A Dual-Model System for Flight Price Forecasting: From Data Ingestion to MLOps

Tyler Venner

October, 2025

Abstract

This report details the design and implementation of a modular machine learning system for predicting flight prices, building upon a previously established ETL pipeline. The core thesis is that a single model is insufficient for providing actionable, user-facing insights. We present a dual-model architecture: (1) a *market-level regression model* to forecast overall route prices using pre-aggregated data, and (2) a *flight-level classification model* to predict individual price movements for a "Buy/Wait" recommendation. We introduce a high-performance data loading strategy that pushes feature and target engineering (using SQL `GROUP BY` and `LEAD` window functions) directly into the database. Furthermore, we describe a robust MLOps pipeline where model artifacts are saved with complete feature lists and categorical mappings, ensuring resilient and error-free inference.

Contents

1 Introduction

1.1 Project Objective

The primary business problem this project addresses is the financial challenge consumers face due to the high volatility and opacity of airline ticket pricing. The cost of a specific flight ticket, p, is a complex function of time-to-departure, market demand, competitor pricing, and exogenous factors (e.g., fuel costs). For a consumer, the goal is to minimize their cost function, $C = p_{ticket}$, over a given purchasing window T. The objective of this project is to move beyond simple data collection and develop a system that provides actionable, data-driven intelligence to help a user achieve $\min(C)$. This requires a system capable of accurately forecasting future price movements.

1.2 The Dual-Model Thesis

A simple, single price prediction, p_{t+n}, is a useful metric but is ultimately insufficient for providing a complete and actionable recommendation. A single forecast answers what a price might be, but not which flight to buy or when to buy it. To provide a truly valuable tool, a more nuanced system is required.

This project proposes a *dual-model thesis*: a complete forecasting system must address two distinct, complementary questions. First, a *market-level regression model* is needed for high-level planning. This model forecasts an aggregated market metric, such as the minimum expected price (p_{min}) for a given route R and departure date D. This answers the user's high-level question: "What is a good price for my trip?" Second, a *flight-level classification model* is required for specific, immediate advice. This model predicts the directional price movement Δp for an individual flight, f_{id}, over a short-term horizon n (e.g., 7 days). The model's output is a simple, actionable recommendation, $R \in \{\textit{Buy, Wait, Hold}\}$. This answers the user's specific, low-level question: "Should I buy this specific ticket right now, or wait?"

1.3 Report Context

This report details the design, training, and operationalization of the machine learning system that consumes the data foundation established by this project's ETL pipeline. It is the second in a two-part series. The preceding document, *Technical Report: An ETL Pipeline for Flight Price Prediction*, detailed the data ingestion, transformation, and "smart" loading processes. This document focuses exclusively on the ML architecture: the high-performance data loading strategy, the implementation of the dual-model system, model evaluation, and the robust MLOps pipeline designed to serve live predictions.

2 ML System Architecture

The machine learning system is built directly upon the data foundation established by the ETL pipeline. Its architecture is designed to achieve two primary goals: flexibility, to support the dual-model thesis from a single, generic codebase, and performance, to handle large-scale time-series data by pushing computational load to the database. This is realized through a config-driven orchestration script and a database-centric data-loading strategy.

2.1 A Config-Driven Design

The entire ML system is designed to be modular and re-usable, avoiding hard-coded logic. The main orchestration script, `train_model.py`, acts as a generic trainer. Its behavior is dictated entirely by a central configuration dictionary, `MODEL_CONFIGS`, located in `src/modeling/config.py`.

This configuration file acts as the system's *control plane*. For any given model M, the configuration specifies all necessary parameters: the target variable y (e.g., `min_price`), the specific feature engineering pipeline to use (e.g., `market_price_pipeline.py`), the model hyperparameters θ, and the final artifact's save path.

This design allows the same generic script, `train_model.py`, to orchestrate the end-to-end training of entirely different models, such as a regressor M_{reg} and a classifier M_{class}, simply by changing the configuration key provided at runtime.

2.2 The Data-to-Model Bridge (`data_loader.py`)

A critical design choice was made to address the performance bottleneck of loading millions of raw flight observations into memory. The system pushes complex data aggregation and target-engineering logic directly into the PostgreSQL database, using pandas only for the final feature transformations. The `data_loader.py` module functions as this *data-to-model bridge*, using efficient, model-specific SQL queries to deliver pre-processed datasets to the Python environment.

