HotelIQ Data-Pipeline Documentation

Team Members:

- Yaksh Ajay Shah
- Devarshi Anil Mahajan
- Sarthak Vikas Sonawane
- Rajkesh Prakash Shetty
- Praveen Ramkumar
- Rakshith Reddy Kokonda

Project Overview

The **HotelIQ-Data-Pipeline** is designed to extract, transform, and load hotel-specific data for an intelligent chatbot system. The pipeline processes hotel metadata, room information, amenities, policies, and reviews, transforming this data into a structured, query-optimized format in a PostgreSQL database.

Purpose: Extract hotel metadata and reviews, transform/validate, and load into a Postgres (Cloud SQL) database for the HotelIQ chatbot RAG system.

Stack:

- Apache Airflow (orchestration via docker-compose)
- Python scripts (ETL logic)
- PostgreSQL (Cloud SQL)
- Google Cloud Storage (optional bucket utilities)

Contents of README Documentation

1. Prerequisites and Environment Setup

Prerequisites

- **Operating System**: Linux machine (project tested on Linux)
- **Docker & Docker Compose**: For running Airflow locally
- **Python**: Version 3.9+ with pip
- **Database Access**: Credentials for Cloud SQL (PostgreSQL) or local Postgres
- **Google Cloud SDK** (Optional): For Cloud SQL Proxy

Installation Steps

Step 1: Clone the Repository

git clone https://github.com/Rakshith-Reddy-K/hotel-iq.git
cd hotel-iq/data_pipeline

Explanation:

- Clones the HotelIQ repository from GitHub
- Navigates to the `data_pipeline` subdirectory where all pipeline code resides

Step 2: Install Python Dependencies

python -m pip install -r requirements.txt

Explanation:

- Installs required Python packages including:
  - `apache-airflow` - Workflow orchestration
  - `psycopg2-binary` - PostgreSQL database adapter
  - `pandas` - Data manipulation
  - `python-dotenv` - Environment variable management
  - Other dependencies for ETL operations

Step 3: Configure Environment Variables

Create a `.env` file in the `data_pipeline/` folder with the following variables:

```
# Database Configuration
DB_HOST=localhost          # Or Cloud SQL proxy host
DB_PORT=5432               # Or Cloud SQL proxy port
DB_NAME=hoteliq_db
DB_USER=your_db_username
DB_PASSWORD=your_db_password

# CSV Data Location
LOCAL_CSV_PATH=intermediate/csv

# (Optional) GCP Configuration
GCP_PROJECT_ID=your-project-id
GCS_BUCKET_NAME=hoteliq-data-bucket
# Gemini API Key
GEMINI_API_KEY = your_gemini_api_key
```

Explanation:

- *DB_ variables**: Used by `db_pool.py` to establish database connections
- **LOCAL_CSV_PATH**: Directory where CSV files are located (defaults to `intermediate/csv`)
- **GCP variables**: For optional cloud storage integration via `bucket_util.py`

Step 4: (Optional) Start Cloud SQL Proxy

If using Google Cloud SQL, configure and start the Cloud SQL Proxy:

1. Edit `scripts/run_proxy.sh` with your:
    - Project ID
    - Region
    - Instance name
    - Service account credentials path
2. Run the proxy:
   bash scripts/run_proxy.sh

Explanation:

- Cloud SQL Proxy creates a secure tunnel to your Cloud SQL instance
- The proxy runs on `localhost:5432` (or configured port)
- `db_pool.py` connects to the proxy, which forwards to Cloud SQL
- This enables secure access without whitelisting IPs

2. Steps to Run the Pipeline

Using Airflow (Recommended for Production)

Step 1: Start Docker Services
docker-compose up -d

Explanation:

- Starts Postgres, Redis, Airflow webserver, scheduler, and worker
- Runs in detached mode (`-d`) so services run in background
- Services defined in `docker-compose.yaml`

Step 2: Access Airflow Web UI

Navigate to: **http://localhost:8080**

Explanation:

- Airflow UI allows monitoring and triggering DAGs
- Default credentials typically: `airflow` / `airflow`
- DAG file: `dags/data_pipeline_airflow.py`

Step 3: Verify DAG Availability

Ensure the `data_pipeline` DAG appears in the Airflow UI:

- Check that `dags/` directory is properly mounted in `docker-compose.yaml`
- Refresh the DAG list if needed

Step 4: Prepare CSV Files

Ensure these CSV files exist in `intermediate/csv/`:

- `hotels.csv`
- `rooms.csv`
- `reviews.csv`
- `amenities.csv`
- `policies.csv`

Explanation:

- The loader expects these specific filenames
- Each CSV must have headers matching column definitions in `load_to_database.py`

- Empty cells should be empty strings (converted to NULL by loader)

Step 5: Trigger the DAG

In the Airflow UI:

1. Find the `data_pipeline` DAG
2. Toggle it to "ON"
3. Click "Trigger DAG" to start execution

Explanation:

- DAG orchestrates: Extract → Transform → Validate → Load
- Monitor task progress in the Airflow UI
- View logs for each task by clicking on task boxes

