DATA CRUNCH

AgroChill Forecasting System

By FoutteBytes- 106
University of Sri Jayawardanapura

1. Introduction

1.1. Project Context & Objectives

This document details the solution developed for the Data Crunch - Final Round competition. The challenge, themed "Legacy of the Market King: The Freezer Gambit," required to build a time series forecasting system for AgroChill, a fictional entity aiming to optimize profits from perishable agricultural goods.

The core objective was to predict weekly fresh prices of fruits and vegetables across various regions 1 to 4 weeks ahead. This forecasting capability is intended to support AgroChill's "Freezer Gambit" strategy, allowing informed decisions on whether to sell produce immediately in the weekly market or utilize cold storage to wait for potentially better future prices, thus mitigating losses from spoilage and maximizing revenue.

Key technical requirements included handling incoming data streams, implementing automated model retraining, providing rolling forecasts based on the latest available data, and adhering to strict operational constraints regarding memory usage, Docker image size, and prediction latency.

1.2. Proposed Solution Overview

Our team developed a comprehensive solution encompassing data preprocessing, extensive feature engineering, rigorous model selection and tuning, and a robust API deployment with automated retraining capabilities.

- **Data Pipeline:** Processes raw weather and price data, handles cleaning, merging, and feature engineering. Incorporates new data saved via the API into subsequent retraining cycles.
- **Modeling:** Explored multiple Gradient Boosted Tree models (LightGBM, XGBoost, CatBoost) and selected XGBoost based on superior performance on a hold-out evaluation set after hyperparameter tuning with Optuna. Four separate models are trained for 1, 2, 3, and 4-week ahead forecasts.
- **API (FastAPI)**: Provides endpoints for retrieving rolling 4-week forecasts (`/api/predict`), ingesting new weather (`/api/data/weather`) and price (`/api/data/prices`) data, checking system status (`/api/status`), and manually triggering retraining (`/api/retrain`).
- **Automated Retraining (APScheduler)**: A background job automatically aggregates all available data, re-tunes hyperparameters, retrains the XGBoost models, and updates the models used by the live API on a regular schedule (default 24 hours).
- **Deployment (Docker)**: The entire application, including models and dependencies, is containerized using Docker for reproducibility and ease of deployment.

1.3. Document Structure

This document is organized as follows:

- **Section 1**: Introduction: Provides project context, objectives, and an overview of the solution.
- **Section 2**: Problem Understanding & Data Analysis: Details the data provided, preprocessing steps, and insights from Exploratory Data Analysis (EDA).
- **Section 3**: Model Selection & Feature Engineering: Explains the rationale for feature choices and the comparative process leading to the selection of XGBoost.
- **Section 4**: Data Pipeline Strategy: Describes the flow of data from ingestion to prediction and retraining.
- **Section 5**: System Design & Architecture: Outlines the overall software architecture, API design, and deployment strategy (Docker).
- **Section 6**: Business Insights & Recommendations: Derives actionable insights from the data and model, providing suggestions for AgroChill.

2. Problem Understanding & Data Analysis

This section details our comprehension of the AgroChill forecasting challenge, the structure of the provided data, the cleaning and preprocessing steps undertaken, and the key insights derived from Exploratory Data Analysis (EDA) that guided the development of our solution.

2.1. Understanding the AgroChill Challenge

The core business problem presented by AgroChill, under the guidance of Magnus Greenvale, revolves around optimizing the profitability of perishable agricultural goods within a system constrained by traditional weekly markets. The inherent risk lies in the potential spoilage of unsold produce if not sold immediately at the weekly market window. The proposed "Freezer Gambit" strategy aims to mitigate this risk by using AgroChill's cold storage facilities to hold produce for sale at potentially more favorable future market prices.

The success of this strategy hinges entirely on accurate price forecasting. Therefore, the primary technical objective of this project was to develop a robust time series forecasting system capable of:

- **Predicting** weekly fresh prices for various fruits and vegetables across multiple geographical regions (economic centers) 1 to 4 weeks into the future.
- Making **rolling forecasts**, meaning predictions generated at any point in time must utilize all available historical data up to that point. As new weekly data arrives, the forecast horizon should roll forward accordingly.
- **Ingesting new data** for weather and prices dynamically via a defined API.
- **Automatically retraining** the predictive models periodically to adapt to new data patterns and maintain forecast accuracy over time.
- Operating within **resource constraints** and delivering predictions with **low latency**.

