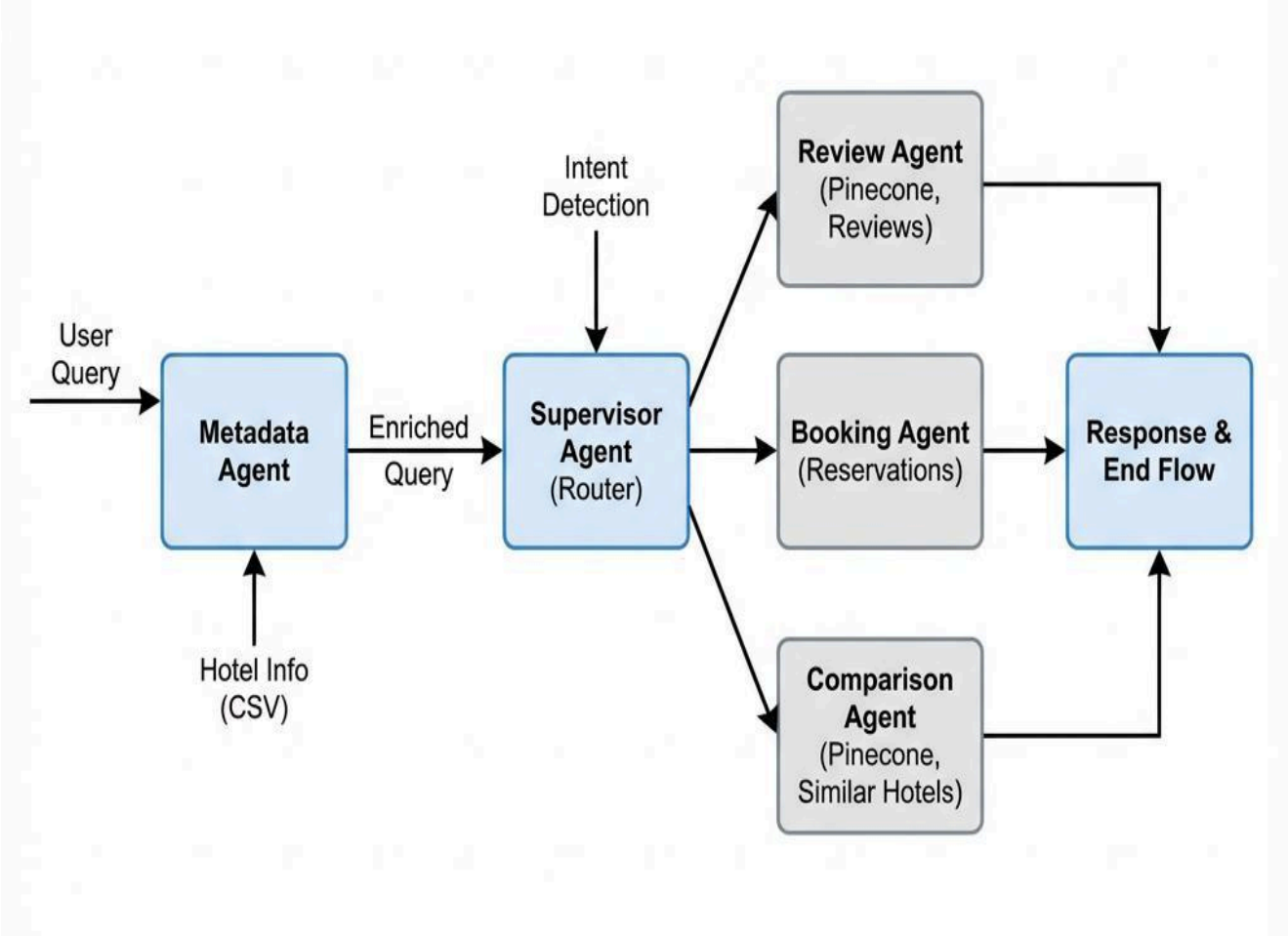
Model Pipeline

System Overview



Detailed Agent Flow

1. Metadata Agent

The Metadata Agent serves as the initial processing layer in the conversational hotel booking system. Its primary responsibility is to enrich user queries with relevant hotel context before passing them to the Supervisor for routing to specialized agents.

The Metadata Agent operates through four sequential steps:

- **Context Cache Verification**
  - The agent first checks whether the conversation already contains cached hotel information. This prevents redundant data retrieval by verifying if the specified hotel_id has been previously loaded and if the associated hotel details are already available in the conversation context.

- **Hotel Information Retrieval**
  - When hotel data is not cached, the agent reads from the hotels.csv database using the provided hotel_id. It extracts key attributes including the hotel name, star rating, physical address, and description. This information is then cached in the conversation context for subsequent use.

- **Query Enrichment**
  - The agent analyzes the user's query for contextual keywords and enriches it with hotel-specific information. For example, a generic query like "amenities" becomes "amenities (Hotel Name)", providing downstream agents with clearer context for generating accurate responses.

- **State Metadata Update**
  - Finally, the agent updates the state object with comprehensive metadata, including the hotel_id, hotel_name, complete hotel_info object, and the resolved (enriched) query. This enriched state is then passed to the Supervisor for appropriate routing.

Input/Output

Input: State containing hotel_id and user messages

Output: Enriched state with full hotel context, ready for Supervisor processing

2. Supervisor Agent

- **Overview of the Supervisor:**
  The Supervisor acts as the central routing mechanism in the conversational hotel booking system. After receiving enriched context from the Metadata Agent, it analyzes the user's intent and directs the conversation flow to the appropriate specialized agent.

- **Functionality: Intent Detection**
  - The Supervisor performs intent detection by analyzing the user's message.
  - It identifies relevant keywords that indicate the user's purpose.

