Technical Report: A Serverless ETL Pipeline for Flight Price Prediction on AWS

Tyler Venner

September 9, 2025

Abstract

This report outlines the design of a scalable, automated, and serverless Extract, Transform, and Load (ETL) pipeline on Amazon Web Services (AWS) for collecting flight price data. The architecture leverages AWS Lambda for event-driven compute, Amazon S3 for building a durable data lake, and Amazon EventBridge for cron-based scheduling. A key challenge in cloud-based data collection, reliable access to dynamic web content, is addressed by integrating a third-party Browser API service. This service manages residential proxy rotation and browser fingerprinting to ensure consistent session initialization. The extraction logic, containerized using Docker and deployed to Lambda, controls a remote browser to acquire authentication tokens from a major travel aggregator before making direct API calls for bulk data retrieval. The transformation phase utilizes AWS Lambda and the pandas library to parse, clean, and enrich the raw data with competitive, temporal, and macroeconomic features. Finally, the transformed data is loaded back into the S3 data lake in a partitioned Parquet format, optimized for cost-effective querying and analysis with Amazon Athena. This serverless design provides a robust, low-maintenance, and highly scalable solution for building a foundational dataset for machine learning-based price prediction.

Contents

1 Introduction

The objective of this project is to architect a fully automated, scalable, and cost-effective data pipeline in a cloud environment for the purpose of collecting flight price data. This report details the design of a serverless ETL (Extract, Transform, Load) system built on Amazon Web Services (AWS), intended to create a comprehensive dataset for training machine learning models to predict future flight prices.

By leveraging a serverless architecture, the pipeline minimizes operational overhead and infrastructure management, allowing for a focus on data processing logic. The core of this design is the use of managed AWS services, which provide robust, event-driven capabilities that are ideal for a scheduled data collection task. The primary data source is a major travel aggregator, a dynamic website requiring robust session handling and JavaScript execution.

This document provides a blueprint for the cloud-native pipeline, organized by its core components:

- **System Architecture:** An overview of the AWS services used and how they interact to form a cohesive data flow.
- **Extraction:** The methodology for running a web scraper within an AWS Lambda function, utilizing a remote browser strategy to handle dynamic content rendering and session management.
- **Transformation:** The process of cleaning the raw JSON data and performing extensive feature engineering within a serverless compute environment.
- **Loading:** The strategy for storing the final, processed data in an Amazon S3 data lake using a partitioned Parquet format, optimized for serverless querying.

2 System Architecture

The pipeline is designed as a fully serverless application on AWS, meaning there are no servers to provision or manage. This architecture is event-driven, cost-effective, and highly scalable. It integrates several key AWS services with an external proxy management service to create a robust and automated data collection workflow.

2.1 Core AWS Services

The architecture is built upon the following managed services:

- **Amazon EventBridge:** Acts as the cron-based scheduler for the entire pipeline. An EventBridge rule is configured to trigger the main processing function on a fixed schedule (e.g., daily).
- **Amazon ECR (Elastic Container Registry):** A managed Docker container registry used to store, manage, and deploy the Python application's container image.
- **AWS Lambda:** The core of the pipeline's compute logic. The function is configured to run from a container image stored in ECR. It executes the Python script that performs the extraction and transformation tasks.
- **Amazon S3 (Simple Storage Service):** Serves as the durable, scalable data lake for the project. It stores the final, processed data in an optimized, partitioned format.
- **Amazon Athena:** A serverless, interactive query service that allows for direct SQL-based analysis of the data stored in S3. It provides the primary interface for accessing the data for analysis and machine learning.

2.2 Data Flow

The data flows through the system in a logical, event-driven sequence:

1. **Trigger:** An Amazon EventBridge rule fires on its schedule (e.g., daily at 12:00 UTC).
2. **Invocation:** The rule invokes the AWS Lambda function. AWS Lambda automatically pulls the latest container image from Amazon ECR and starts the execution environment.
3. **Extraction:** The Python script inside the Lambda function calls out to external sources:

- It connects to the third-party Browser API, Bright Data, which provides a remote headless browser running on a residential IP. The script uses Selenium to control this browser and extract the necessary session tokens from the target website.
- It makes direct API calls to macroeconomic data providers (EIA, FRED).