2.2. Data Description

Two primary datasets were provided for training and evaluation:

- **Weather Data**: Contained historical records presumably associated with weekly market cycles, including:
    - `Date`: The date of the record.
    - `Region`: The geographical economic center.
    - `Temperature (K)`: Temperature in Kelvin.
    - `Rainfall (mm)`: Precipitation in millimeters.
    - `Humidity (%)`: Relative humidity percentage.
    - `Crop Yield Impact Score`: A calculated metric indicating the favorability of environmental conditions for crop yield (higher score implies better conditions).

- **Price Data**: Contained historical price records, including:
    - `Date`: The date corresponding to the weather record.

    - `Region`: Matching the weather record region.
    - `Commodity`: The specific fruit or vegetable (Mapped from 'Crop' if necessary).
    - `Price per Unit (Silver Drachma/kg)`: The target variable for prediction.
    - `Type`: Classification as 'Fruit' or 'Vegetable'.

2.3. Data Preprocessing and Cleaning

Before analysis and modeling, the following preprocessing steps were performed using Python and the Pandas library:

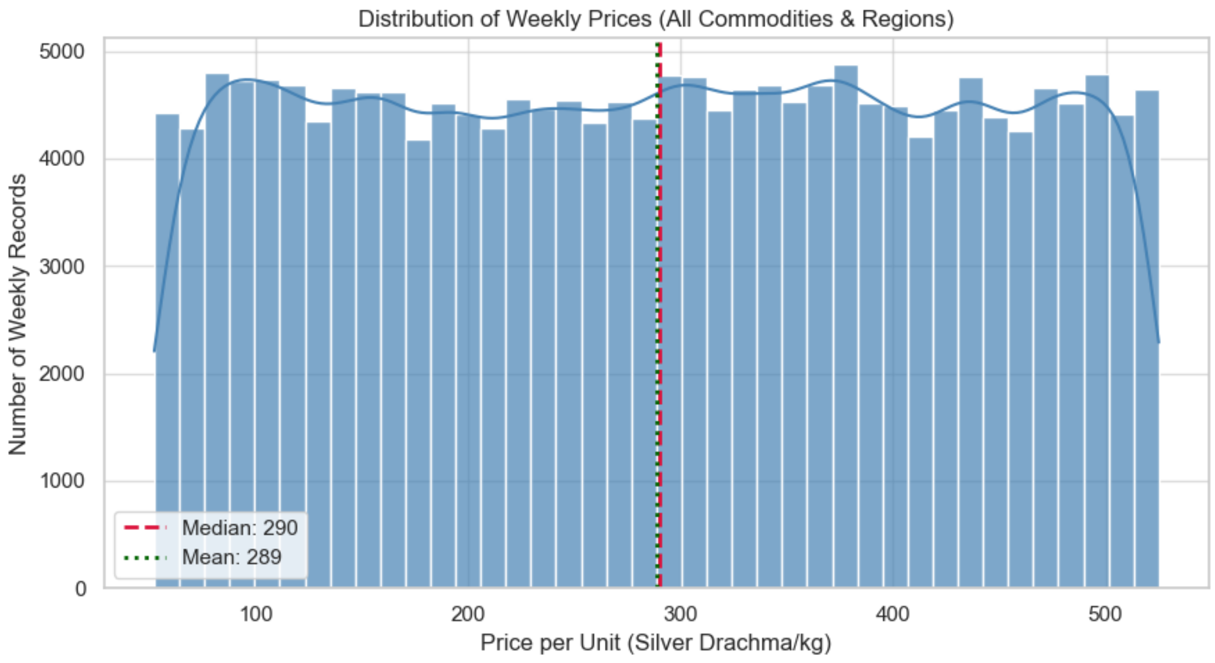
- **Loading**: Data was loaded from the provided CSV files into Pandas DataFrames.
- **Column Name Cleaning**: All column names were converted to lowercase, and special characters (except underscores) were removed (e.g., `Price per Unit (Silver Drachma/kg)` became `priceperunitsilverdrachmakg`).
- **Data Type Conversion**: Date columns converted to datetime objects; numerical and categorical types confirmed.
- **Duplicate Handling**: Significant weather duplicates (`Date, Region`) and minor price duplicates (`Date, Region, Commodity`) were identified and removed, keeping the latest entry (`keep='last'` used in retraining context, `keep='first'` in initial EDA load).
- **Missing Value Check**: No missing values were found in the original datasets. `fillna(0)` was used to handle NaNs generated during feature engineering (lags/rolls).
- **Merging**: Weather and Price data combined via an inner merge on Date and Region.
- **Sorting**: Merged data critically sorted by `Region, Commodity`, then `Date` for time series integrity.