- **Intent Categories and Trigger Keywords:**
  - **Review:** Trigger Keywords: `reviews`, `ratings`, `feedback`.
    - *Description:* User wants to read or inquire about guest experiences and hotel ratings.
  - **Booking:** Trigger Keywords: `book`, `reserve`.
    - *Description:* User intends to make a reservation or inquire about availability.
  - **Comparison:** Trigger Keywords: `amenities`, `similar`, `location`.
    - *Description:* User wants to compare hotel features, find alternatives, or explore location-based information.

- **Intent Classification Logic:**
  - Examines the resolved query from the Metadata Agent.
  - Matches the query against known trigger keywords for each category.
  - The intent is determined purely by the content of the user's message.

- **Routing Logic:**
  - Upon detecting the user's intent, the Supervisor updates the state object with two key values:
    - `state["intent"]`: Stores the detected intent for downstream processing.
    - `state["route"]`: Determines which specialized agent receives the query.
- **Input/Output:**
  - **Input:** Enriched state containing the resolved query from the Metadata Agent.
  - **Output:** State updated with detected intent and routing destination, passed to the corresponding specialized agent (Review Agent, Booking Agent, or Comparison Agent).

3. Review Agent

The Review Agent is a specialized component responsible for handling all review-related queries within the conversational hotel booking system. It leverages Pinecone's vector database to retrieve semantically similar reviews based on user queries and delivers contextually relevant.

The Review Agent operates through three sequential steps:

- **Vector-Based Review Retrieval**
    - The agent queries the Pinecone Reviews Index using the user's message as the search query.
    - Pinecone performs semantic similarity matching to retrieve review documents that are most relevant to what the user is asking about, rather than relying on simple keyword matching.
- **Hotel-Specific Filtering**
    - After retrieving semantically similar reviews from Pinecone, the agent filters the results to include only reviews associated with the current hotel.
    - This is accomplished by comparing each document's metadata `hotel_id` against the `hotel_id` provided in the state, ensuring users receive information specific to the hotel they are inquiring about.
- **Intent-Based Response Generation**
    - The agent analyzes the user's query to determine the specific type of review information requested and generates an appropriate response:
    - **Sub-Intent: Recent Reviews**
        - **Behavior:** Identifies and displays the latest guest feedback
        - **Output:** 2-3 most recent reviews with ratings and excerpts
    - **Sub-Intent: Summary**
        - **Behavior:** Synthesizes all filtered reviews into a cohesive overview
        - **Output:** LLM-generated summary highlighting common themes, strengths, and areas of concern
    - **Sub-Intent: General Query**
        - **Behavior:** Answers specific questions using review content as context
        - **Output:** Contextual response based on relevant review information

Input/Output

- **Input:** Hotel ID and user query (routed from Supervisor)
- **Output:** Formatted response containing either individual reviews, an AI-generated summary, or a contextual answer based on review data. The conversation then terminates (END state).

4. Comparison Agent

The Comparison Agent is a specialized component responsible for handling hotel comparison queries and general hotel information requests. It utilizes both the Pinecone vector database for finding similar hotels and CSV-based hotel data for answering specific questions about the current hotel.

The Comparison Agent operates by first detecting the query type and then following the appropriate processing path.

- Query Type Detection
    - The agent analyzes the user's message to determine which type of information is being requested:
        - Similar hotels queries — User wants to find alternative or comparable hotels
        - Hotel information queries — User wants details about the current hotel (amenities, location, features, etc.)
- Similar Hotels Processing
    - When the user asks for similar or alternative hotels, the agent performs the following steps:
        - Retrieves the current hotel's description from the `hotel_info` metadata
        - Queries Pinecone using the hotel description to find semantically similar hotels
        - Filters out the current hotel from the results to avoid redundancy
        - Returns the top 3 most similar hotels with formatted details including hotel name, star rating, description, and booking link
- Hotel Information Processing
    - When the user asks about the current hotel's features or details, the agent:
        - Retrieves relevant documents from the CSV database
        - Filters results to match the exact `hotel_id`
        - Generates a natural language response using an LLM based on the retrieved hotel information

Input/Output

- **Input:** Hotel ID, user query, and `hotel_info` (routed from Supervisor via Metadata Agent)
- **Output:** Formatted response containing either a list of similar hotel recommendations or detailed information about the current hotel. The conversation then terminates (END state).

5. Booking Agent:

Yet to implement this the workflow

Conversation Context

The conversation context is a thread-specific data structure that maintains session state throughout a user's interaction with the hotel booking system. It is keyed by a unique `thread_id`, allowing the system to manage multiple concurrent conversations while preserving context for each user session.

The conversation context serves several key functions:

- Eliminates redundant database queries by caching hotel information after initial retrieval.
- Enables context-aware responses by maintaining awareness of previous questions and answers.

The conversation context stores the following information for each active thread:

- **Hotel Identification**
  - `hotel_id`: The unique identifier for the hotel currently being discussed in the conversation.
  - `hotel_name`: Cached hotel name for quick reference without repeated database lookups.
- **Hotel Information Cache**
  - `hotel_info`: Complete hotel data object containing: name, description, star_rating, and address.
- **Conversation History**
  - `questions`: Sequential list of user queries submitted during the session, enabling the system to track what information has been requested.
  - `conversation_pairs`: Complete conversation history stored as query-response tuples, preserving the full dialogue between the user and the system.

Purpose

The conversation context serves several key functions:

- Eliminates redundant database queries by caching hotel information after initial retrieval.
- Enables context-aware responses by maintaining awareness of previous questions and answers.
- Supports multi-turn conversations where users can ask follow-up questions without re-specifying the hotel.
- Allows the system to reference prior interactions when generating responses.

