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

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/
├── data_pipeline/
│   ├── dags/
│   │   └── data_pipeline_airflow.py    # Airflow DAG definition
│   ├── src/
│   │   ├── extract.py                  # Data extraction functions
│   │   ├── utils.py                    # Util functions to extract hotel data
│   │   ├── transform.py                # Data transformation helpers
│   │   └── bucket_util.py              # GCS utilities (optional)
│   ├── sql/
│   │   ├── db_pool.py                  # Database connection pool
│   │   ├── load_to_database.py         # Main loader with batch upsert
│   │   ├── queries.py                  # SQL DDL and query helpers
│   │   └── test_connection.py          # Connection test script
│   ├── scripts/
│   │   └── run_proxy.sh                # Cloud SQL Proxy launcher
│   ├── intermediate/
│   │   └── csv/                        # CSV files directory
│   │       ├── hotels.csv
│   │       ├── rooms.csv
│   │       ├── reviews.csv
│   │       ├── amenities.csv
│   │       └── policies.csv
│   ├── docker-compose.yaml             # Airflow services configuration
│   ├── requirements.txt                # Python dependencies
│   ├── .env                            # Environment variables
│   └── README.md                       # Project documentation
```

3.1 Core Components Explanation

`dags/data_pipeline_airflow.py`

Purpose: Defines the Airflow DAG that orchestrates the entire ETL pipeline

Key Components:

```
# DAG tasks:
# 1. extract_task: Downloads raw hotel data
# 2. transform_task: Processes and enriches data
# 3. validate_task: Runs data quality checks
# 4. load_task: Loads CSVs into database
```

Workflow:

extract_task → transform_task → validate_task → load_task

Why Airflow:

- Automatic retry on failure
- Task dependency management
- Scheduling capabilities (weekly refreshes)
- Visual monitoring via web UI
- Distributed execution support

`sql/db_pool.py`

Purpose: Implements a singleton database connection pool

Key Class: `DatabasePool`

```
Features:
- Reads DB credentials from environment variables
- Creates connection pool on first use
- Reuses connections for efficiency
- Thread-safe connection management
```

Usage:

```
from sql.db_pool import db_pool

conn = db_pool.get_connection()
cursor = conn.cursor()
# Execute queries...
db_pool.release_connection(conn)
```

Why Connection Pooling:

- Reuses connections instead of creating new ones
- Improves performance (connection setup is expensive)
- Manages connection limits efficiently
- Automatic connection recovery on failure

`sql/load_to_database.py`

Purpose: Main loader module with batch upsert functionality

Key Functions:

1. `get_db_connection()`

```
def get_db_connection():
    """
    Returns a psycopg2 connection via db_pool

    Returns:
        psycopg2.connection: Active database connection
    """
    return db_pool.get_connection()
```

Why: Centralizes connection logic, making it easy to swap connection methods

2. `batch_upsert_csv_auto()`

Purpose: Performs intelligent batch UPSERT with automatic type casting

```
def batch_upsert_csv_auto(
    conn,
    table_name,
    csv_path,
    columns,
    conflict_columns,
    batch_size=1000
):
    """
    Batch UPSERT with automatic type detection and casting

    Process:
    1. Queries information_schema.columns to get column types
    2. Reads CSV into pandas DataFrame
    3. For each column, determines appropriate cast:
       - Numeric types: CAST(NULLIF(%s, '') AS INTEGER/NUMERIC)
       - Text types: NULLIF(%s, '')
       - Date/Timestamp: CAST(NULLIF(%s, '') AS DATE/TIMESTAMP)
    4. Builds INSERT ... ON CONFLICT ... DO UPDATE query
    5. Batches rows (default 1000) for performance
    6. Executes using psycopg2.extras.execute_batch
    7. Commits transaction

    Args:
        conn: Database connection
        table_name: Target table name
        csv_path: Path to CSV file
        columns: List of column names to insert
        conflict_columns: Columns for ON CONFLICT clause (usually primary keys)
        batch_size: Number of rows per batch