2.2.1 Market-Level Data Aggregation

For the *market-level regression model*, the data must be transformed from many granular observations per day to a single row representing the market for that day. The `load_aggregated_market_data` function achieves this using a SQL Common Table Expression (CTE) with a `GROUP BY` clause on `scrape_date`, `route_name`, and `departure_date`. Crucially, it calculates robust statistics like `MIN(price)` for the target p_{min} and uses `PERCENTILE_CONT(0.5)` to find the median price, as shown in the query snippet below. This SQL-side aggregation reduces the data volume by orders of magnitude before it enters the Python environment.

```sql
SELECT
    CAST(scrape_timestamp AS DATE) as scrape_date,
    route_name,
    CAST(departure_datetime_local AS DATE) as departure_date,

    -- Price aggregations
    MIN(price) as min_price,
    AVG(price) as avg_price,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY price) as median_price,

    -- Flight availability
    COUNT(flight_id) as num_flights_available,

    -- Other feature aggregations
    MIN(days_until_departure) as days_until_departure

FROM flight_observations
GROUP BY 1, 2, 3
```

Listing 1: SQL query for market-level data aggregation.

2.2.2 Time-Shifted Target Engineering

For the *flight-level classification model*, the challenge is to create the target variable y, which is the price of the same flight n days in the future, p_{t+n}. Performing this operation in pandas requires a `groupby('flight_id').shift(-n)` operation, which is computationally expensive and memory-intensive on large datasets.

The system bypasses this by delegating the task to SQL's functions in the database server. The `load_data_for_movement_model` function uses `LEAD()` to "look ahead" n rows within a partition of a single flight's history. This retrieves the future price p_{t+n} and appends it to the current row t. This elegant operation, shown below, delivers a perfectly pre-built training set (features X_t and target p_{t+n}) directly from the database.

```sql
WITH future_price_cte AS (
    SELECT
        *,
        -- Looks 'n' rows ahead within each flight's history
        LEAD(price, 7) OVER (
            PARTITION BY flight_id
            ORDER BY scrape_timestamp
        ) AS future_price
    FROM
        flight_observations
)
SELECT *
FROM future_price_cte
-- Pre-filter data to only include rows
-- where a future price was found
WHERE future_price IS NOT NULL;
```

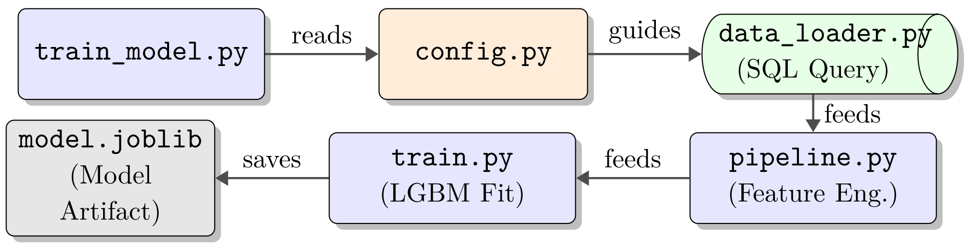
Listing 2: SQL query for time-shifted target engineering using LEAD().

2.3 The Generic Training Pipeline

This architecture culminates in a streamlined and repeatable training pipeline. The end-to-end process is as follows:

1. A user executes `train_model.py` with a model name (e.g., market_price_predictor).
2. The script fetches the corresponding configuration from `config.py`.
3. It calls the specific function in `data_loader.py` (e.g., `load_aggregated_market_data`) to retrieve the SQL-processed data.
4. This data is passed to the correct feature engineering pipeline specified in the config (e.g., `market_price_pipeline.py`).
5. The final features X and target y are fed into the generic `train.py` module.
6. The trained model artifact (containing the model θ, feature list X_{cols}, and categories C) is saved to a `.joblib` file.

This entire flow is illustrated in Figure 1.



train_model.py
reads
config.py
guides
data_loader.py (SQL Query)
feeds
pipeline.py (Feature Eng.)
feeds
train.py (LGBM Fit)
saves
model.joblib (Model Artifact)

Figure 1: The generic, config-driven training pipeline.