2.4. Exploratory Data Analysis (EDA)

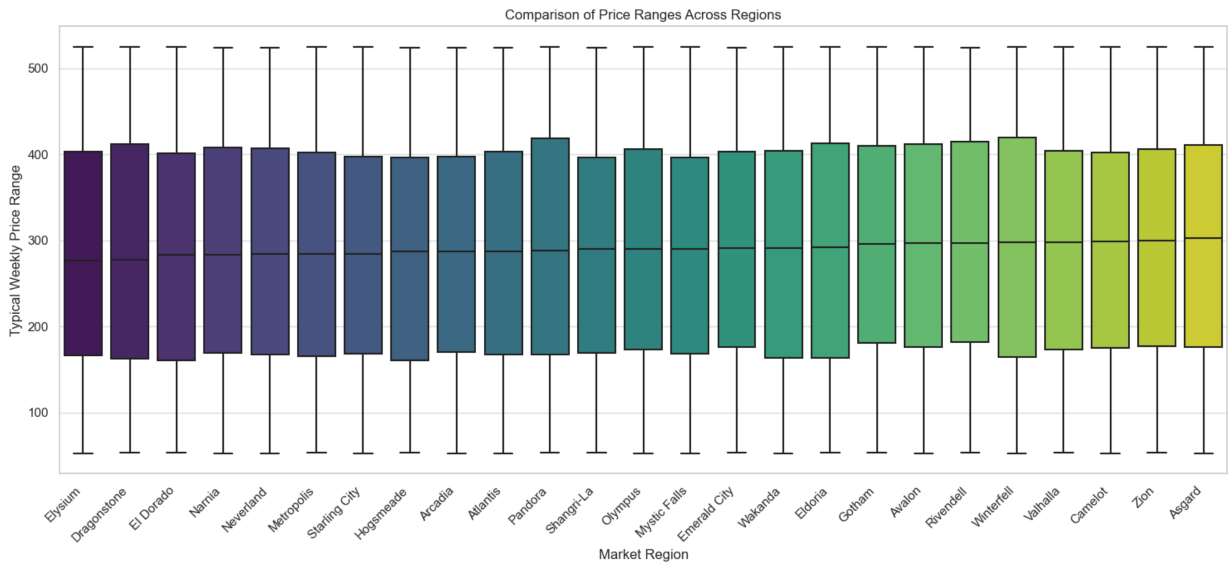
EDA was performed on the preprocessed training dataset. Key findings include:

2.4.1. Price Characteristics

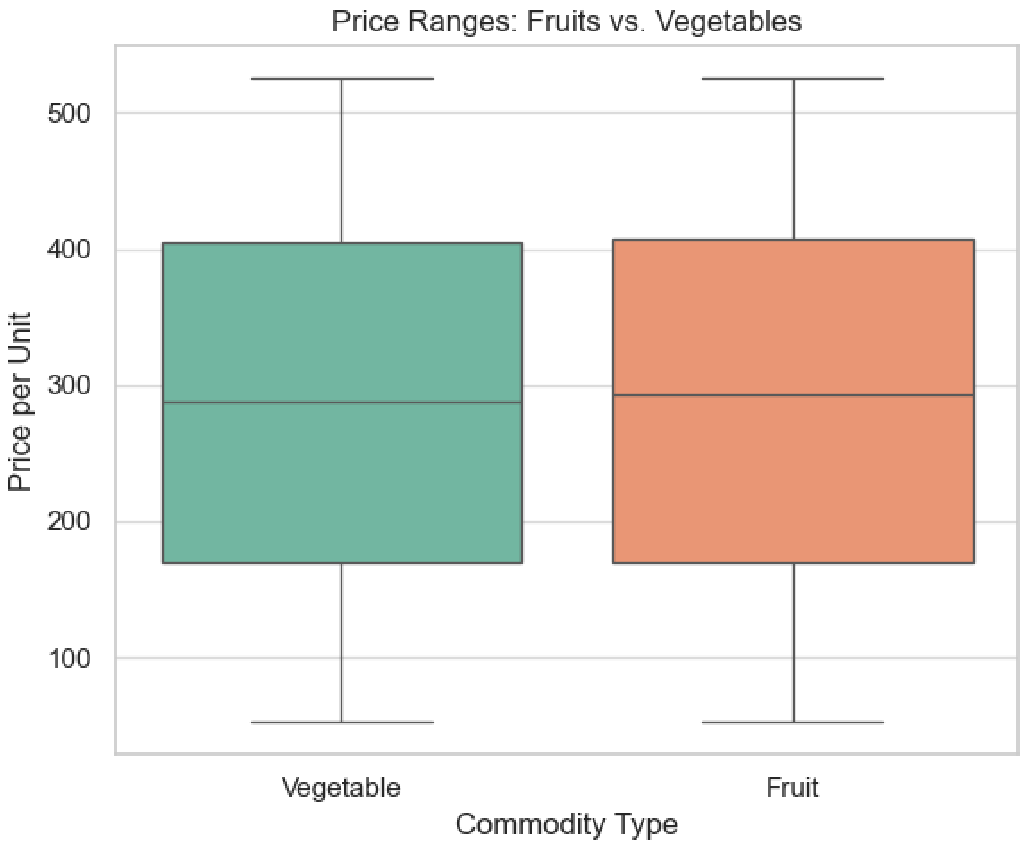
- **Overall Distribution**: Prices span a wide range (approx. 53 to 525 Silver Drachma/kg) with a distribution that is somewhat flat but slightly right-skewed (Mean: ~289, Median: ~290). This wide spread and lack of a single strong peak suggests diverse pricing behaviors across commodities/regions and supports the use of non-linear models.