    Returns:
        int: Number of rows inserted/updated
    """
```

Why Automatic Casting:

- CSVs store everything as strings
- Database expects proper types (integer, numeric, date)
- Empty CSV cells become NULL (via `NULLIF(%s, '')`)
- Prevents type mismatch errors
- Handles schema changes gracefully

---

3. `batch_upsert_csv()`

Purpose: Simple batch UPSERT without automatic type detection

```
def batch_upsert_csv(
    conn,
    table_name,
    csv_path,
    columns,
    conflict_columns,
    batch_size=1000
):
```

When to Use:

- CSV preprocessing ensures correct types
- Performance optimization (avoids type detection query)
- Custom type handling needed

---

4. Table-Specific Importers

```
def import_hotels_from_local(csv_path):

    Imports hotels.csv into hotels table

    Columns:
    - hotel_id (PK)
    - name, address, city, state, country
    - star_rating, description
    - check_in_time, check_out_time
    - website_url, phone, email
```

```
def import_reviews_from_local(csv_path):
    """
    Imports reviews.csv into reviews table

    Columns:
    - review_id (PK)
    - hotel_id (FK → hotels)
    - user_id, rating, title, text
    - verified_purchase, helpful_votes
    - review_date
    """

def import_amenities_from_local(csv_path):
    """
    Imports amenities.csv into amenities table

    Columns:
    - amenity_id (PK)
    - hotel_id (FK → hotels)
    - category, name, description
    - is_free, availability, location
    """

def import_policies_from_local(csv_path):
    """
    Imports policies.csv into policies table

    Columns:
    - policy_id (PK)
    - hotel_id (FK → hotels)
    - policy_type, title, full_text
    - key_points, last_updated
    """
```

Why Separate Importers:

- Each table has different columns and constraints
- Allows customization per table
- Easier to debug specific table issues
- Maintains referential integrity (FK constraints)

5. `load_all_hotel_data_to_database()`

Purpose: Orchestrates the entire loading process

```
def load_all_hotel_data_to_database(csv_directory):
    """
    Main orchestrator function

    Process:
    1. Validates csv_directory exists
    2. Imports tables in FK-safe order:
       a. hotels (parent table, no FKs)
       b. rooms (FK → hotels)
       c. reviews (FK → hotels)
       d. amenities (FK → hotels)
       e. policies (FK → hotels)
    3. Logs progress and row counts
    4. Returns results dictionary

    Args:
        csv_directory: Path containing CSV files

    Returns:
        dict: Row counts per table or -1 on failure
        {
            'hotels': 5,
            'rooms': 47,
            'reviews': 1203,
            'amenities': 85,
            'policies': 25
        }
    """

    # Import order respects foreign key constraints
    results = {}

    try:
        results['hotels'] = import_hotels_from_local(
            os.path.join(csv_directory, 'hotels.csv')
        )
        results['rooms'] = import_rooms_from_local(
            os.path.join(csv_directory, 'rooms.csv')
        )
```

```
        results['reviews'] = import_reviews_from_local(
            os.path.join(csv_directory, 'reviews.csv')
        )
        results['amenities'] = import_amenities_from_local(
            os.path.join(csv_directory, 'amenities.csv')
        )
        results['policies'] = import_policies_from_local(
            os.path.join(csv_directory, 'policies.csv')
        )

        return results

    except Exception as e:
        logger.error(f"Failed to load data: {e}")
        return {table: -1 for table in ['hotels', 'rooms', 'reviews', 'amenities', 'policies']}
```

Why Specific Import Order:

- `hotels` first (no dependencies)
- `rooms`, `reviews`, `amenities`, `policies` reference `hotels.hotel_id`
- Prevents foreign key constraint violations
- Ensures data integrity

`src/transform.py`

Purpose: Data transformation and enrichment helpers

Key Functions:

1. `extract_reviews_based_on_city()`

```
def extract_reviews_based_on_city(city, output_dir):
    """
    Filters reviews for hotels in a specific city

    Process:
    1. Loads city-specific hotels CSV
    2. Extracts list of hotel_ids
    3. Filters main reviews CSV to keep only matching hotel_ids
    4. Writes city-specific reviews CSV

    Args:
        city: City name (e.g., "Boston")
```

```
    output_dir: Directory to save filtered CSV

  Returns:
    str: Path to city-specific reviews CSV
  """
```

Use Case: Filter reviews by geography for targeted analysis

---

2. `compute_aggregate_ratings()`