3 Model Deep Dive 1: Market Price Prediction (Regression)

3.1 Problem Definition

This first model addresses the high-level, strategic question a user might have: "What is a good price for my trip?" We define this as a classical regression problem. The objective is to predict the *minimum price* (p_{min}) for a given route R and departure date D.

Formally, we seek to find a function f such that:

$$p_{min} = f(R, D, X_{market}) + \epsilon$$

where X_{market} is a vector of aggregated market features for that specific route and date, and ϵ is the irreducible error. The output is a single continuous value, p_{min}, representing the expected cheapest price for that day's market.

3.2 Data & Feature Engineering

The input data for this model is generated by the `load_aggregated_market_data` function. As detailed in the previous section, this function delivers a pre-aggregated dataset where each row represents a single market, defined as a unique combination of `(scrape_date, route_name, departure_date)`.

The feature vector X for this model is composed of these aggregated statistics. Key features include: *avg_price_on_route*, which captures the market's central tendency; *num_flights_available*, which acts as a proxy for supply; *days_until_departure*, which captures the temporal booking window; and the macroeconomic indicators, such as *jet_fuel_price_usd_gal* and *us_cpi*, which provide global context on operational costs and inflation.

3.3 Model & Evaluation

Given the tabular, heterogeneous nature of the feature set (a mix of counts, prices, and temporal features), a *LightGBM Regressor* (`LGBMRegressor`) was selected for its high performance and native handling of categorical features. The model is trained to predict the target variable $y = p_{min}$.

To evaluate the model's performance, we use two primary regression metrics. The first is the *Mean Absolute Error* (MAE), which is the primary metric for business interpretation:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

The MAE provides a direct, interpretable measure of the average prediction error in dollars. For example, an MAE of \$25.50 means that, on average, the model's price prediction is \$25.50 off from the actual minimum price.

The second metric is the *Root Mean Squared Error* (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

The RMSE is used to penalize larger errors more heavily, as the squaring term gives them more weight. A low RMSE indicates the model does not make many large, egregious prediction errors, which is critical for building user trust.

4 Model Deep Dive 2: Flight Movement Prediction (Classification)

4.1 Problem Definition

This second model addresses the tactical, low-level question a user faces: "Should I buy this specific flight, `flight_id`, now, or should I wait?" We define this as a multiclass classification problem. The objective is to predict the *directional movement* of an individual flight's price, not its exact value.

Formally, let p_t be the price of a specific flight f_{id} at the current time t. We seek to find a function g that predicts the class of price movement C over a future horizon n:

$$C = g(f_{id}, t, X_{micro})$$

The output is a categorical prediction, $C \in \{\textit{increase, decrease, stable}\}$, which directly powers a human-readable recommendation for the user.

4.2 Target Engineering

The success of a classification model is critically dependent on the definition of its target variable, y. This system's secret sauce is in how this target is engineered from a continuous, time-series problem into a discrete, categorical one.

The process begins in the database, where the `load_data_for_movement_model` function uses a `LEAD(price, 7) OVER (...)` window function. This efficiently fetches the price of the same flight 7 days into the future, which we denote p_{t+7}, and provides it on the same row as the current price p_t.

The feature engineering pipeline then computes the fractional price change, $\Delta p_{\%}$, and maps this continuous value to one of the three discrete classes C. This logic is defined as:

$$\Delta p_{\%} = \frac{p_{t+7} - p_t}{p_t}$$

The target variable y is then assigned according to the following thresholds:

$$y = C = \begin{cases} \textit{increase} & \text{if } \Delta p_{\%} > 0.05 \\ \textit{decrease} & \text{if } \Delta p_{\%} < -0.05 \\ \textit{stable} & \text{otherwise} \end{cases}$$

A threshold of $\pm 5\%$ was chosen to ensure the model focuses on significant price changes, treating minor, everyday fluctuations as stable.

4.3 Data & Feature Engineering

Unlike the regression model, which uses market-level aggregates, the classification model uses *micro-level* features specific to a single `flight_id`. The feature vector X_t is heavily time-based, designed to capture the price's recent trajectory and momentum.