- **Regional Variation**: Box plots comparing regions show clear differences in median price levels (e.g., Elysium lowest at ~276, Asgard highest at ~303) and price variability (spread of the boxes). This confirms Region is a vital predictive feature.

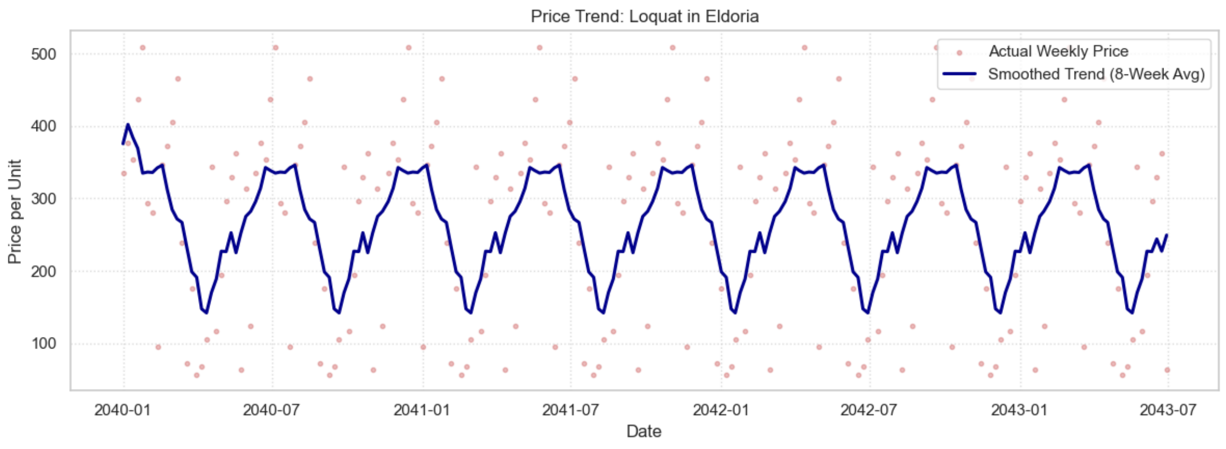


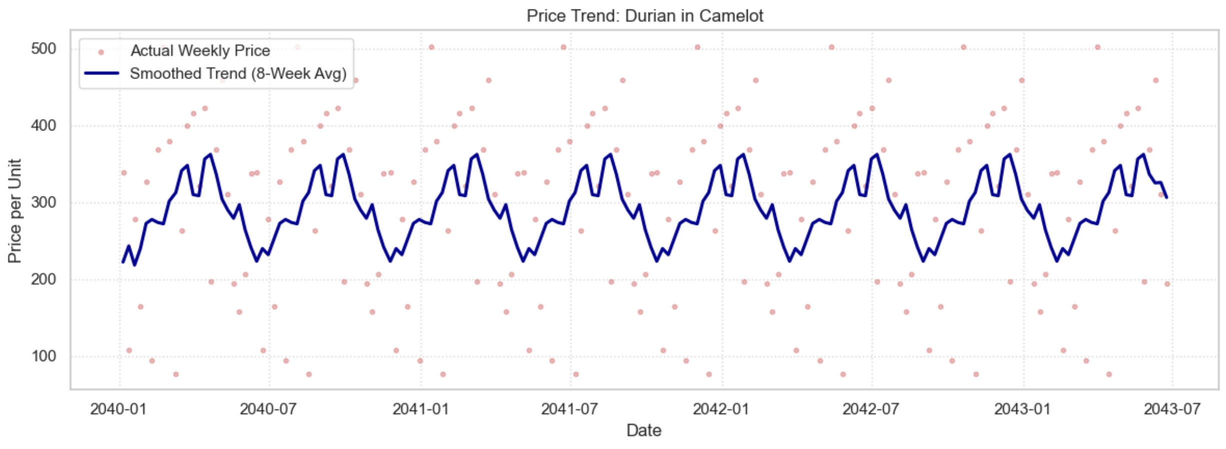
- **Type Variation**: The price distributions for Fruits (Median ~293) and Vegetables (Median ~288) are broadly similar in terms of central tendency and spread. This suggests that while Type might offer some information, the specific Commodity is likely a much stronger predictor.