```
def compute_aggregate_ratings(city, output_dir):
  """
  Computes aggregate ratings per hotel

  Aggregations:
  - overall_rating: Average of all ratings
  - cleanliness_rating: Average cleanliness score
  - service_rating: Average service score
  - location_rating: Average location score
  - value_rating: Average value score
  - review_count: Total number of reviews

  Output:
    hotel_ratings_{City}.csv
  """
```

Use Case: Pre-compute ratings for faster chatbot responses

---

3. `enrich_hotels_perplexity()`

```
def enrich_hotels_perplexity(city, output_dir, delay_seconds=2, max_hotels=None):
  """
  Enriches hotel data with external API calls

  Process:
  1. Iterates through hotels
  2. Builds enrichment payload per hotel
  3. Makes API calls (with rate limiting via delay_seconds)
  4. Writes JSONL output (one JSON object per line per hotel)
```

Use Case:
- Add descriptions from external sources
- Enrich with POI data
- Fetch additional images

Why JSONL Format:

- Easy to append (one hotel = one line)
- Resumable (can continue after failure)
- Streaming-friendly for large datasets

---

4. `merge_sql_tables()`

```
def merge_sql_tables(city, output_dir):
    """
    Merges enrichment data with base hotel data

    Process:
    1. Loads hotels CSV
    2. Loads enrichment JSONL
    3. Loads ratings CSV (if exists)
    4. Joins data on hotel_id
    5. Normalizes into separate DataFrames:
       - hotels_df
       - rooms_df
       - amenities_df
       - policies_df
    6. Exports final CSVs ready for database load

    Returns:
        Paths to merged CSV files
    """
```

Why Normalization:

- Prevents data redundancy
- Optimizes database storage
- Maintains referential integrity
- Enables efficient queries

`sql/queries.py`

Purpose: SQL DDL and query helpers

Contains:

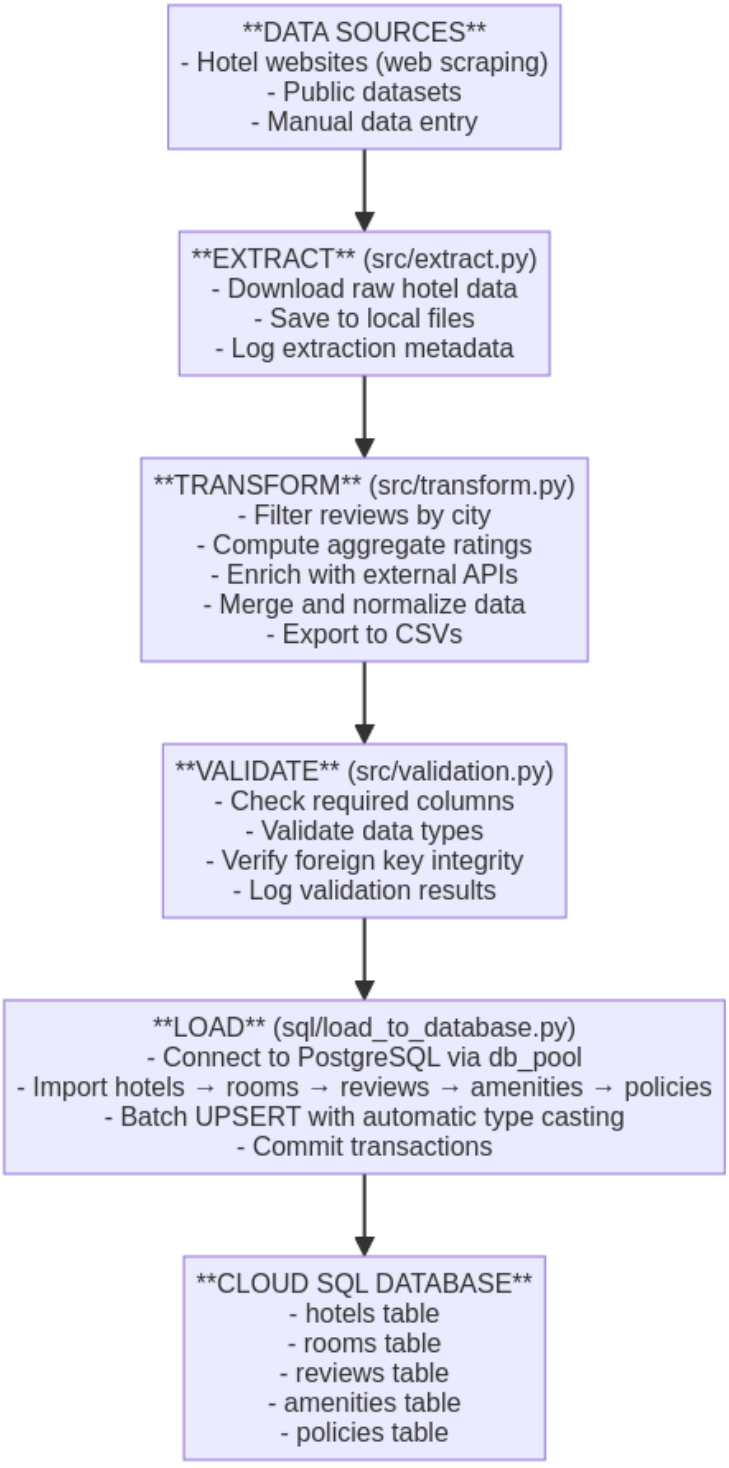
```
# Table creation DDL
CREATE_HOTELS_TABLE = """
CREATE TABLE IF NOT EXISTS hotels (
    hotel_id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT,
    city VARCHAR(100),
    state VARCHAR(100),
    country VARCHAR(100),
    star_rating INTEGER CHECK (star_rating BETWEEN 1 AND 5),
    description TEXT,
    check_in_time TIME,
    check_out_time TIME,
    website_url TEXT,
    phone VARCHAR(20),
    email VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
"""