Scalability Considerations

Overview
The conversational hotel booking system has been designed with scalability in mind, ensuring reliable performance as user demand and data volume grow. The current architecture addresses several key scalability requirements.
Current Architecture Capabilities

- **Multiple Concurrent Users**
  The system implements thread-based isolation, allowing numerous users to interact with the booking assistant simultaneously. Each user session operates independently within its own thread, preventing cross-contamination of conversation data and ensuring consistent experiences regardless of system load.
- **Multiple Hotels**
  Hotel data is separated using unique hotel_id identifiers throughout the system. This separation enables the platform to support an extensive hotel inventory without architectural modifications. Each query, review retrieval, and comparison operation is scoped to the appropriate hotel context.
- **Long Conversations**
  The architecture includes robust message history management, enabling users to engage in extended multi-turn conversations. The system maintains conversation pairs and query history, allowing for complex interactions that span multiple questions and follow-ups without losing context or degrading response quality.

Context Persistence
The MemorySaver checkpointer ensures that conversation context is preserved reliably across interactions. This persistence mechanism safeguards against data loss and enables the system to maintain session state even during high-traffic periods or unexpected interruptions.

Bias and Failure Detection

System Overview

This MLOps pipeline provides automated quality assurance for the hotel chatbot by analyzing responses for bias and monitoring system health through failure detection. The system combines LLM-based intelligent analysis with embedding similarity techniques to identify responses that may mislead users or misrepresent hotel quality.

The pipeline consists of two primary components:

- **Bias Detection Pipeline (HotelBiasDetection)** — Analyzes chatbot responses for three distinct bias types that could affect user trust and decision-making accuracy.
- **Failure Monitoring System (HotelFailureDetection)** — Continuously monitors bias metrics against configurable thresholds and sends automated email alerts when issues are detected.

Architecture and Workflow

The bias detection system operates as a post-processing pipeline that evaluates chatbot responses after they are generated. The workflow proceeds through the following stages:

- **Stage 1: Data Ingestion** — The system loads chatbot responses from a Parquet file containing `hotel_id`, `question`, and `answer` fields for each interaction.
- **Stage 2: Bias Analysis** — For each hotel, the pipeline fetches corresponding reviews from Pinecone and performs LLM-based bias analysis to identify potential issues.
- **Stage 3: Result Generation** — Analysis results are stored in multiple formats (JSON, Parquet, CSV) along with summary statistics for monitoring purposes.
- **Stage 4: Failure Monitoring** — The failure detection system evaluates metrics against predefined thresholds and sends appropriate alerts via email.

Bias Detection Pipeline

Configuration Parameters

The bias detection system operates with the following configurable thresholds:

- **min_reviews_threshold (10)** — Minimum number of reviews required before data is considered sufficient for confident claims
- **rating_disparity_threshold (4.5)** — Rating above which negative response tone is flagged as suspicious
- **embedding_similarity_threshold (0.75)** — Semantic similarity threshold for sparse data phrase matching
- **severity_threshold (0.85)** — Minimum confidence score required to flag a response as biased

Detection Workflow

- **Step 1: Data Loading**
  - The pipeline loads `chatbot_responses.parquet` and extracts a DataFrame containing `hotel_id`, `question`, and `answer` columns for analysis.
- **Step 2: Review Retrieval**
  - For each unique hotel in the dataset, the system queries the Pinecone index via REST API to retrieve associated reviews.
  - The query filters by `hotel_id` (string match) and `data_type="review"`, fetching up to 10,000 reviews per hotel.
  - Retrieved metadata includes `overall_rating`, `service_rating`, `cleanliness_rating`, and other rating dimensions.
- **Step 3: LLM-Based Bias Analysis**
  - The system uses GPT-4o-mini to analyze each response against the retrieved review data, checking for three mutually exclusive bias types described below.

Bias Types

Bias Type 1: Over-Reliance on Negative Feedback

This bias occurs when a chatbot response disproportionately emphasizes negative aspects of a hotel while ignoring the majority of positive reviews.

- **Detection Criteria:**
  - Response sentiment analysis indicates negative tone (`negative_prob` > 0.6)
  - Actual review distribution shows more positive reviews than negative
- **Example Scenario:**
  - A hotel has a 4.3 average rating with 80% of reviews rated 4-5 stars. The chatbot responds with "Guests complain about poor service and dirty rooms." This response is flagged because it ignores the overwhelmingly positive feedback from the majority of guests.
- **Why This Matters:**
  - Over-reliance on negative feedback can unfairly damage a hotel's reputation and lead users to reject otherwise excellent options based on cherry-picked complaints.

Bias Type 2: Missing Data Acknowledgment

This bias occurs when the chatbot makes definitive claims about a hotel without acknowledging that the underlying data is limited or insufficient.

- **Detection Criteria:**
  - Number of available reviews falls below the minimum threshold (default: 10)
  - Response contains no caveats such as "based on limited reviews" or "few guests mentioned"
- **Two-Stage Detection Process:**
  - **LLM Analysis** — The system first checks whether the response makes confident, definitive statements about the hotel.
  - **Embedding Verification** — The response is compared against a set of sparse data acknowledgment phrases using semantic similarity. If the similarity score falls below the threshold (0.75), the response is flagged for lacking appropriate caveats.
- **Example Scenario:**
  - A hotel has only 3 reviews available. The chatbot responds with "This hotel is excellent and guests love it!" The embedding similarity to phrases like "limited reviews" scores 0.23, well below the 0.75 threshold. This response is flagged because it presents confident claims without acknowledging the insufficient data.

- **Why This Matters:**
    - Users deserve to know when recommendations are based on limited information so they can adjust their confidence accordingly and seek additional sources if needed.

Bias Type 3: Rating Disparity

This bias occurs when the chatbot's response tone contradicts the hotel's numerical rating data, typically by emphasizing negatives for highly-rated hotels.