2.4.2. Temporal Patterns

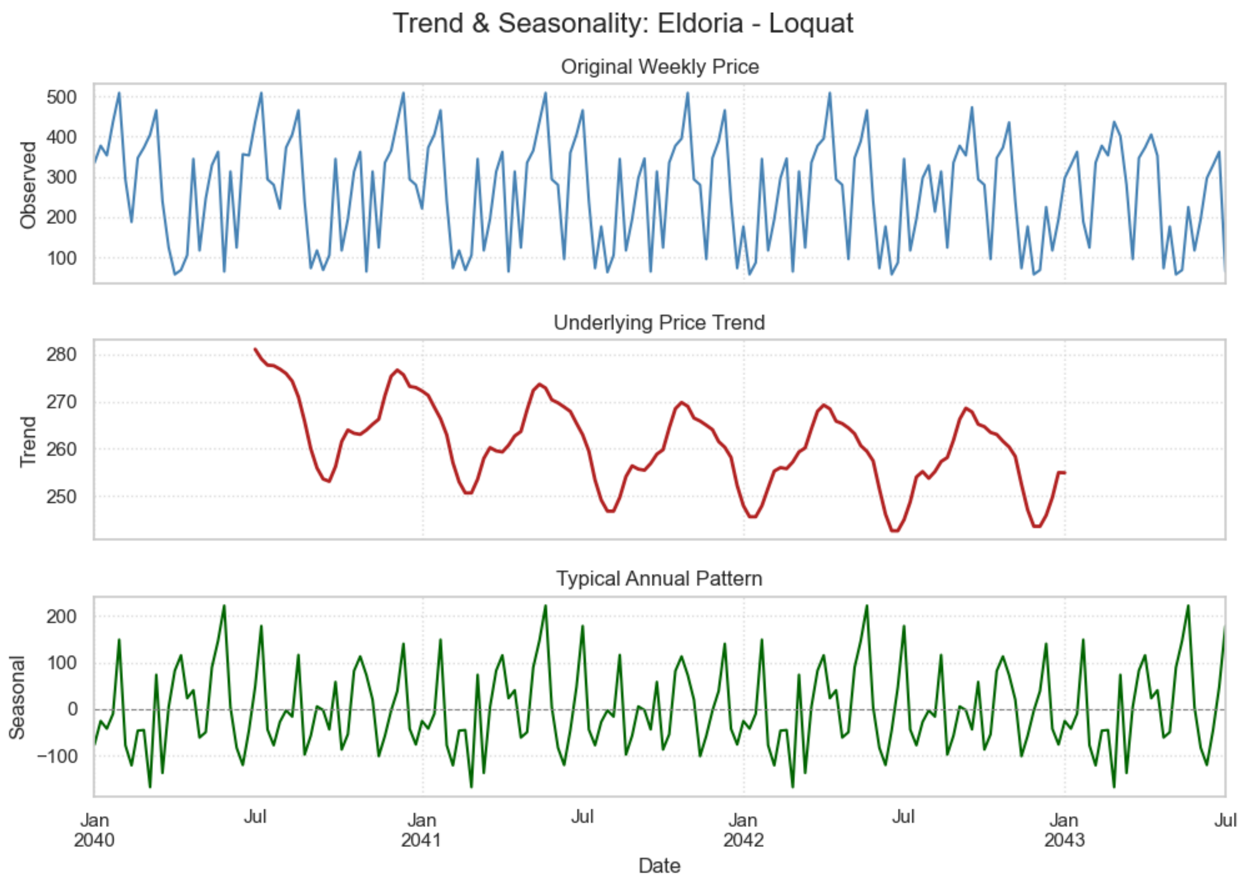
- **Individual Price Trends**: Examining individual time series (e.g., Loquat in Eldoria, Durian in Camelot) reveals distinct patterns, volatility, and potential seasonality specific to each item/location. Smoothing these series with an 8-week rolling average highlights underlying trends and cyclical behavior more clearly than the noisy weekly data.





This strongly justifies a time-series approach with appropriate features.