# Similar DDL for rooms, reviews, amenities, policies tables

# Helper query snippets
INSERT_HOTEL_QUERY = "..."
UPDATE_HOTEL_QUERY = "..."
```

Use Case:

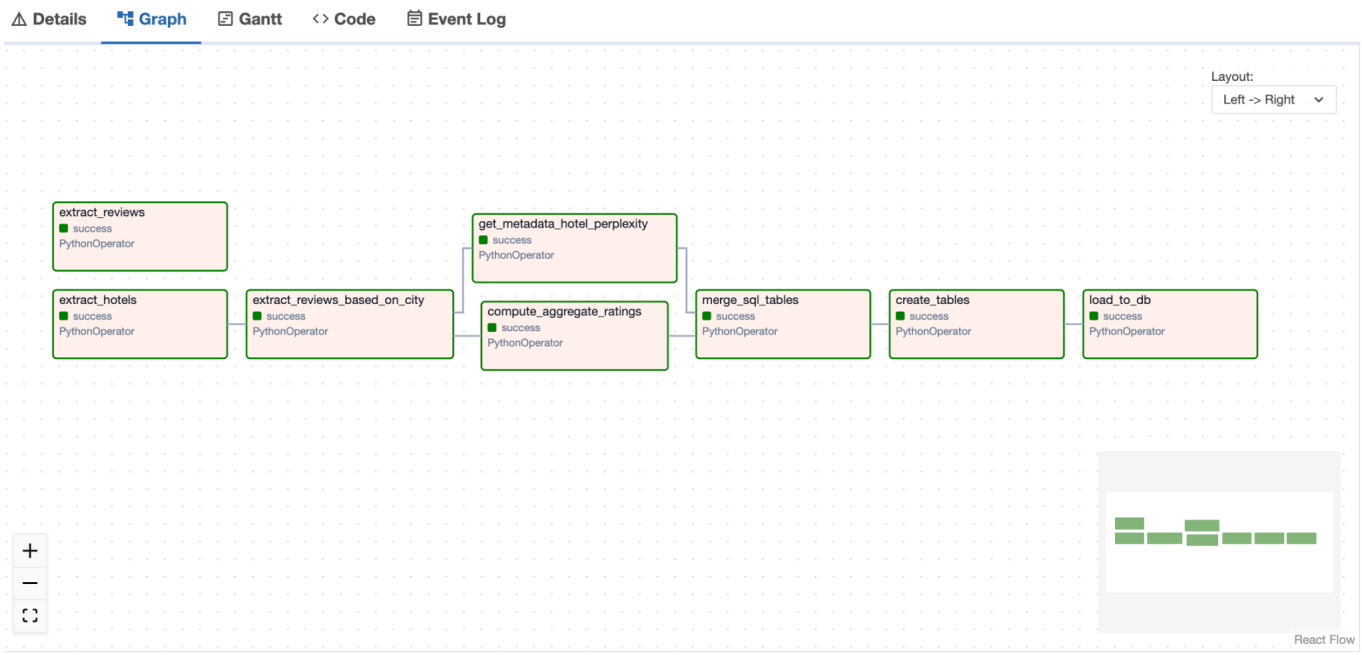
- Reference for schema expectations
- Initialize database schema
- Understand primary keys and constraints

4. Data Flow and Architecture

4.1 End-to-End Pipeline Flow



4.2 Airflow DAG Visualization



5. Explanation of Each Task

Task 1: `extract_reviews`

Type: PythonOperator

Purpose: Extracts raw review data from source files or APIs and saves them locally.

Process:

1. Downloads or reads review data from configured source
2. May include reviews from multiple hotels
3. Saves raw review data to intermediate storage
4. Logs extraction metadata (timestamp, record count)

Output:

- Raw reviews file (JSON or CSV format)
- Stored in intermediate directory for downstream processing

Why This Task:

- Reviews are critical for the chatbot to answer questions about guest experiences
- Provides the foundation for aggregate ratings calculation
- Enables sentiment analysis and quality insights

Task 2: `extract_hotels`

Type: PythonOperator

Purpose: Extracts hotel metadata including basic information like name, address, amenities, and policies.

Process:

1. Scrapes or retrieves hotel information from source websites
2. Collects structured data: hotel names, addresses, descriptions
3. Gathers unstructured content: amenities, policies, FAQs
4. Saves raw hotel data locally

Output:

- Raw hotels data file
- Contains hotel identifiers used by downstream tasks

Why This Task:

- Hotels data is the foundation - all other data references hotel_id
- Provides core information for chatbot responses
- Must complete before dependent tasks can proceed

Task 3: `extract_reviews_based_on_city`

Type: PythonOperator
Depends On: `extract_hotels`

Purpose: Filters the raw reviews dataset to include only reviews for hotels in a specific city (e.g., Boston).

Process:

1. Loads the hotels dataset extracted in the previous task
2. Extracts list of `hotel_id` values for the target city
3. Filters the main reviews dataset to keep only matching hotel_ids
4. Writes city-specific reviews file

Output:

- `reviews_{City}.csv` - City-specific filtered reviews

Why This Task:

- Reduces data volume for downstream processing
- Enables city-specific analysis and chatbot responses
- Maintains data relevance for targeted deployments

Task 4: `get_metadata_hotel_perplexity`

Type: PythonOperator
Depends On: `extract_reviews_based_on_city`

Purpose: Enriches hotel data with additional metadata by making API calls to external enrichment services (e.g., Perplexity AI or similar).

Process:

1. Iterates through each hotel in the city dataset
2. Builds enrichment payload with hotel details
3. Makes API calls to fetch:
    - Enhanced descriptions
    - Points of interest nearby
    - Additional amenity details
    - Contextual information
4. Writes enrichment data as JSONL (one JSON object per line per hotel)
5. Implements rate limiting to respect API quotas

Output:

- `enrichment_{City}.jsonl` - Enhanced hotel metadata