- **Detection Criteria:**
    - Hotel average rating is objectively good (≥ 4.5)
    - Response sentiment emphasizes problems or negative aspects
- **Example Scenario:**
    - A hotel has a 4.7 average rating based on 250 reviews. The chatbot responds with "The hotel has noise issues and outdated rooms that guests complain about." This response is flagged because its negative tone contradicts the excellent rating that reflects overall guest satisfaction.
- **Why This Matters:**
    - Rating disparity can mislead users by creating a false impression of hotel quality, potentially steering them away from well-reviewed properties based on minor issues that most guests did not find significant.

Embedding-Based Sparse Data Verification

The embedding verification step serves as a secondary check to catch cases where the LLM analysis might miss implicit acknowledgments or fail to flag inappropriately confident claims.

- **Process:**
    - Generate embeddings for the chatbot response using OpenAI's `text-embedding-3-small` model
    - Compare against pre-defined sparse data acknowledgment phrases
    - Calculate cosine similarity scores
    - Flag responses with similarity below the threshold as potentially lacking appropriate caveats
- This dual-layer approach (LLM + embeddings) improves detection accuracy by combining semantic understanding with pattern-based verification.

Result Storage

Per-Hotel Results (`hotel-bias-scores.json`)

- Stores detailed bias analysis for each hotel including review counts, average ratings, detected bias types, severity levels, confidence scores, and explanations.

Detailed Results (`hotel-bias-results.csv`)

- Contains row-level analysis with `hotel_id`, response text, question, review counts, ratings, bias detection flags, bias types, confidence scores, and sentiment probabilities.

Summary Statistics (`bias-summary.json`)

- Aggregates overall metrics including timestamp, total hotels analyzed, number of biased hotels, overall bias rate, distribution of bias types, count of low-review hotels, Type 2 detection rate, model used, and configuration parameters.

Failure Detection System

Purpose

The failure detection system provides automated monitoring of bias detection results, triggering alerts when metrics exceed acceptable thresholds to enable rapid response to quality degradation.

Threshold Configuration

The system monitors against three configurable thresholds:

- **max_bias_rate (0.3)** — Maximum acceptable percentage of hotels with detected bias (30%)
- **max_rating_disparity (5)** — Maximum acceptable count of rating disparity bias instances
- **max_over_reliance_negative (3)** — Maximum acceptable count of over-reliance on negative feedback instances

Monitoring Workflow

- Load the `bias-summary.json` file generated by the bias detection pipeline
- Parse key metrics including `total_hotels`, `biased_hotels`, `bias_rate`, and `bias_distribution`
- Evaluate each metric against its corresponding threshold
- Generate appropriate alert status (SUCCESS if all thresholds pass, FAILURE if any threshold is exceeded)
- Send email notification with detailed status information

Email Alert System

- The system uses Gmail SMTP with SSL encryption (Port 465) for sending alerts.
- Configuration requires three environment variables: `SENDER_EMAIL`, `SENDER_PASSWORD` (Gmail app password), and `RECEIVER_EMAIL`.

Complete Execution Flow

Phase 1: Bias Detection

Execute `python hotel_bias_detection.py` to perform the following operations:

- Load chatbot responses from Parquet file
- Iterate through each hotel to fetch reviews from Pinecone
- Analyze responses using LLM for bias detection
- Run embedding-based sparse data verification for hotels with fewer than 10 reviews
- Store individual hotel results
- Generate and save summary statistics in JSON, Parquet, and CSV formats

Phase 2: Failure Detection

Execute `python hotel_failure_detection.py` to perform the following operations:

- Load bias summary statistics
- Evaluate overall bias rate against 30% threshold
- Evaluate rating disparity count against threshold of 5
- Evaluate over-reliance negative count against threshold of 3
- Send email alert indicating SUCCESS or FAILURE status with detailed metrics

RAG Performance Evaluation System

System Overview

The RAG Performance Evaluation System is a specialized testing framework engineered to validate the functional competence of the HotelIQ chatbot. While standard unit tests verify code logic, this system assesses the **semantic quality** of the Retrieval-Augmented Generation (RAG) pipeline.

Given the non-deterministic nature of LLMs, traditional string-matching evaluation is insufficient. This system employs the **LLM-as-a-Judge** paradigm using the **Ragas** framework, where a highly capable model (GPT-4o) evaluates the chatbot's outputs against retrieved context to quantify accuracy, relevance, and hallucination rates.

Primary Objectives:

- **Quantify Trustworthiness:** Measure how often the bot "hallucinates" information not present in the source text.
- **Validate Retrieval:** Ensure the vector database and metadata lookups are finding the correct hotel policies and reviews.
- **Prevent Regression:** Provide a baseline metric (Pass Rate) that must be maintained as the model pipeline evolves.

Architecture and Workflow

The evaluation operates as a decoupled pipeline that mirrors the production environment but runs offline for batch processing.

Workflow Stages:

- **Stage A: Synthetic Ground Truth Generation**
  The system programmatically generates a diverse set of test queries (Comparison, Review, Edge Cases) rooted in the actual data of a specific "Golden Set" hotel (e.g., Hilton Boston Park Plaza). This ensures questions are answerable if the system functions correctly.
- **Stage B: Inference Execution**
  The Model Evaluation Runner executes the agent graph against the test dataset. Unlike the production API, this runner extracts internal state artifacts—specifically the retrieved_context—which is invisible to end-users but critical for grading.
- **Stage C: Automated Grading**
  The Ragas Grading Wrapper loads the inference results and utilizes OpenAI's GPT-4o-mini to score each response. It compares the User Query, Chatbot Answer, and Retrieved Context to calculate multi-dimensional metrics.

Component Breakdown

Validation Query Generator (generate_validation_queries.py)

This component constructs a challenging, domain-specific test dataset. It avoids generic questions in favor of precise queries that require specific data retrieval to answer.