3. Code Structure and Explanation

Repository Structure

```
hotel-iq
    ├── README.md                          # Project documentation
    └── data_pipeline
        ├── dags
        │   ├── data_pipeline_airflow.py   # Airflow DAG definition
        │   └── src
        │       ├── accumulated.py         # Utilities for accumulating or aggregating data
        │       ├── batch_selection.py     #  Logic for selecting data batches for processing.
        │       ├── bucket_util.py         # GCS utilities (optional)
        │       ├── filtering.py           # Functions for filtering datasets
        │       ├── path.py                # Path utilities for consistent file location handling.
        │       ├── prompt.py              # Prompt templates or helpers
        │       ├── state_management.py    # State tracking for batch or pipeline runs
        │       ├── transform.py           # Data transformation logic
        │       ├── utils.py               # Util functions to extract hotel data
        │       └── utils_gemini.py        # Utilities for Gemini API or model integration.
        ├── docker-compose.yaml            # Airflow services configuration
        ├── requirements.txt
        ├── scripts
        │   └── run_proxy.sh
        └── sql
            ├── db_pool.py                 # Database connection pool
            ├── queries.py                 # SQL DDL and query helpers
            └── test_connection.py         # Connection test script
```

3.1 Pipeline Flow

The data pipeline executes in 6 sequential stages with some parallel processing, orchestrated by the Airflow DAG in data_pipeline_airflow.py

Stage 1: Conditional Filtering

The pipeline starts by checking if city-filtered data already exists in GCS to avoid redundant processing

If filtering is needed:

1.Downloads raw hotels.txt from GCS and filters by city (e.g., "Boston")

2.Downloads raw reviews.txt and filters reviews for city hotels

3. Uploads filtered CSVs back to GCS

If filtering exists: Skips directly to batch selection

Both paths converge at a join point before proceeding.

Stage 2: Batch Selection

The pipeline selects the next batch of unprocessed hotels using state tracking:

Downloads state.json to identify already-processed hotel IDs

Filters out processed hotels and selects next 25 (configurable via BATCH_SIZE)

Saves batch_hotels.csv to GCS

Filters reviews for the current batch hotels and saves batch_reviews.csv

Stage 3: Parallel Enrichment

Two tasks run in parallel to enrich the batch data:

Path A - Ratings Computation:

1.Calculates aggregate ratings (mean overall, cleanliness, service, etc.) from batch reviews
2.Saves batch_ratings.csv

Path B - AI Enrichment:

1. Iterates through each hotel in the batch
2. Calls Gemini Live 2.5 Flash API to extract comprehensive hotel data (rooms, amenities, policies)
3. Includes 5-second delays between API calls for rate limiting
4. Saves enrichment data as JSONL

Stage 4: Data Transformation

After both enrichment paths complete, the pipeline merges all data:

1.Loads batch hotels, enrichment JSONL, and ratings CSV
2. Skips hotels with enrichment errors
3. Merges original hotel data with AI enrichment and ratings
4.Normalizes into 5 database-ready CSVs: batch_hotels.csv, batch_rooms.csv, batch_amenities.csv, batch_policies.csv, batch_reviews.csv

Stage 5: Database Load

The pipeline loads the normalized data into PostgreSQL:

1. Creates tables if they don't exist (hotels, rooms, reviews, amenities, policies)
2. Bulk inserts data using psycopg2.extras.execute_values()

Stage 6: State Management

Finally, the pipeline updates tracking state:
1. Appends batch results to accumulated city-wide datasets in GCS
2. Updates state.json with processed hotel IDs and batch metadata

3.2 Core Components Explanation

3.2.1. Data Processing

a. transform.py - Handles data transformation and aggregation

compute_aggregate_ratings() - Computes per-hotel mean ratings from reviews

prepare_hotel_data_for_db() - Merges batch hotels, enrichment data, and ratings into normalized CSVs for database loading

b. utils.py - Provides parsing, cleaning, and data manipulation utilities

parse_raw_hotels() - Parses JSONL hotel data into DataFrames

parse_raw_reviews() - Parses JSONL review data into DataFrames

merge_hotel_data() - Merges hotel metadata with Gemini enrichment data

Safe conversion functions (safe_int(), safe_decimal(), safe_date(), etc.) for type validation

3.2.2. Data Selection & Filtering

a. filtering.py - Filters raw data by city

check_if_filtering_needed() - Checks if filtered data already exists in GCS

filter_all_city_hotels() - Filters master hotels dataset for a specific city

filter_all_city_reviews() - Filters master reviews dataset for city hotels

b. batch_selection.py - Manages batch processing

select_next_batch() - Selects next batch of unprocessed hotels based on state tracking

filter_reviews_for_batch() - Filters reviews for hotels in current batch

3.2.3 AI Enrichment

a. prompt.py - Orchestrates hotel data enrichment using AI APIs

extract_hotel_data_gemini() - Extracts hotel data using Gemini API
enrich_hotels_gemini() - Enriches hotels in batch using Gemini

b. utils_gemini.py - Gemini API client implementation

get_hotel_data_async() - Async function to get hotel data from Gemini Live 2.5 Flash API
get_hotel_data() - Synchronous wrapper for async Gemini API calls