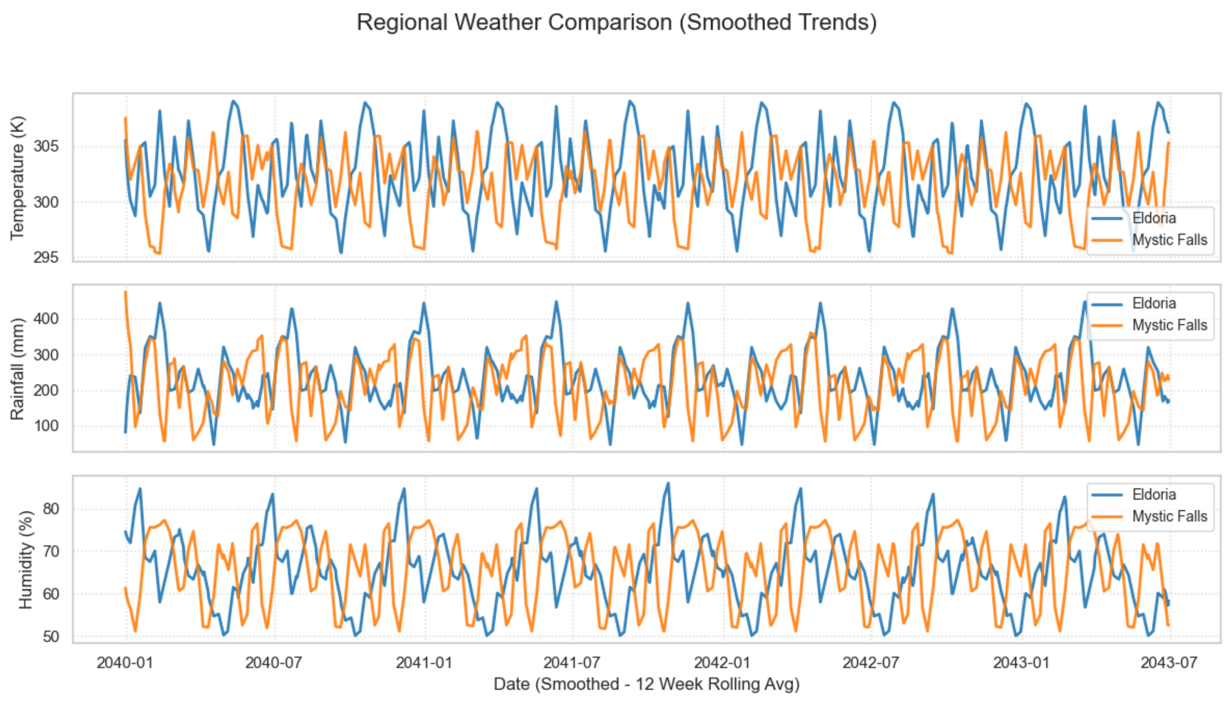
- **Seasonality**: Seasonal decomposition of the 'Eldoria - Loquat' weekly price series clearly isolates a repeating annual pattern (Seasonal component) from the longer-term underlying trend.



This confirms the presence of strong seasonality and necessitates features like `month` or `weekofyear` to capture this predictable variation.