Why JSONL Format:

- Easy to append (resumable if interrupted)
- One hotel = one line (easy to parse)
- Streaming-friendly for large datasets

Why This Task:

- Adds rich context that may not be on hotel websites
- Improves chatbot response quality with comprehensive information
- Enables competitive intelligence and market insights

Task 5: `compute_aggregate_ratings`

Type: PythonOperator
Depends On: `get_metadata_hotel_perplexity`

Purpose: Computes aggregate rating statistics per hotel from review data.

Process:

1. Loads city-specific reviews file
2. Groups reviews by `hotel_id`
3. Calculates aggregate metrics:
    - **Overall Rating**: Average of all ratings
    - **Cleanliness Rating**: Average cleanliness scores
    - **Service Rating**: Average service scores
    - **Location Rating**: Average location scores
    - **Value Rating**: Average value-for-money scores
    - **Review Count**: Total number of reviews per hotel
4. Writes aggregate ratings CSV

Why This Task:

- Pre-computed ratings enable fast chatbot responses
- No need to calculate averages on-the-fly during user queries
- Provides quick insights for hotel comparisons

Task 6: `merge_sql_tables`

Type: PythonOperator
Depends On: `compute_aggregate_ratings`

Purpose: Merges enrichment data, ratings, and base hotel data into normalized tables ready for database loading.

Process:

1. Loads three data sources:
    - Base hotels CSV
    - Enrichment JSONL (from Perplexity API)
    - Ratings CSV (from aggregate computation)
2. Joins data on `hotel_id`
3. Normalizes into separate DataFrames following relational schema:
    - **hotels_df**: Core hotel information
    - **rooms_df**: Room types and details (extracted from enrichment)
    - **amenities_df**: Hotel facilities (extracted and normalized)
    - **policies_df**: Hotel policies (extracted and structured)
    - **reviews_df**: Individual reviews (with ratings merged)
4. Exports final CSVs to `intermediate/csv/` for database loading

Output:

- `hotels.csv` - Normalized hotel records
- `rooms.csv` - Room type details
- `amenities.csv` - Hotel amenities
- `policies.csv` - Hotel policies
- `reviews.csv` - Review records (copied from city reviews)

Why Normalization:

- Eliminates data redundancy
- Maintains referential integrity (foreign keys)
- Optimizes database storage and query performance
- Follows relational database best practices

Why This Task:

- Transforms denormalized enrichment data into proper database schema
- Ensures data consistency across tables
- Prepares data in exact format expected by database loader

Task 7: `create_tables`

Type: PythonOperator
Depends On: `merge_sql_tables`

Purpose: Creates database tables with proper schemas if they don't already exist.

Process:

1. Connects to PostgreSQL database via `db_pool`
2. Executes CREATE TABLE IF NOT EXISTS statements for:
    - `hotels` table (with PRIMARY KEY on `hotel_id`)
    - `rooms` table (with FOREIGN KEY to `hotels`)
    - `reviews` table (with FOREIGN KEY to `hotels`)
    - `amenities` table (with FOREIGN KEY to `hotels`)