- **Logic:** It uses a template-based approach to inject the specific target hotel name into three query categories:
    - *Comparison Queries:* Test the logic for comparing attributes (e.g., "How does the location of Hilton Boston Park Plaza compare to the Sheraton?").
    - *Review Queries:* Test semantic search over guest sentiment (e.g., "Do guests complain about noise at this hotel?").
    - *Edge Case Queries:* Test strict metadata lookup (e.g., "What is the exact phone number?").
- **Output:** A Parquet file containing the "Golden Dataset" for the target hotel.

Model Evaluation Runner (model_evaluation_runner.py)

This script acts as the test harness. To ensure accurate measurement of the RAG pipeline's latency and logic, it bypasses the FastAPI web server and invokes the LangGraph agents directly.

- **Context Capture:** A critical feature is the extraction of the `retrieved_context` key from the agent state. This captures the exact raw text chunks (reviews, policy text) the LLM used to generate its answer, allowing the grader to verify if the answer was derived from this text or hallucinated.
- **State Isolation:** It initializes a unique `thread_id` for every test query, ensuring that context from previous questions does not bleed into and corrupt subsequent tests.

Ragas Grading Wrapper (ragas_eval_wrapper.py)

This is the analytical engine of the pipeline. It configures the Ragas evaluators with a specific LLM and Embedding model configuration to ensure consistency with the production environment.

- **Judge Configuration:** Uses `ChatOpenAI` (GPT-4o-mini) with `temperature=0` for deterministic grading.
- **Embedding Alignment:** Uses `OpenAIEmbeddings` (text-embedding-3-large) to match the vector space used by the Pinecone index, ensuring that relevance scores accurately reflect the system's retrieval performance.

Key Performance Metrics

The system evaluates the chatbot on four critical axes, each diagnosing a specific part of the RAG pipeline:

- **Faithfulness (Hallucination Metric)**
    - **Definition:** Measures the extent to which the claims in the generated answer can be inferred from the retrieved context.
    - **Goal:** A score of 1.0 means the bot only stated facts present in its database.
    - **Example Failure:** The retrieved text says "Pool closed for renovation," but the bot answers "Enjoy our beautiful pool." (Low Faithfulness).
- **Context Recall (Retrieval Metric)**
    - **Definition:** Measures if the retrieval system successfully found the information needed to answer the question.
    - **Goal:** A score of 1.0 means the database returned all necessary details.
    - **Example Failure:** User asks "Is breakfast included?", but the retrieved documents only talk about room size. (Low Recall).
- **Answer Relevancy (Quality Metric)**
    - **Definition:** Measures how pertinent the response is to the user's query, ignoring correctness.
    - **Goal:** High scores indicate the bot understood the intent (e.g., didn't answer a pricing question with location info).
- **Context Precision (Ranking Metric)**
    - **Definition:** Evaluates the signal-to-noise ratio. It checks if the relevant documents were ranked at the top of the retrieval list or buried by irrelevant text.

Integration Dependencies

The Evaluation System is tightly coupled with the Data Pipeline to ensure valid tests:

- **Local Data Requirement:** It relies on the presence of `hotels.csv` (generated by the ETL process) in the local environment to simulate the Metadata Agent's lookups.
- **Vector Database:** It requires a live connection to the `hotel-iq-reviews` Pinecone index to verify the Review Agent's performance.

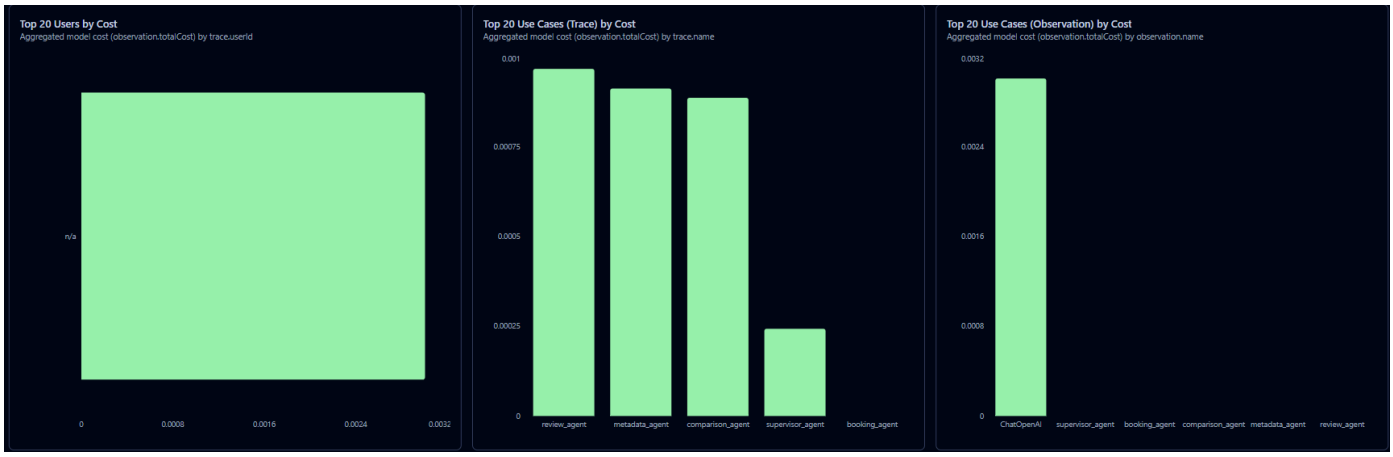
This dependency ensures that the evaluation reflects the real-world state of the application's data, rather than a mocked or stale dataset.