3.2.4 State & Storage Management

a. state_management.py - Tracks processing state

update_processing_state() - Updates state.json with processed hotel IDs and batch metadata to prevent duplicate processing

b. accumulated.py - Manages accumulated results

append_batch_to_accumulated() - Appends batch results to accumulated artifacts in GCS

c. bucket_util.py - GCS integration utilities

get_bucket() - Manages GCS client connection with singleton pattern
upload_file_to_gcs() - Uploads files to GCS with retry logic
download_file_from_gcs() - Downloads files from GCS

d. path.py - Centralized path management for local and GCS storage

Defines directory constants (RAW_DIR, FILTERED_DIR, PROCESSING_DIR,PROCESSED_DIR)
Provides path resolution functions for all data layers

3.3. SQL Core Files

a. db_pool.py - Connection Pool Management

This file implements a singleton connection pool using psycopg2.pool.ThreadedConnectionPool to efficiently manage database connections across concurrent Airflow tasks

Key functions:

initialize_pool() - Creates a connection pool with 1-20 connections using environment variables (CLOUD_DB_HOST, CLOUD_DB_PORT, etc.)
get_connection() - Context manager that safely acquires connections from the pool, handles commits/rollbacks, and returns connections automatically
close_all() - Closes all connections in the pool
The pool is initialized automatically on module import

b. queries.py - Database Schema and Bulk Loading

This file contains all SQL DDL statements and bulk insert logic.

Schema Creation Functions:

create_hotels_table() - Creates the main hotels table with 24 columns including ratings, location, and metadata
create_rooms_table() - Creates rooms table with foreign key to hotels and unique constraint on (hotel_id, room_type)
create_reviews_table() - Creates reviews table with unique constraint on (hotel_id, reviewer_name, review_date)
create_amenities_table() - Creates amenities table with unique constraint on (hotel_id, category)
create_policies_table() - Creates policies table with unique constraint on hotel_id
create_all_tables() - Orchestrates creation of all five tables in dependency order

Data Loading Functions:

bulk_insert_from_csvs(csv_dir) - Main entry point that reads 5 CSV files, groups data by hotel_id, and calls insert_one_hotel_complete() for each hotel
insert_one_hotel_complete() - Atomically inserts one hotel and all related records (rooms, amenities, policies, reviews) using psycopg2.extras.execute_values() for batch inserts
clean_dict_for_db() - Converts pandas NaN values to Python None for database compatibility

Utility Functions:

list_tables() - Queries information schema to list all tables in the public schema

All INSERT statements use ON CONFLICT DO NOTHING clauses for idempotency, allowing safe re-runs without duplicate key violations

c. test_connection.py - Connection Testing

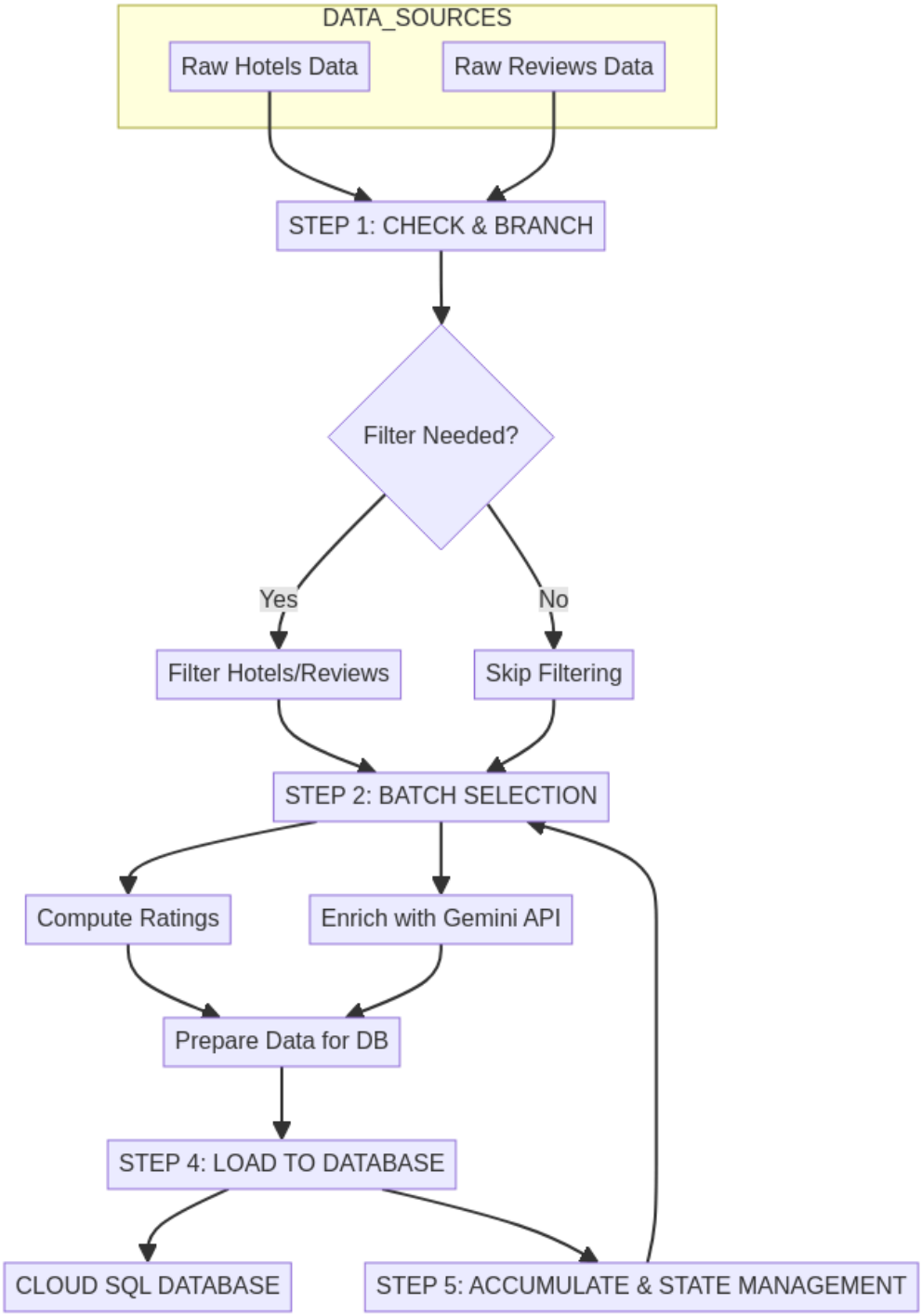
Simple utility script to verify database connectivity and list tables
imports list_tables() from queries.py and can be run standalone to test the connection pool setup.