    - `policies` table (with FOREIGN KEY to `hotels`)
3. Creates necessary indexes for query optimization
4. Commits schema changes

Why This Task:

- Ensures database schema exists before data loading
- Idempotent (safe to run multiple times with IF NOT EXISTS)
- Defines constraints (primary keys, foreign keys) for data integrity
- Sets up indexes for query performance

Task 8: `load_to_db`

Type: PythonOperator
Depends On: `create_tables`

Purpose: Loads the processed CSV files into the PostgreSQL database using batch UPSERT operations.

Process:

1. Calls `load_all_hotel_data_to_database('intermediate/csv')`
2. Loads tables in foreign key-safe order:
    - **Step 1**: `hotels` (no dependencies)
    - **Step 2**: `rooms` (depends on `hotels`)
    - **Step 3**: `reviews` (depends on `hotels`)
    - **Step 4**: `amenities` (depends on `hotels`)
    - **Step 5**: `policies` (depends on `hotels`)

3. Uses `batch_upsert_csv_auto()` for each table:
    - Reads CSV into pandas DataFrame
    - Detects column types from database schema
    - Applies automatic type casting (empty strings → NULL)
    - Batches rows (1000 per batch) for performance
    - Executes INSERT ... ON CONFLICT ... DO UPDATE (UPSERT)
    - Commits transaction
4. Returns row counts per table

Why UPSERT (not INSERT):

- **Idempotent**: Running multiple times doesn't create duplicates
- **Updates existing records**: If hotel data changes, it updates rather than fails
- **Handles data refreshes**: Weekly runs update changed data gracefully

Why This Task:

- Final step: Makes processed data available to chatbot
- Ensures data integrity through foreign key constraints
- Optimized for performance with batch operations
- Provides detailed logging for monitoring

---

6. Workflow Summary

Data Transformation Journey

Input: Raw hotel websites, review sources
Output: Normalized PostgreSQL database ready for chatbot queries

Transformations:

1. **Raw** → **Structured**: HTML/JSON → CSV
2. **Global** → **City-Specific**: All reviews → Boston reviews
3. **Raw** → **Enriched**: Basic hotel data → Enhanced with API data
4. **Individual** → **Aggregated**: 1000+ reviews → Per-hotel averages
5. **Denormalized** → **Normalized**: Flat files → Relational tables
6. **Files** → **Database**: CSVs → PostgreSQL tables

7. Reproducibility and Data Versioning

CSV File Requirements

Each CSV must include a header row with column names matching the definitions in `load_to_database.py`:

hotels.csv:

hotel_id,name,address,city,state,country,star_rating,description,check_in_time,check_out_time,
website_url,phone,email
hotel_001,Grand Hotel,123 Main St,Boston,MA,USA,4,Luxury
hotel...,15:00,11:00,https://...,+1234567890,info@hotel.com

rooms.csv:

room_id,hotel_id,room_type,description,amenities,bed_type,max_occupancy,size_sqft,base_pri
ce
room_001,hotel_001,Deluxe King,Spacious room...,WiFi;TV;Mini-bar,King,2,400,199.99

Missing Values:

- Empty cells should be empty strings
- Loader converts empty strings to NULL for typed columns
- Example: `,,` for missing middle column

Reproducibility Steps

1. Prepare Environment