4. **Transformation:** With the raw data collected, the Lambda function uses the `pandas` library in-memory to parse the JSON, clean the data, and engineer the full set of predictive features.
5. **Loading:** The function converts the final, transformed DataFrame into the Parquet file format and uploads it to a designated Amazon S3 bucket. The data is stored using a Hive-style partitioning scheme (e.g., `/scrape_date=YYYY-MM-DD/`) to optimize query performance.
6. **Querying:** Once the data is in S3, Amazon Athena can be used to run standard SQL queries against the dataset, enabling data exploration, analysis, and preparation for machine learning.

2.3 Visual Representation

The serverless architecture and data flow are illustrated in Figure 1.

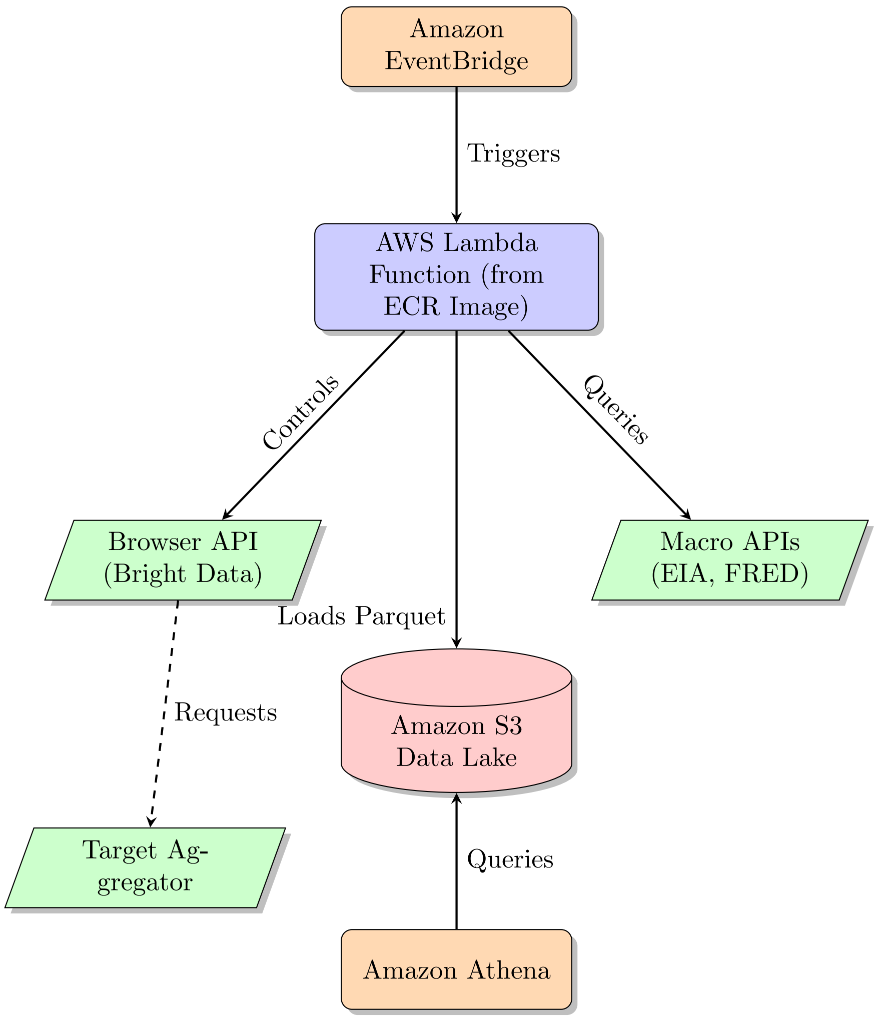

Figure 1: Serverless ETL Architecture on AWS.

3 Extraction

The Extraction phase in a cloud environment presents a unique set of challenges compared to a local implementation, primarily centered around handling dynamic web applications. This section details the methodology for successfully extracting data from modern websites and external APIs within the AWS Lambda environment.

3.1 Containerized Application

The entire Python application is packaged into a Docker container image. This image is then pushed to Amazon ECR, where it is stored and versioned, ready for deployment to Lambda.

3.2 Primary Data Extraction: Remote Browser Session Management

Retrieving data from modern, dynamic web applications requires a browser environment to correctly render JavaScript and manage session state. Cloud environments, however, lack native display capabilities. To address this, the pipeline integrates with a third-party Browser API, Bright Data. This service provides programmatic access to a remote Selenium-compatible browser, ensuring consistent behavior across execution environments.