Usage in Pipeline
These SQL files are used by the Airflow DAG in two tasks:

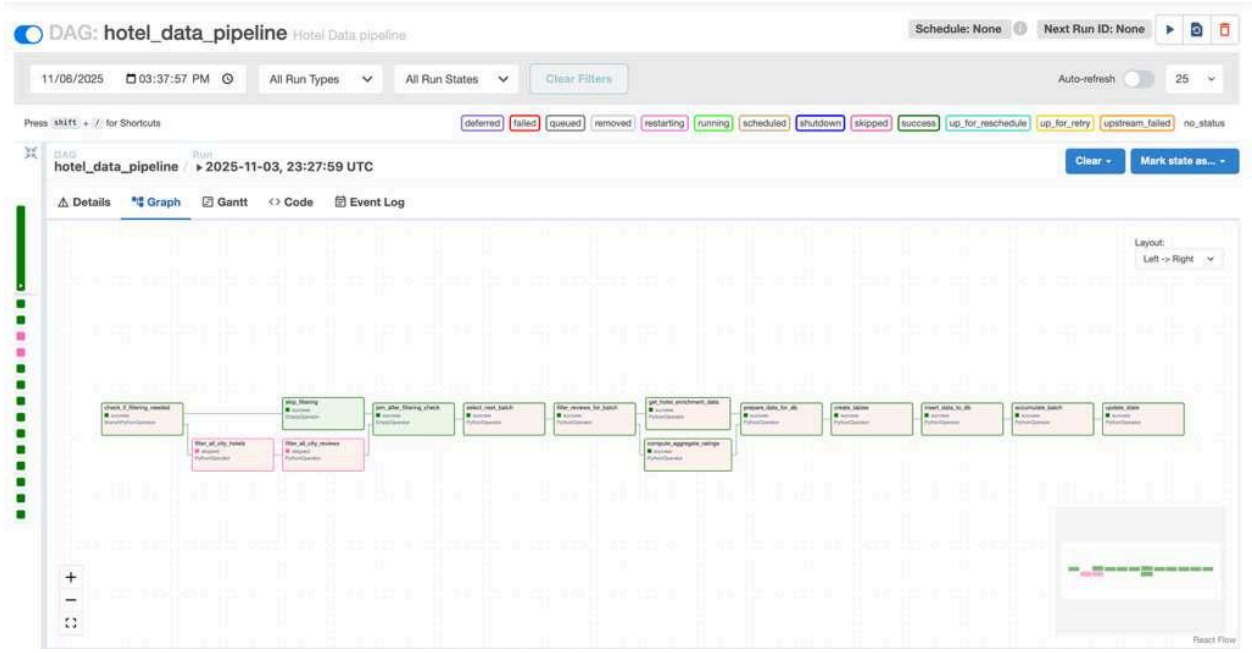
1. create_tables task - Calls create_all_tables() to ensure schema exists
2. insert_data_to_db task - Calls bulk_insert_from_csvs() to load the 5 normalized CSV files produced by the transformation stage

4. Data Flow and Architecture

4.1 End-to-End Pipeline Flow



4.2 Airflow DAG Visualization



5. Explanation of Each Task

Task 1: check_if_filtering_needed

Type: BranchPythonOperator
Purpose: Determines whether city-based filtering of hotel and review data is required before proceeding with the pipeline.
Process:

- Evaluates whether filtered city data already exists in the intermediate storage
- Uses logic from `src.filtering.check_if_filtering_needed` module
- Makes a branching decision based on the presence/absence of filtered data
- Returns the task_id of the appropriate next task to execute

Output: Branches the DAG to either filtering tasks (`filter_all_city_hotels`) or skips directly to batch selection (`select_next_batch`), optimizing pipeline execution by avoiding redundant work.

Task 2: skip_filtering

Type: EmptyOperator
Purpose: Serves as a placeholder task when filtering operations are not required.
Process:

- Performs no operations
- Simply passes control to the next downstream task
- Maintains proper task dependency structure

Output: Continues DAG execution to the next logical step without any data processing.