```
 cd hotel-iq/data_pipeline
pip install -r requirements.txt
```

2. Configure Database

```
 # Edit .env with your DB credentials
nano .env
```

3. Start Cloud SQL Proxy (if needed)

```
 bash scripts/run_proxy.sh
```

4. Verify CSV Files

```
 ls -lh intermediate/csv/
# Should show: hotels.csv, rooms.csv, reviews.csv, amenities.csv, policies.csv
```

5. Run Pipeline

```
 # Option A: Via Airflow
docker-compose up -d
# Trigger DAG in UI at http://localhost:8080

# Option B: Direct execution
python -c "from sql.load_to_database import load_all_hotel_data_to_database;
print(load_all_hotel_data_to_database('intermediate/csv'))"
```

6. Verify Results

```
 python sql/test_connection.py
```

8. Error Handling and Logging

Error Handling Strategies

Database Connection Errors

```
try:
    conn = db_pool.get_connection()
except psycopg2.OperationalError as e:
    logger.error(f"Database connection failed: {e}")
    # Retry logic with exponential backoff
    time.sleep(5)
    conn = db_pool.get_connection()  # Retry
```

CSV File Errors

```
try:
    df = pd.read_csv(csv_path)
except FileNotFoundError:
    logger.error(f"CSV file not found: {csv_path}")
    return -1
except pd.errors.EmptyDataError:
    logger.warning(f"Empty CSV file: {csv_path}")
    return 0
```

Type Casting Errors

```
try:
    cursor.execute(query, batch)
except psycopg2.DataError as e:
    logger.error(f"Type mismatch in batch: {e}")
    # Log problematic rows
    for row in batch:
        try:
            cursor.execute(query, [row])
        except psycopg2.DataError as row_error:
            logger.error(f"Bad row: {row}, Error: {row_error}")
```

Foreign Key Violations