Key Technologies

Component	Technology
LLM Analysis	OpenAI GPT-4o-mini
Embeddings	OpenAI text-embedding-3-small
Vector Database	Pinecone (via REST API, 3072-dimensional vectors)
Data Processing	Pandas, NumPy,ragas
Email Alerts	Gmail SMTP (SSL/TLS)
Storage Formats	Parquet, CSV, JSON

Langfuse Model Observability Dashboard

Dashboard 1: Cost Analysis by User and Use Case



Purpose: Provides aggregated cost analysis to identify high-cost operations and users.

Panels:

1.1 Top 20 Users by Cost

- **Metric**: Aggregated model cost by trace.userId
- **Description**: Tracks API consumption costs per user, enabling identification of heavy users and implementation of usage-based billing or rate limiting.
- **Business Value**: Supports cost allocation, user tier management, and identification of anomalous usage patterns.

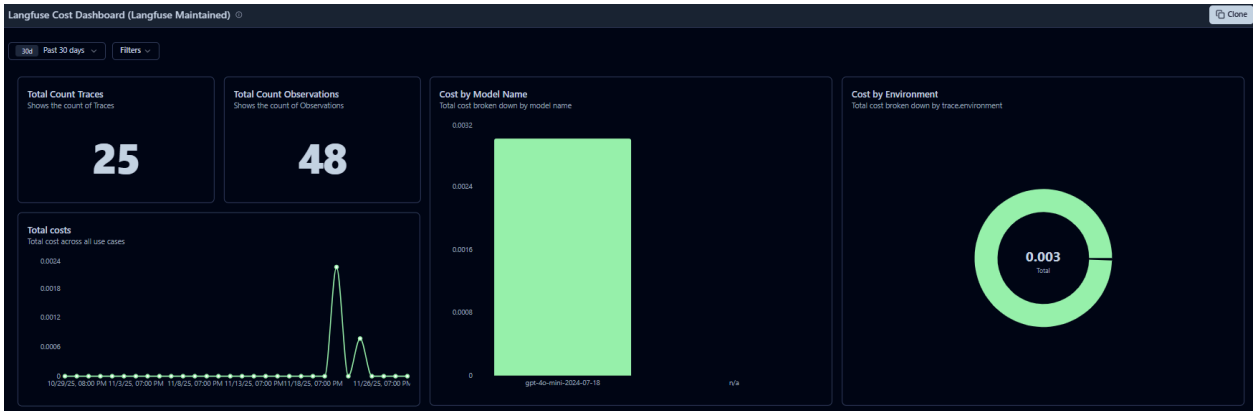
1.2 Top 20 Use Cases (Trace) by Cost

- **Metric**: Aggregated model cost by trace.name
- **Description**: Analyzes cost distribution across different agent workflows in the HotelIQ system. Traces represent complete user interactions handled by our five specialized agents:
    - `review_agent`: Retrieves and analyzes hotel reviews
    - `metadata_agent`: Extracts and processes hotel metadata
    - `comparison_agent`: Compares multiple hotels based on user criteria
    - `supervisor_agent`: Routes requests to appropriate agents
    - `booking_agent`: Handles reservation workflows
- **Business Value**: Identifies expensive workflows for optimization, informs pricing strategies, and guides resource allocation decisions.

1.3 Top 20 Use Cases (Observation) by Cost

- **Metric**: Aggregated model cost by observation.name
- **Description**: Drills down to individual LLM API calls (observations) within traces. Each observation represents a single generation from ChatOpenAI or other LLM operations.
- **Business Value**: Enables granular cost optimization by identifying specific expensive API calls that may benefit from prompt engineering, caching, or model selection changes.

Dashboard 2: Langfuse Cost Dashboard



Purpose: High-level overview of system-wide costs and usage patterns maintained by Langfuse.

Panels:

2.1 Total Count Traces

- **Metric**: Total number of traces
- **Description**: Displays the aggregate count of user interactions tracked through the system. Each trace represents one complete user journey through the HotelIQ workflow.
- **Current Value**: 25 traces tracked

2.2 Total Count Observations

- **Metric**: Total number of observations (LLM API calls)
- **Description**: Shows the cumulative number of individual LLM generations across all traces. The observation-to-trace ratio indicates system complexity.
- **Current Value**: 48 observations (ratio: 1.92 observations per trace)

2.3 Total Costs (Time Series)

- **Metric**: Cost over time
- **Description**: Time-series visualization of cumulative costs, enabling identification of usage spikes, cost trends, and anomalies. Critical for budget monitoring and capacity planning.
- **Key Insight**: Shows concentrated usage patterns with peak activity, indicating batch processing or testing phases.