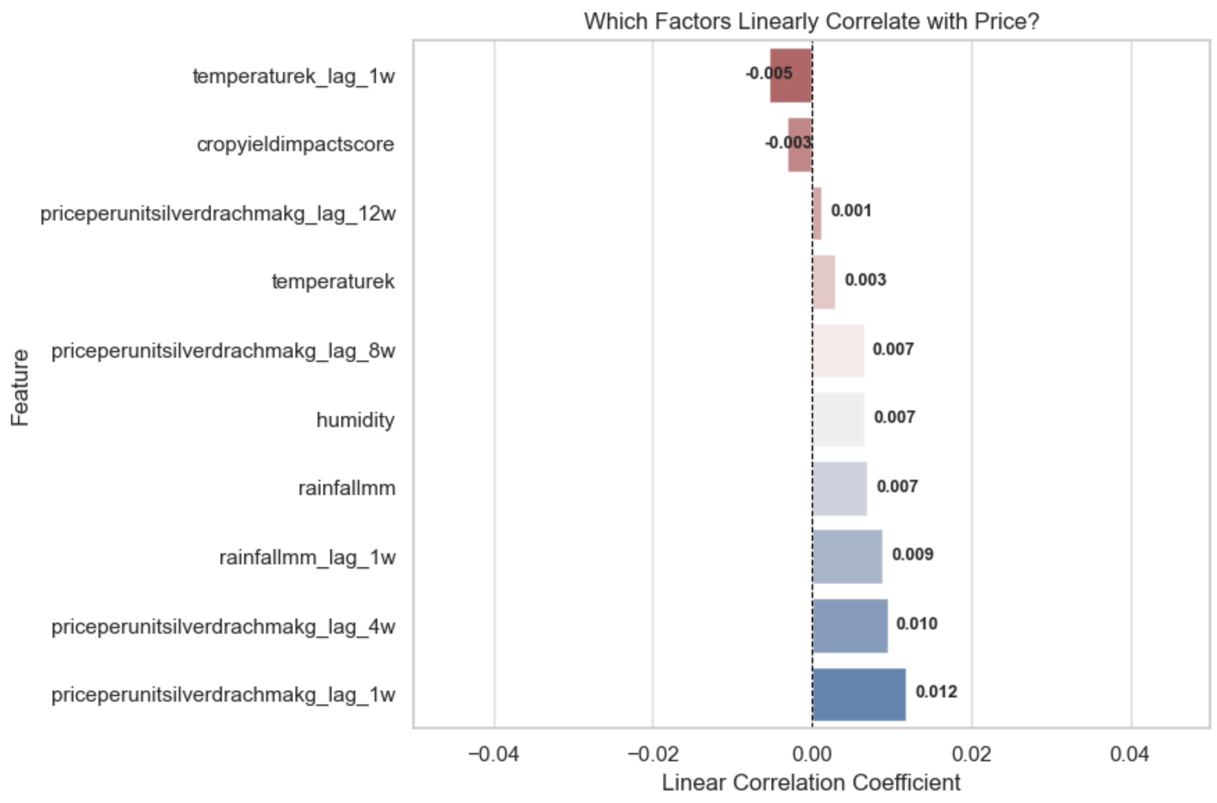
2.4.3. Weather Patterns & Correlation

- Regional Weather Trends: Comparing smoothed (12-week rolling average) weather patterns for representative regions like Eldoria and Mystic Falls shows distinct regional climates and clear seasonality, especially in temperature. Rainfall and humidity patterns also differ between regions.



This provides context for why region is important and suggests weather features might interact with region.

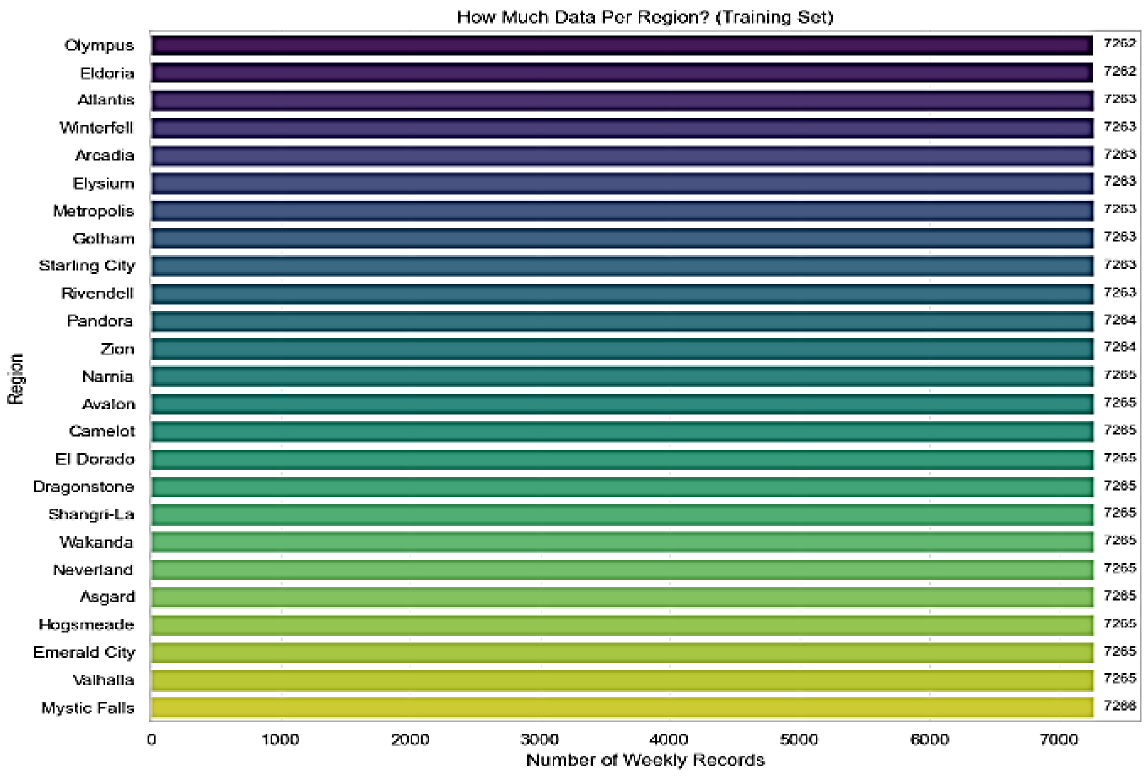
- **Correlation Analysis**: A bar chart showing linear correlations with the current price reveals:
  - The strongest positive correlations are with recent price lags (`price_lag_1w`, `price_lag_4w`), confirming strong autocorrelation.
  - Correlations with weather variables (current and lagged `temperaturek`, `rainfallmm`, `humidity`) and `cropyieldimpactscore` are very weak (close to zero).