Key features are those calculated by the *smart uploader* during the ETL process, such as `price_t_minus_1` (the price from the last scrape), `days_since_last_scrape`, and `price_change_from_last`. These are supplemented by other time-series features created in the pipeline, such as 3-day and 5-day rolling price averages ($\bar{p}_3, \bar{p}_5$) and the flight's current `price_percentile` relative to its peers on the same day.

4.4 Model & Evaluation

A *LightGBM Classifier* (`LGBMClassifier`) is used, as it is highly effective for tabular data and scales well. The model is trained to predict the categorical target $y = C$.

Evaluation of a classifier, especially one with potentially imbalanced classes (e.g., fewer *increase* events than *stable*), requires a more nuanced approach than a simple *Accuracy* score. We use *Logarithmic Loss* (LogLoss) as the primary training metric, as it heavily penalizes predictions that are both wrong and confident. The LogLoss for a single observation is:

$$\text{LogLoss} = -\sum_{j=1}^{M} y_j \log(p_j)$$

where $M = 3$ is the number of classes, y_j is 1 if the observation belongs to class j and 0 otherwise, and p_j is the model's predicted probability for that class.

5 MLOps & Operationalization

A model is only useful if it can be reliably operationalized. The final stage of this project was to build a robust MLOps pipeline to bridge the gap between the trained model (a `.joblib` file) and an end-user prediction (an actionable recommendation). This was achieved by creating an intelligent model artifact and a resilient inference pipeline.

5.1 The Model Artifact

The `train.py` script does not simply save the final model object. Doing so would create a "brittle" artifact, as the model's performance is intrinsically tied to the exact features and data-types it was trained on.

Instead, the script saves a complete *model artifact*, A, as a single `.joblib` file. This file is a Python dictionary containing three key components:

- A_θ: The trained LightGBM model object (e.g., the `LGBMClassifier`).
- $A_{X_{cols}}$: The ordered list of all feature names $(X_1, X_2, ..., X_n)$ that the model A_θ was trained on.
- $A_{C_{map}}$: A dictionary mapping each categorical feature to its list of known, "seen" categories. For example, $C_{airline} \rightarrow$ [*'ZIPAIR', 'Delta', 'Japan Airlines'*].

5.2 A Robust Inference Pipeline (`predict.py`)

The design of this model artifact is crucial for production stability. A common failure mode in ML systems is when new, live inference data X_{new} contains a value in a categorical feature that was not present in the training data X_{train}. For example, if a new airline, *'NewAir'*, appears in the scraping data, but the model's $C_{airline}$ map only contains {*'ZIPAIR', 'Delta'*}, the prediction would fail with an "unseen category" error.

The `predict.py` script is designed to be resilient to this exact problem. The inference pipeline is as follows:

1. The script loads the complete artifact A (not just the model).
2. It extracts the list of required features $A_{X_{cols}}$ and the category map $A_{C_{map}}$.
3. It aligns the new data X_{new} to match the features $A_{X_{cols}}$, dropping any columns that were not used in training and adding any that are missing (as `null`).
4. It then iterates through $A_{C_{map}}$ and explicitly re-casts each categorical column in X_{new} using pandas' `Categorical` dtype, passing the saved list as the `categories` parameter.

This final step forces X_{new} to conform to the exact data schema of X_{train}. Any unseen values (like *'NewAir'*) are automatically converted to `null`, which the LightGBM model is natively equipped to handle. The benefit of this design is the prevention of production-breaking errors, ensuring the inference pipeline is robust and resilient to data drift.

5.3 Serving Predictions

This dual-model architecture is fully operationalized via two command-line scripts, each designed to answer one of the two core questions in our thesis.

To answer the strategic question, "What is a good price for my trip?", the user executes `predict_price.py`. This script loads the regression model artifact A_{reg} (e.g., market_price_predictor.joblib), runs the robust inference pipeline on the aggregated market data, and outputs a single, continuous, predicted dollar value for p_{min}.

To answer the tactical question, "Should I buy this specific ticket now?", the user executes `predict_movement.py`. This script loads the classification model artifact A_{class} (e.g., flight_movement_predictor.joblib), predicts the probability for each class $C \in \{$*increase, decrease, stable*$\}$, and provides a simple, actionable recommendation, such as: *Recommendation: Wait.*