The extraction process utilizes a two-step hybrid approach:

1. **Acquiring Session Credentials:** The Lambda function uses the Selenium library to connect to the remote Browser API via a WebSocket URL. This URL, stored securely as an environment variable, contains the authentication credentials. The function then executes the logic to navigate to the target search page, wait for the JavaScript to render, and extract the critical session tokens and cookies. This ensures the session is established authentically, mirroring a standard user interaction.

2. **Polling the Internal API:** With the valid credentials acquired, the Lambda function makes a direct API call to the provider's internal endpoint. This call is a standard HTTPS request that relies on the valid session ID and cookies for authentication. This allows for the efficient bulk download of flight data in a lightweight JSON format without the overhead of rendering graphical elements for every request.

This strategy delegates the complex task of session management to a specialized service while keeping the core application logic clean and efficient.

```python
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Get the remote browser connection URL from an environment variable
remote_webdriver_url = os.environ.get( BRIGHTDATA_REMOTE_URL )

chrome_options = Options()
# Add capability to ensure standard browser behavior
chrome_options.set_capability('brd:block_robots', False)

# Connect to the remote browser instead of creating a local one
driver = webdriver.Remote(
    command_executor=remote_webdriver_url,
    options=chrome_options
)

# Navigate to the target aggregator to initialize session
driver.get( https://www.example-travel-site.com/flights/... )
# ...
```

Listing 1: Connecting to a remote browser with Selenium

3.3 Secondary Data Source Extraction

The extraction of macroeconomic data remains identical to the local implementation. The Lambda function makes direct, authenticated API calls to ExchangeRate-API, the EIA, and FRED to retrieve the latest data points. The credentials for these services are stored as secure environment variables in the Lambda function's configuration.

4 Transformation

Once the raw data is extracted, the Transformation phase begins. This critical process is executed entirely in-memory within the AWS Lambda function's execution environment. It converts the raw JSON into a structured, feature-rich format, preparing it for loading into the data lake. This phase is handled by the `parser.py` and `transformer.py` modules.

4.1 Parsing and Initial Structuring

The first step is to parse the complex, nested JSON object returned by the source's internal API. The `parser.py` module is responsible for navigating this structure to extract the essential data points for each flight, such as price, airline, flight segments, and durations. This process transforms the deeply nested object into a flat list of Python dictionaries, where each dictionary represents a single flight observation. The macroeconomic data fetched during the extraction phase is also integrated into each record at this stage.

4.2 Feature Engineering with Pandas

With the data parsed and structured, the `transformer.py` module performs extensive feature engineering. The list of dictionaries is loaded into a DataFrame to facilitate efficient, vectorized calculations. The following categories of features are engineered:

4.2.1 Competitive Analysis Features

To contextualize each flight's price relative to its market, the data is grouped by its route and the date of the scrape. Several competitive metrics are then calculated:

- **Price Rank:** The ordinal rank of the flight's price on its route for that day, from cheapest (1) to most expensive.
- **Price Percentile:** The normalized rank, indicating the percentage of other flights that are cheaper.

  $$P_p = \frac{\text{rank} - 1}{N - 1}$$

  where N is the total number of flights available on the route.
- **Price Delta from Cheapest:** The absolute price difference from the route's minimum price.

  $$\Delta_p = p_i - p_{\text{min}}$$

4.2.2 Time-Based and Holiday Features

To capture temporal patterns, a variety of time-based features are generated from the flight's departure date and the scrape date:

- **Days Until Departure:** The number of days in the booking window.
- **Departure Day of Week:** The day of the week (e.g., Monday, Tuesday).
- **Booking Window Category:** A categorical feature grouping the booking window into intuitive segments like "Last Week," "Short Term," "Mid Term," etc.
- **Holiday Proximity:** Using the Python `holidays` library, the number of days until the next major public holiday (in either the US or Japan) is calculated, providing a proxy for travel demand.