Task 3: filter_all_city_hotels

Type: PythonOperator
Purpose: Filters the master hotel dataset to include only hotels from the specified target city.
Process:

- Reads the complete hotel dataset from source
- Applies city-based filtering criteria to identify relevant hotels
- Extracts only hotel records matching the target city
- Writes the filtered subset to an intermediate CSV file

Output: A filtered hotel data CSV containing only hotels from the specified city, reducing data volume for downstream processing.

Task 4: filter_all_city_reviews

Type: PythonOperator
Purpose: Filters the master reviews dataset to include only reviews associated with hotels in the specified city.
Process:

- Reads the complete reviews dataset from source
- Matches reviews to hotels from the filtered city hotel list
- Extracts only reviews for hotels in the target city
- Writes the filtered reviews to an intermediate CSV file

Output: A filtered reviews CSV containing only reviews for the city's hotels, ensuring relevance for downstream analytics.

Task 5: select_next_batch

Type: PythonOperator
Purpose: Selects the next batch of hotels for enrichment and processing, enabling incremental pipeline execution.
Process:

- Reads the filtered city hotel list
- Selects a configurable batch of hotels (e.g., 25 hotels) for processing
- Updates batch state tracking to maintain progress
- Identifies hotels not yet processed

Output: A batch hotel list file containing the subset of hotels to be enriched in this iteration, supporting scalable parallel processing.

Task 6: filter_reviews_for_batch

Type: PythonOperator
Purpose: Filters reviews to include only those corresponding to hotels in the current processing batch.
Process:

- Reads the city-level filtered reviews file
- Matches reviews to the hotels in the current batch
- Filters reviews based on batch hotel IDs

- Writes batch-specific reviews to an intermediate file

Output: A batch reviews CSV containing only reviews for the hotels being processed in the current batch, reducing data size for enrichment operations.

Task 7: enrich_hotels_gemini

Type: PythonOperator
Purpose: Enriches hotel data using the Gemini API or model to extract structured information such as amenities, policies, and other features.
Process:

- Reads batch hotel data and associated reviews
- Calls Gemini API/model with appropriate prompts
- Extracts structured information from unstructured text
- Parses and validates enrichment results

Output: Enriched hotel data file (JSONL or CSV format) containing extracted amenities, policies, and LLM-generated features for downstream use.

Task 8: append_batch_to_accumulated

Type: PythonOperator
Purpose: Appends the processed batch data to accumulated city-level datasets, building the complete dataset incrementally.
Process:

- Reads batch enrichment results
- Appends data to accumulated city-level files (hotels, amenities, policies)
- Performs deduplication to ensure data integrity
- Updates tracking metadata

Output: Updated accumulated files for the city, containing all processed batches to date, supporting recovery and incremental reruns.

Task 9: compute_aggregate_ratings

Type: PythonOperator
Purpose: Computes aggregate ratings and statistics for hotels based on filtered reviews.
Process:

- Reads city or batch reviews data
- Calculates overall ratings (average, median, etc.)
- Computes category-specific ratings (cleanliness, service, location, etc.)
- Aggregates sentiment and quality metrics
- Writes aggregate ratings to a CSV file

Output: Hotel ratings CSV containing summary statistics for analytics, reporting, and chatbot responses.

Task 10: prepare_hotel_data_for_db

Type: PythonOperator
Purpose: Prepares final normalized tables (hotels, rooms, amenities, policies) for efficient database loading.
Process:

- Merges hotel, enrichment, ratings, and review data
- Normalizes data into separate relational tables
- Formats data according to database schema requirements
- Writes final CSVs for each table

Output: Multiple CSV files (hotels.csv, rooms.csv, amenities.csv, policies.csv) ready for database import.

Task 11: create_all_tables

Type: PythonOperator
Purpose: Creates all required database tables if they do not already exist, ensuring schema readiness.
Process:

- Connects to the target database
- Executes DDL (Data Definition Language) statements
- Creates tables with appropriate schema, constraints, and indexes
- Handles idempotent creation (CREATE IF NOT EXISTS)

Output: Database tables ready for data loading, preventing load failures due to missing schema.

Task 12: bulk_insert_from_csvs

Type: PythonOperator
Purpose: Loads the prepared CSV files into the database using efficient bulk insert operations.
Process:

- Reads final prepared CSV files
- Establishes database connection
- Performs batch/bulk insert operations for each table
- Handles transaction management and error handling
- Logs insertion metrics (record counts, duration)

Output: Data successfully loaded into database tables, completing the ETL pipeline and making data available for analytics and chatbot queries.