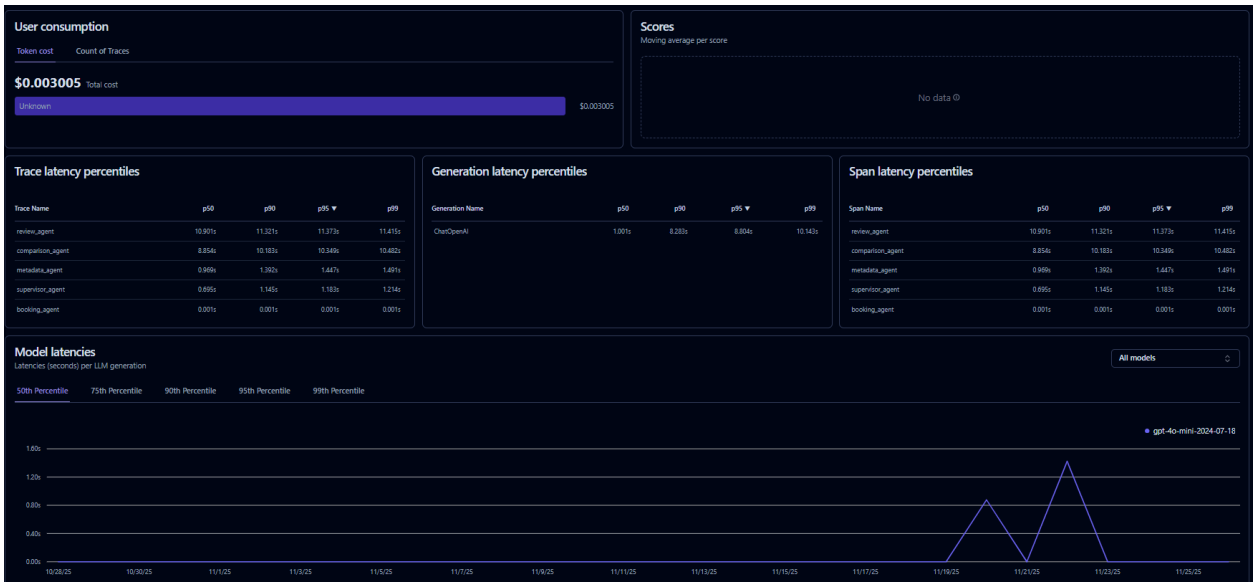
2.4 Cost by Model Name

- **Metric**: Aggregated cost per model
- **Description**: Breaks down costs by LLM model used. Supports multi-model strategies and cost optimization through model selection.
- **Current Model**: gpt-4o-mini-2024-07-18 (optimized for cost-efficiency)

2.5 Cost by Environment

- **Metric**: Cost distribution across environments
- **Description**: Segments costs by deployment environment (development, staging, production). Essential for accurate cost attribution and preventing development costs from skewing production metrics.

Dashboard 3: Performance and Quality Metrics



Purpose: Monitors system performance, latency characteristics, and quality indicators.

Panels:

3.1 User Consumption

- **Metrics**: Token cost and trace count per user
- **Description**: Visualizes resource consumption at the user level, combining both token usage costs and request volume. Supports usage-based pricing models and user behavior analysis.
- **Application**: Identifies power users, detects abuse, and informs user tier definitions.

3.2 Trace Latency Percentiles

- **Metrics**: p50, p90, p95, p99 latency by trace name
- **Description**: Distribution of end-to-end response times for complete user workflows. Higher percentiles (p95, p99) reveal tail latencies that affect user experience.
- **Current Findings**:
  - `review_agent`: 10.9s - 11.4s (highest latency - retrieval-heavy)
  - `comparison_agent`: 8.8s - 10.4s
  - `metadata_agent`: 0.9s - 1.4s
  - `supervisor_agent`: 0.6s - 1.2s (routing overhead)
  - `booking_agent`: 0.001s (minimal processing)

3.3 Generation Latency Percentiles

- **Metrics**: p50, p90, p95, p99 latency by generation name
- **Description**: Isolates LLM API response times from total trace latency. Helps differentiate between model latency and application overhead.
- **Use Case**: Identifies whether slow traces are due to LLM response times or application logic.

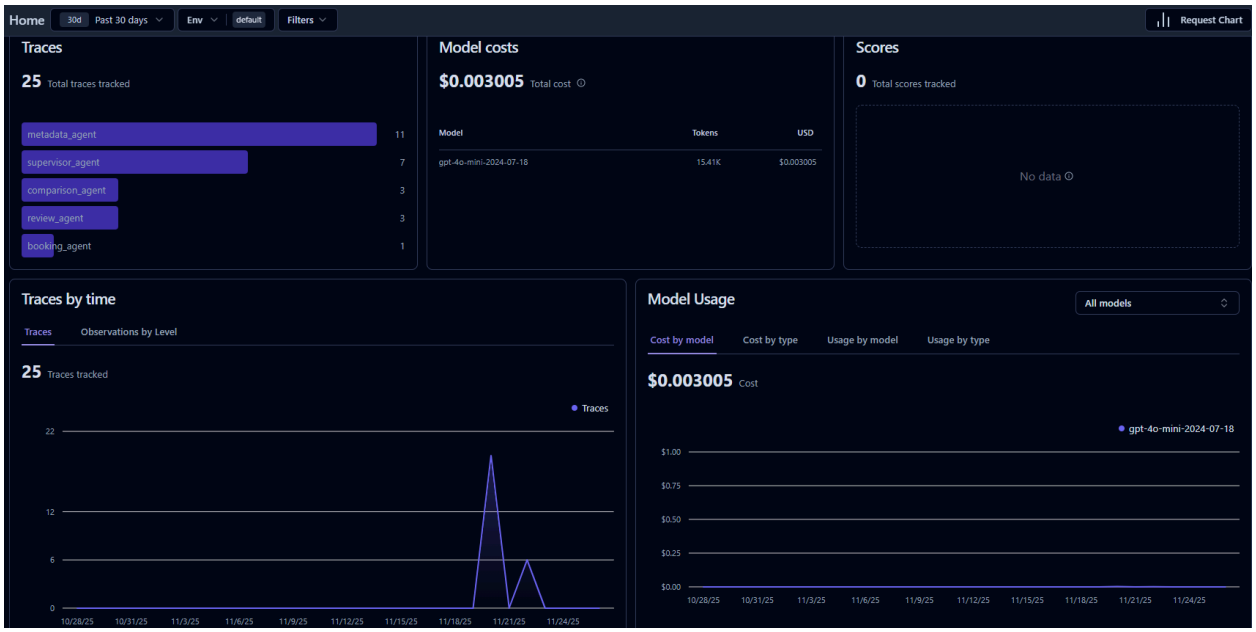
3.4 Span Latency Percentiles

- **Metrics**: p50, p90, p95, p99 latency by span name
- **Description**: Measures latency of individual operations within traces (database queries, API calls, processing steps). Most granular performance view.
- **Optimization Strategy**: Identify and optimize slowest spans first for maximum impact.

3.5 Model Latencies

- **Metric**: LLM latency over time (50th to 99th percentile)
- **Description**: Time-series visualization of model response times. Helps identify model performance degradation or improvements over time.
- **Monitoring Value**: Detects when OpenAI or other providers experience latency issues.