4.2.3 Calculated Flight Metrics

To normalize for different trip lengths and provide measures of value, the following metrics are computed:

- **Layover to Flight Time Ratio:** A measure of travel efficiency.

  $$R_{\text{layover}} = \frac{T_{\text{total\_layover}}}{T_{\text{total\_flight}}}$$

- **Price Per Minute in Air:** A normalized cost metric.

  $$C_{\text{minute}} = \frac{\text{Price}}{T_{\text{total\_flight}}}$$

4.2.4 Example Transformed Record

After the parsing and feature engineering steps are complete, the final output is a list of structured records. Each record is a flat Python dictionary containing all the core, engineered, and external data features. A sample of a single, fully transformed flight observation is shown below.

```
{
    flight_id :  374 d7d32da51d15b767df69c64e55aef ,
    route_name :   TYO_to_USA ,
    scrape_timestamp :  2025-09-04 20:38:18.438141 ,
    price : 542.5,
    provider_name :   Mytrip ,
    num_providers_offering : null,
    marketing_airline :   ZIPAIR ,
    number_of_stops : 0,
    total_duration_minutes : 590,
    total_layover_minutes : 0,
    total_flight_time : 842,
    layover_to_flight_time_ratio : 0.0,
    price_per_minute_in_air : 0.4864406779661017,
    origin_airport :   NRT ,
    destination_airport :   SFO ,
    departure_datetime_local :  2025-09-09 21:30:00 ,
    final_arrival_datetime_local :  2025-09-09 15:20:00 ,
    co2_total : 1.119907021522522,
    co2_per_minute : 0.0018981474941059694,
    is_overnight_flight : true,
    has_entire_journey_wifi : true,
    total_entertainment_segments : 1,
    price_rank_on_route : 1,
    price_percentile : 7.2,
    price_delta_from_cheapest : 123.1,
    days_until_departure : 5,
    departure_day_of_week : 1,
    departure_period :   Evening ,
    is_weekend_departure : false,
    is_peak_departure_day : false,
    booking_window_category :   Last Week ,
    days_until_next_holiday : 6,
    next_holiday_name :   \u656c\u8001\u306e\u65e5 ,
    usd_to_jpy_rate : 148.4087,
    jet_fuel_price_usd_gal : 2.12,
    us_cpi : 322.132
}
```

Listing 2: Sample of a Final Transformed Flight Record

5 Loading

The final phase of the cloud-native pipeline is Loading, where the transformed, feature-rich data is persisted into a durable and query-optimized data store. In this serverless architecture, a traditional database is replaced with an Amazon S3-based data lake.

5.1 Storage Format: Apache Parquet

Instead of loading data as plain text (like CSV) or JSON, the pipeline uses the Apache Parquet format. Parquet is a columnar storage file format that is highly optimized for analytical querying. Its key advantages include:

- **Columnar Storage:** Data is stored by column rather than by row. This allows analytical queries that only need a subset of columns (e.g., 'SELECT price, departure_date') to read only the necessary data, dramatically reducing the amount of data scanned and improving performance.
- **High Compression:** Parquet uses efficient compression algorithms, which significantly reduce the storage footprint in S3, leading to lower costs.
- **Schema Evolution:** The format is designed to handle changes in schema over time, such as the addition of new columns.

The Python pyarrow library is used within the Lambda function to convert the final pandas DataFrame into the Parquet format in memory before uploading.

5.2 Data Partitioning

To further optimize query performance and manage costs, the data is written to S3 using a Hive-style partitioning scheme. The data is partitioned by the scrape date, creating a directory structure that includes the partition key and value.

```
s3://bucket-name/processed-ml/scrape_date=2025-09-05/data.parquet
s3://bucket-name/processed-ml/scrape_date=2025-09-06/data.parquet
```

Listing 3: Example S3 Partitioned Path

This partitioning acts as a virtual index. When querying the data with Amazon Athena, a filter on the partition key (e.g., ‘WHERE scrape_date = ’2025-09-05’‘) will instruct Athena to only scan the data within that specific directory, dramatically reducing the amount of data processed and, consequently, the cost of the query.

5.3 Querying with Amazon Athena

With the data loaded into the S3 data lake, it becomes immediately available for querying via Amazon Athena. A one-time setup is required to create an external table that maps to the data's schema and location in S3.

```sql
CREATE EXTERNAL TABLE flight_data (
    flight_id STRING,
    price DOUBLE,
    -- ... all other columns with their data types
    price_percentile DOUBLE
)
PARTITIONED BY (scrape_date STRING)
STORED AS PARQUET
LOCATION 's3://bucket-name/processed-ml/';
```

Listing 4: Example Athena CREATE EXTERNAL TABLE Statement

After the table is created, running ‘MSCK REPAIR TABLE flight_data;‘ will make Athena discover all the existing date partitions. From that point on, standard SQL can be used to analyze the entire historical dataset directly from S3.