6. Workflow Summary

Data Transformation Journey

Input: Raw hotel and review data files (CSV/Text)
Output: Normalized PostgreSQL database ready for analytics and chatbot queries

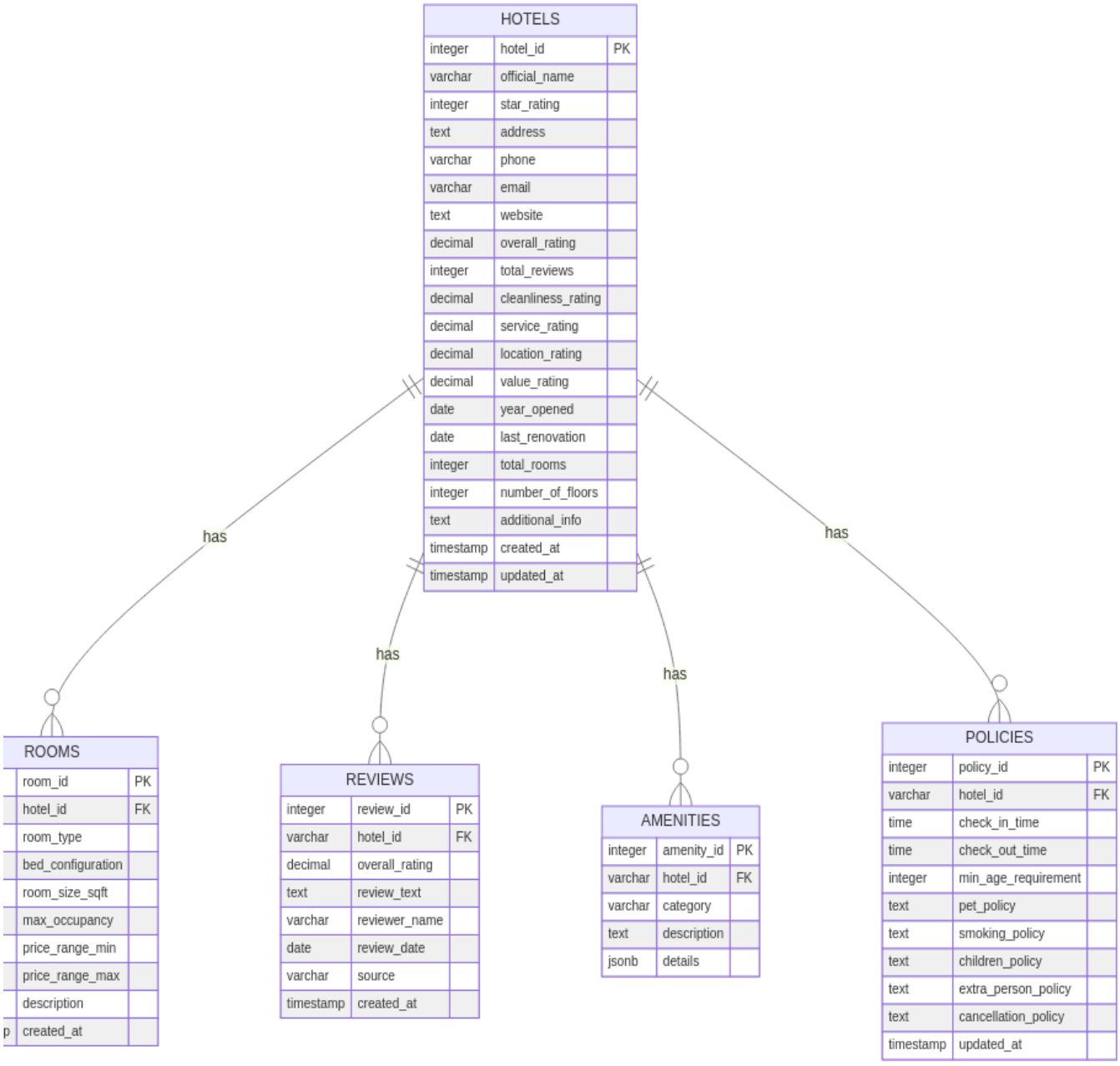
Transformations:

1. **Global → City-Specific:** All hotels/reviews → Target city data only
    - `filter_all_city_hotels`: Filter master hotel dataset by city
    - `filter_all_city_reviews`: Filter master reviews dataset by city
2. **Bulk → Batch:** City dataset → Manageable batches (25 hotels at a time)
    - `select_next_batch`: Select next hotel batch for processing
    - `filter_reviews_for_batch`: Extract reviews for current batch only
3. **Raw → Enriched:** Basic hotel data → AI-enhanced structured data
    - `enrich_hotels_gemini`: Extract amenities, policies, features via Gemini API
    - Transform unstructured text → Structured attributes
4. **Individual → Aggregated:** Hundreds of reviews → Hotel-level metrics
    - `compute_aggregate_ratings`: Calculate overall and category ratings
    - Generate summary statistics (average, sentiment scores)
5. **Scattered → Merged:** Separate enrichments → Unified hotel profiles
    - `prepare_hotel_data_for_db`: Merge ratings, enrichments, base data
    - Combine batch results into complete hotel records
6. **Denormalized → Normalized:** Flat files → Relational schema
    - Split into separate tables: hotels, rooms, amenities, policies, reviews
    - Establish proper foreign key relationships
7. **Incremental → Accumulated:** Batch results → City-wide dataset
    - `append_batch_to_accumulated`: Build up complete city data
    - Track processing state for recovery and monitoring
8. **Files → Database:** Prepared CSVs → PostgreSQL tables
    - `create_all_tables`: Establish database schema
    - `bulk_insert_from_csvs`: Load data efficiently into Cloud SQL

Pipeline Flow:

Raw Data → Filtered → Batched → Enriched → Aggregated → Normalized → Loaded → Accumulated