Dashboard 4: Home Dashboard (System Overview)



Purpose: Unified view of key system metrics for daily operations monitoring.

Panels:

4.1 Traces

- **Metric**: Total trace count with breakdown by type
- **Description**: Displays trace volume and distribution across agent types, revealing usage patterns and most-utilized workflows.
- **Current Distribution**:
  - `metadata_agent`: 44% (most frequently invoked)
  - `supervisor_agent`: 28% (routing/orchestration)
  - `comparison_agent`: 12%
  - `review_agent`: 12%
  - `booking_agent`: 4%
- **Insight**: Metadata extraction is the most common operation, suggesting users frequently query hotel information.

4.2 Model Costs

- **Metrics**: Total cost, model name, token count, USD cost
- **Description**: Consolidated view of model usage costs with token consumption details. Central metric for budget tracking.
- **Current Metrics**:
  - Model: gpt-4o-mini-2024-07-18
  - Tokens: 15.41K
  - Cost: $0.003005
- **Efficiency**: $0.000195 per trace (highly cost-effective)

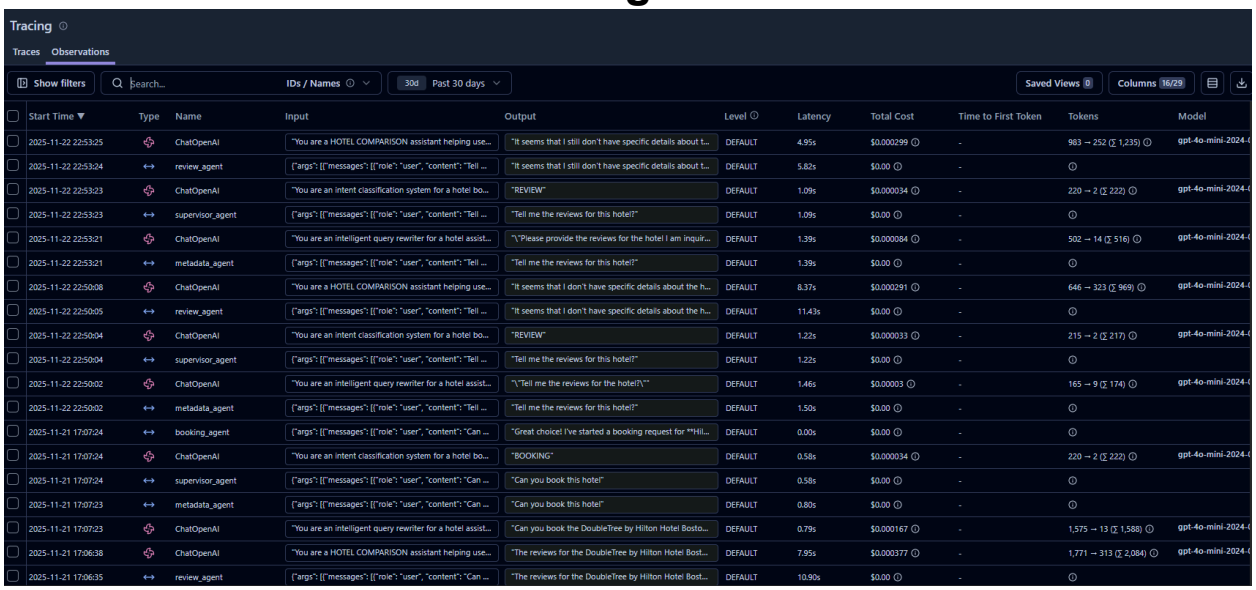
4.3 Traces by Time

- **Metric**: Trace volume over time
- **Description**: Time-series of request volume showing usage patterns, peak periods, and growth trends.
- **Pattern Analysis**: Current data shows concentrated testing/development activity with a clear peak period.
- **Capacity Planning**: Use for infrastructure scaling decisions and cost forecasting.

4.4 Model Usage

- **Metrics**: Cost by model over time (with tabs for cost by model/type and usage by model/type)
- **Description**: Tracks model selection and cost trends. Supports multi-model strategies and cost optimization.
- **Tabs Available**:
    - Cost by model: Financial tracking
    - Cost by type: Categorization by operation type
    - Usage by model: Volume metrics
    - Usage by type: Request distribution

Dashboard 5: Detailed Tracing View



Purpose: Granular trace-level debugging and analysis interface.

Interface Components:

5.1 Filters and Search

- **Functionality**: Advanced filtering by IDs, names, time ranges, and custom metadata
- **Use Cases**:
    - Debug specific user sessions
    - Analyze time-bound incidents
    - Filter by environment or model
    - Search by trace/observation names

5.2 Trace Table

Comprehensive table view with the following columns:

- **Start Time**: Timestamp of trace initiation (for temporal analysis)
- **Type**: Visual indicator distinguishing traces from generations
- **Name**: Agent or operation identifier
- **Input**: Complete user input or system prompt (critical for debugging)
- **Output**: Model response or agent output
- **Level**: Trace hierarchy level (DEFAULT, DEBUG, ERROR)
- **Latency**: Individual operation duration in seconds
- **Total Cost**: Per-trace cost in USD
- **Time to First Token**: Streaming response metric (when applicable)
- **Tokens**: Token consumption breakdown (prompt + completion)
- **Model**: LLM model used for the operation

5.3 Use Cases

- **Debugging**: Trace-by-trace analysis of failures or unexpected behavior
- **Performance Analysis**: Identify slow operations requiring optimization
- **Cost Investigation**: Drill down into expensive traces
- **Quality Assurance**: Review actual inputs/outputs for accuracy
- **Conversation Flow Analysis**: Understand multi-turn agent interactions