```
try:
    import_rooms_from_local(rooms_path)
except psycopg2.IntegrityError as e:
    if 'foreign key constraint' in str(e):
        logger.error("Foreign key violation: hotel_id in rooms doesn't exist in hotels table")
        # Validate which hotel_ids are missing
```

Logging Configuration

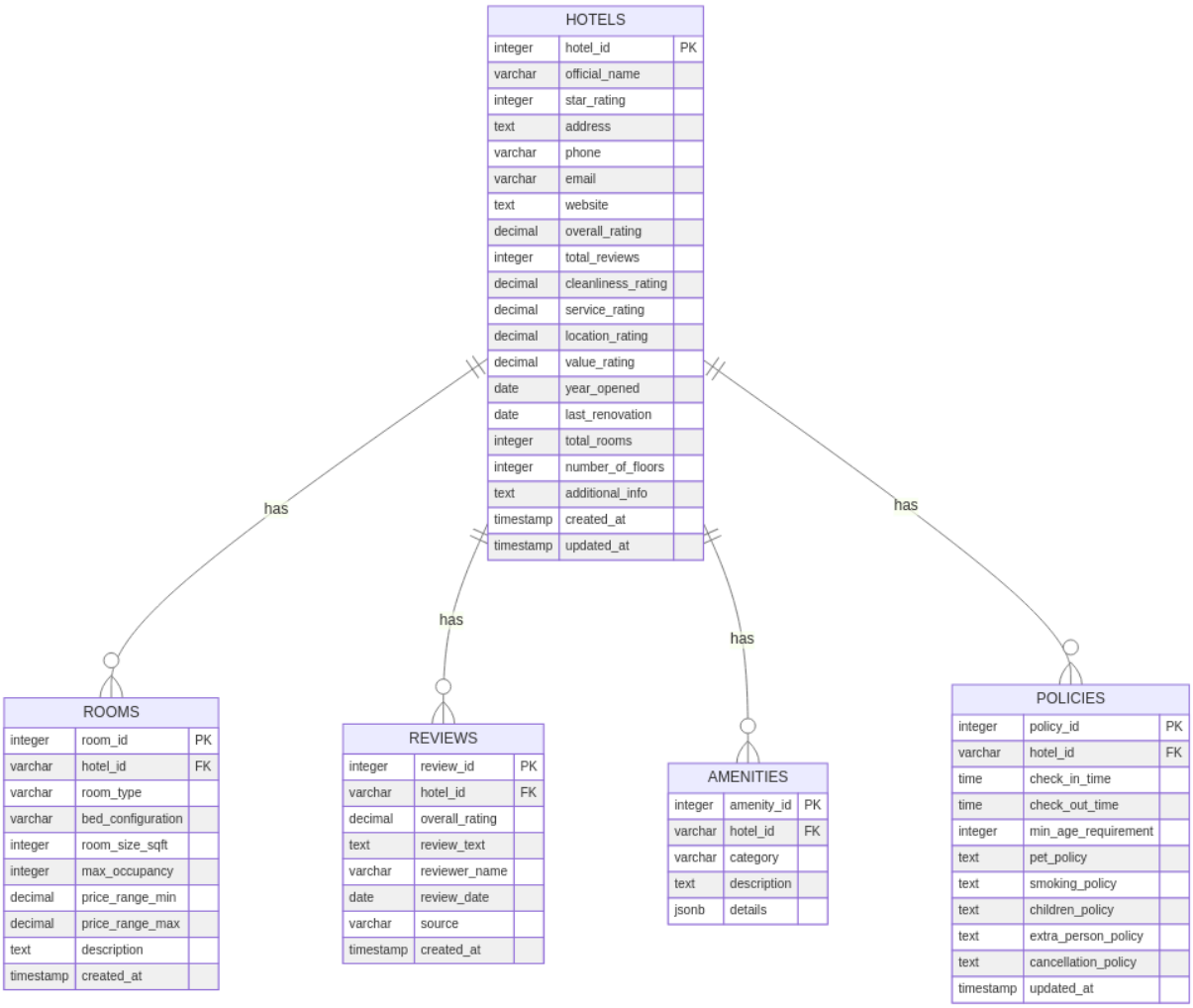
```
import logging

# Configure logger
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('logs/pipeline.log'),
        logging.StreamHandler()  # Also print to console
    ]
)

logger = logging.getLogger(__name__)

# Log examples from loader
logger.info(f"Starting import of {table_name}...")
logger.info(f"Processing batch {batch_num}/{total_batches}")
```

logger.info(f"Successfully loaded {row_count} rows into {table_name}")
logger.warning(f"Empty value in column {col_name}, converting to NULL")
logger.error(f"Failed to load batch: {error}", exc_info=True)

Log Locations

- **Pipeline logs**: `logs/pipeline.log`
- **Airflow logs**: `logs/airflow/scheduler/`, `logs/airflow/dag_processor_manager/`
- **Docker logs**: `docker-compose logs -f`

9. ER Diagram



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