9. ER Diagram


HOTELS
integer hotel_id PK
varchar official_name
integer star_rating
text address
varchar phone
varchar email
text website
decimal overall_rating
integer total_reviews
decimal cleanliness_rating
decimal service_rating
decimal location_rating
decimal value_rating
date year_opened
date last_renovation
integer total_rooms
integer number_of_floors
text additional_info
timestamp created_at
timestamp updated_at
has
has
has
has
ROOMS
room_id PK
hotel_id FK
room_type
bed_configuration
room_size_sqft
max_occupancy
price_range_min
price_range_max
description
created_at
REVIEWS
integer review_id PK
varchar hotel_id FK
decimal overall_rating
text review_text
varchar reviewer_name
date review_date
varchar source
timestamp created_at
AMENITIES
integer amenity_id PK
varchar hotel_id FK
varchar category
text description
jsonb details
POLICIES
integer policy_id PK
varchar hotel_id FK
time check_in_time
time check_out_time
integer min_age_requirement
text pet_policy
text smoking_policy
text children_policy
text extra_person_policy
text cancellation_policy
timestamp updated_at

9.1 HOTELS

Purpose: The main table storing core information about each hotel property.

Attributes:

- hotel_id (PK): Unique identifier for each hotel, used to link all other tables.
- official_name: The hotel's official name as displayed to guests.
- star_rating: Hotel classification from 1 to 5 stars, indicating quality level.
- address: Full street address of the hotel property.
- phone: Primary contact phone number for the hotel.
- email: Contact email address for guest inquiries.
- website: URL of the hotel's official website.
- overall_rating: Average rating calculated from all guest reviews (0-5 scale).
- total_reviews: Count of total reviews received by this hotel.
- cleanliness_rating: Average score specifically for room and property cleanliness.
- service_rating: Average score for staff service quality and responsiveness.
- location_rating: Average score for the hotel's location convenience.
- value_rating: Average score for price-to-quality ratio (value for money).
- year_opened: The year the hotel first opened for business.
- last_renovation: Date of the most recent property renovation or major update.
- total_rooms: Total number of guest rooms in the hotel.
- number_of_floors: Number of floors in the hotel building.
- additional_info: Extra details such as unique features, historical information, or enrichment data from external APIs.
- created_at: Timestamp when this hotel record was first created in the database.
- updated_at: Timestamp of the most recent update to this hotel record.

Relationships:

- Has (to ROOMS): One hotel can have multiple room types (one-to-many relationship).
- Has (to REVIEWS): One hotel can have multiple guest reviews (one-to-many relationship).
- Has (to AMENITIES): One hotel can have multiple amenities (one-to-many relationship).
- Has (to POLICIES): One hotel has one set of policies (one-to-one relationship, though stored as separate table).

9.2 ROOMS

Purpose: Stores information about different room types and configurations available at each hotel.

Attributes:

- **room_id** (PK): Unique identifier for each room type.
- **hotel_id** (FK): Links to the parent hotel in the HOTELS table, indicating which hotel this room belongs to.
- **room_type**: Category or classification of the room (e.g., "Standard Queen", "Deluxe King", "Executive Suite").
- **bed_configuration**: Description of bed arrangements in the room (e.g., "1 King Bed", "2 Queen Beds", "1 King + 1 Sofa Bed").
- **room_size_sqft**: Size of the room measured in square feet.
- **max_occupancy**: Maximum number of guests allowed in this room type.
- **price_range_min**: Minimum nightly rate for this room type (lowest price during off-season).
- **price_range_max**: Maximum nightly rate for this room type (highest price during peak season).
- **description**: Detailed description of room features, amenities, views, and other selling points.
- **created_at**: Timestamp when this room type was added to the database.

Relationships:

- **Belongs to** (HOTELS): Each room type is associated with exactly one hotel via the **hotel_id** foreign key.

9.3 REVIEWS

Purpose: Stores individual guest reviews and ratings for hotels.

Attributes:

- **review_id** (PK): Unique identifier for each review.
- **hotel_id** (FK): Links to the hotel being reviewed in the HOTELS table.
- **overall_rating**: The reviewer's overall rating of the hotel (typically 1-5 scale).
- **review_text**: The full text of the guest's review, providing detailed feedback.
- **reviewer_name**: Name of the person who wrote the review (may be anonymized for privacy).
- **review_date**: Date when the review was posted or submitted.
- **source**: Where the review came from (e.g., "Google Reviews", "Booking.com", "TripAdvisor", "Direct").
- **created_at**: Timestamp when this review was added to the database.

Relationships:

- **Belongs to** (HOTELS): Each review is associated with exactly one hotel via the **hotel_id** foreign key.
- **Can Write**: Users can write multiple reviews for different hotels (implied relationship).

9.4 AMENITIES

Purpose: Stores information about facilities and services offered by each hotel.

