# End-to-End Solution for Exchange Rate Pipeline

## Overview

This solution is designed to automate the retrieval, processing, and storage of exchange rate data from the Open Exchange Rates API. It ensures that the latest and historical exchange rates are efficiently stored and made available for analysis, while also incorporating mechanisms for scalability and resilience.

## Key Components & Workflow

### 1. Data Extraction

The solution uses aiohttp to asynchronously fetch exchange rate data, allowing multiple requests to be handled concurrently for improved performance.

Supports fetching both:

- Historical Data (when --start-date and --end-date are provided).

- Latest Data (when no date range is given, defaulting to the last 30 days if not specified).

Implements rate limit handling via exponential backoff, ensuring API requests do not get throttled and that errors are retried appropriately.

### 2. Data Processing

Parses API responses to extract relevant exchange rate values for each currency pair.

Generates unique row IDs using a hashing function (SHA-256) to ensure deduplication and consistency in storage.

Converts timestamps from the API into readable date formats.

Enriches the data with the following metadata:

- Retrieval time: Timestamp when the data was fetched.

- API version: Tracks the API version being used.

- Day of the week: Useful for trend analysis.

- Weekend flag: Indicates whether the date falls on a weekend.

### 3. Data Storage & Partitioning

Saves the exchange rate data in Apache Parquet format, which is optimized for fast querying and low storage footprint.

Uses a partitioning strategy based on:

- Year (year column)

- Month (month column)

Ensures automatic output path creation to prevent issues with missing directories.

If the dataset grows significantly, partitioning by day/hour can further improve query efficiency.

### 4. Testing & Validation

Includes unit tests implemented with pytest and pytest-asyncio to verify core functionalities:

- ID generation: Ensures that hash-based IDs are unique and deterministic.

- Data fetching: Mocks API responses to ensure proper extraction of exchange rates.

- File creation: Validates that Parquet files are correctly written and stored in the expected partitions.

### 5. Execution & Automation

Runs via a Command Line Interface (CLI) allowing users to specify parameters like date range and output location.

Can be scheduled using cron jobs, Airflow, or deployed as a serverless function (AWS Lambda, Google Cloud Functions) for periodic updates.

Supports incremental fetching where only new exchange rate data is retrieved instead of fetching redundant records.

## Potential Improvements & Enhancements

### 1. Hourly Data Storage for More Granular Insights

The script currently stores exchange rate data at the daily level.

The Open Exchange Rates API updates hourly, but the solution does not take advantage of this granularity.

Enhancements:

- Modify the schema to include hour metadata.

- Partition data by year/month/day/hour for finer-grained storage and efficient querying.

- Implement an append-only storage mechanism instead of overwriting data.

- Introduce data validation checks to handle missing or inconsistent hourly records.

### 2. Data Deduplication & Upserts

If the script fetches data multiple times a day, it might store duplicate records.

Enhancements:

- Implement a deduplication step before writing data.

- Introduce upserts (update existing records instead of duplicating them) to maintain a clean dataset.

### 3. Automated Scheduling & Incremental Fetching

Running the script manually is inefficient; automating execution ensures up-to-date data availability.

Enhancements:

- Schedule the script to run hourly using cron jobs or as a cloud function.

- Modify the logic to fetch only new data since the last recorded timestamp.

- Implement logging & error handling for better debugging and tracking of execution history.

### 4. Database Integration for Better Querying

While Parquet is optimized for batch analytics, it does not allow real-time querying or transactional updates.

Enhancements:

- Store exchange rate data in a time-series database such as InfluxDB, TimescaleDB, or ClickHouse.

- Enable querying of exchange rates for specific time ranges and currency pairs.

- Provide an API endpoint (e.g., using FastAPI) that allows real-time retrieval of exchange rates.

### 5. Real-Time Alerts & Monitoring

There is no system in place to detect abnormal exchange rate fluctuations.

Enhancements:

- Implement alerts via email, Slack, or webhooks if a currency pair’s exchange rate fluctuates beyond a set threshold.

- Use Grafana or Prometheus to visualize exchange rate trends and monitor API request success rates.

## Final Thoughts & Next Steps

The current pipeline is a solid foundation for fetching, processing, and storing exchange rate data. However, by implementing hourly updates, deduplication, automation, and real-time querying, the system can be significantly improved for scalability and usability.

Priority Roadmap:

1. Implement hourly data storage and append functionality.

2. Enhance deduplication and upsert logic.

3. Automate execution and enable incremental fetching.

4. Introduce a time-series database for faster querying.

5. Develop real-time alerting and monitoring features.

By iterating on these improvements, the pipeline can evolve into a real-time financial data ingestion and analytics system that provides high-value insights to users.