache_resource decorator, the application ensures that heavy resources like Vector DBs and LLM connections are loaded only once, significantly improving performance for subsequent queries.

Debug Mode (Explainability): A dedicated sidebar displays the Retrieved Documents (raw context) for both models. This feature provides transparency into which specific data points were selected by the retriever, aligning the project with Explainable AI (XAI) principles.