Attributes:

- **amenity_id** (PK): Unique identifier for each amenity record.
- **hotel_id** (FK): Links to the hotel offering this amenity in the HOTELS table.
- **category**: Type or classification of the amenity (e.g., "WiFi", "Parking", "Dining", "Fitness", "Pool", "Business Center").
- **description**: Detailed description of the amenity, including specifics about what's offered.
- **details** (JSONB): Flexible structured data storing additional information such as:
    - Operating hours (e.g., `{"hours": "6 AM - 10 PM"}`)
    - Pricing (e.g., `{"cost": "Free"}` or `{"cost": "$45/night"}`)
    - Specifications (e.g., `{"speed": "100 Mbps"}` for WiFi)
    - Location within hotel (e.g., `{"location": "Rooftop, 12th floor"}`)

Relationships:

- **Belongs to** (HOTELS): Each amenity is associated with exactly one hotel via the **hotel_id** foreign key.

9.5 POLICIES

Purpose: Stores hotel policies, rules, and operational information that guests need to know.

Attributes:

- **policy_id** (PK): Unique identifier for each policy record.
- **hotel_id** (FK): Links to the hotel with these policies in the HOTELS table.
- **check_in_time**: Standard check-in time (e.g., "3:00 PM", "15:00").
- **check_out_time**: Standard check-out time (e.g., "11:00 AM", "11:00").
- **min_age_requirement**: Minimum age required to book a room (typically 18 or 21).
- **pet_policy**: Rules regarding pets, including fees, size restrictions, and allowed areas (e.g., "Pets allowed with $75 fee. Maximum 2 pets, 50 lbs each").
- **smoking_policy**: Smoking rules and restrictions (e.g., "Non-smoking property. $250 cleaning fee for violations").
- **children_policy**: Policies related to children, such as age limits for free stays (e.g., "Children under 12 stay free with existing bedding").
- **extra_person_policy**: Charges for additional guests beyond standard occupancy (e.g., "$25 per person per night").
- **cancellation_policy**: Terms for canceling reservations, including deadlines and fees (e.g., "Free cancellation until 48 hours before check-in. Late cancellations charged one night").
- **updated_at**: Timestamp of the most recent update to these policies

Relationships:

- **Belongs to** (HOTELS): Each policy set is associated with exactly one hotel via the **hotel_id** foreign key.

10. Key Relationships Explained

Foreign Key Constraints

What is a Foreign Key? A foreign key is a column that creates a link between two tables. It ensures that data in one table matches data in another table.

In HotelIQ:

1. **ROOMS → HOTELS**
    - Each room's **hotel_id** must match a **hotel_id** that exists in the HOTELS table
    - This prevents creating rooms for non-existent hotels
    - If a hotel is deleted, all its rooms are automatically deleted (CASCADE DELETE)
2. **REVIEWS → HOTELS**
    - Each review's **hotel_id** must match a **hotel_id** in the HOTELS table
    - Ensures reviews are always connected to valid hotels
    - If a hotel is deleted, all its reviews are automatically deleted
3. **AMENITIES → HOTELS**
    - Each amenity's **hotel_id** must match a **hotel_id** in the HOTELS table
    - Prevents orphaned amenity records
    - If a hotel is deleted, all its amenities are automatically deleted
4. **POLICIES → HOTELS**
    - Each policy's **hotel_id** must match a **hotel_id** in the HOTELS table
    - Ensures policies are always tied to a specific hotel
    - If a hotel is deleted, its policies are automatically deleted

One-to-Many Relationships

HOTELS (1) ──→ (Many) ROOMS

- **Meaning**: One hotel can have many different room types
- **Example**: Grand Boston Hotel has Standard Queen, Deluxe King, and Executive Suite rooms
- **Real-world**: A single hotel property offers multiple room options for guests

HOTELS (1) ──→ (Many) REVIEWS

- **Meaning**: One hotel can receive many guest reviews
- **Example**: Grand Boston Hotel has 1,203 reviews from different guests
- **Real-world**: Multiple guests write reviews about their stay at the same hotel

HOTELS (1) ──→ (Many) AMENITIES

- **Meaning**: One hotel can offer many different amenities
- **Example**: Grand Boston Hotel offers WiFi, Parking, Pool, Gym, Restaurant, etc.
- **Real-world**: Hotels provide various facilities and services to guests

HOTELS (1) ──→ (One) POLICIES

- **Meaning**: One hotel has one set of policies (though stored as separate record)
- **Example**: Grand Boston Hotel has specific check-in/out times, pet rules, and cancellation terms
- **Real-world**: Each hotel has its own rules and operational guidelines

11. Summary

The HotelIQ Data Pipeline provides a robust, production-ready ETL system for hotel chatbot data. Key features include:

Technical Highlights

- **Batch UPSERT**: Efficient INSERT...ON CONFLICT for idempotent loads
- **Automatic Type Casting**: Converts CSV strings to proper database types
- **Connection Pooling**: Reuses database connections for performance
- **Foreign Key Safety**: Enforces load order to prevent constraint violations
- **Comprehensive Logging**: Tracks progress and errors at every step

Operational Benefits

- **Airflow Orchestration**: Automated scheduling and retry logic
- **Cloud SQL Integration**: Secure access via Cloud SQL Proxy
- **Reproducible**: Clear documentation and environment setup
- **Scalable**: Handles growing datasets with batch processing
- **Maintainable**: Modular code structure with single-responsibility functions

Data Handling

- **Transformation**: City filtering, aggregation, enrichment
- **Normalization**: Proper database schema with referential integrity
- **Error Handling**: Graceful failure recovery with detailed error messages