- **Insight**: This lack of strong linear correlation with weather/yield doesn't mean they are unimportant, but that their influence is likely non-linear or indirect. This strongly justifies using models like XGBoost that can capture complex interactions, and reinforces the importance of lagged price features.

2.4.4. Data Volume

- **Records per Region**: The training data is remarkably well-balanced across regions, with each having a very similar number of records (approx. 7262-7266). This reduces concerns about model bias towards heavily represented regions.



2.5. Key Findings & Impact on Solution Design

- **Strong Time Dependence & Autocorrelation:** → **Action:** Implemented extensive lag features and rolling window statistics.
- **Seasonality:** → **Action**: Included `month` and `weekofyear` features.
- **Regional & Commodity Importance:** → **Action**: Used `Region, Commodity, Type` as categorical features (One-Hot Encoded).
- **Non-Linear Relationships:** → **Action**: Selected XGBoost after comparison, capable of capturing complex interactions.
- **Data Balance:** → **Action**: Increased confidence in a single model structure applicable across regions.
- **Rolling Forecast Requirement:** → **Action**: Designed `/api/predict` to load all current data, find the true latest point, and generate features based on actual history for that point.
- **Retraining Requirement:** → **Action**: Implemented automated retraining pipeline (`run_retraining_job` with `APScheduler`) using aggregated data, Optuna, and live model updates.

3. Model Selection & Feature Engineering

This section outlines the process undertaken for selecting the appropriate machine learning model and the feature engineering techniques employed to capture the complex dynamics of the crop price forecasting problem, while respecting the competition's resource constraints.

3.1. Requirements & Challenges

The primary requirements influencing model selection and feature engineering were:

- **Accuracy**: Predict prices 1-4 weeks ahead with the lowest possible error (RMSE).
- **Rolling Forecast**: The model must operate on the latest available data.
- **Time Series Nature**: Prices exhibit strong temporal dependencies (autocorrelation, seasonality, trends).
- **Multiple Factors**: Prices are likely influenced by historical prices, weather conditions (potentially lagged), seasonality, region, and specific commodity characteristics.
- **Constraints**: Strict limits on inference RAM (< 2GB), Docker image size (< 8GB), and an implicit need for low prediction latency.

The EDA (Section 2.4) revealed weak linear correlations between price and contemporaneous weather/yield features, suggesting the need for models capable of handling non-linear relationships and justifying robust feature engineering.

3.2. Feature Engineering Strategy

Based on the EDA and time series principles, the following features were engineered from the preprocessed (cleaned, merged, sorted) data to provide the model with relevant historical and contextual information:

- **Lag Features**:
    - **Purpose**: Capture autocorrelation and direct dependence on recent past values. Given the weekly nature, recent weeks are highly indicative. Longer lags capture lower-frequency effects.
    - **Implementation**: For the target variable (`priceperunitsilverdrachmakg`) and key weather indicators (`temperaturek, rainfallmm, humidity, cropyieldimpactscore`), lagged values were created for 1, 4, 8, and 12 weeks prior (`_lag_1w, _lag_4w, _lag_8w, _lag_12w`). These lags were calculated within each `Region-Commodity` group to prevent data leakage across different items.
    - **Effectiveness**: Lagged price features consistently showed the highest correlation with the current price and proved highly important in model training.

- **Rolling Window Features**:
    - **Purpose**: Capture recent trends and volatility by summarizing data over moving windows.

- **Implementation**: Rolling mean and standard deviation were calculated for the same set of variables as lags, using window sizes of 4, 8, and 12 weeks (e.g., `_roll_mean_4w`, `_roll_std_4w`). To prevent target leakage, the rolling window calculations excluded the current observation (`.shift(1).rolling(...)`). Calculations were performed within Region-Commodity groups.
- **Effectiveness**: These features provide context about recent price levels and stability, contributing moderately to model performance.

- **Time-Based Features**:
  - **Purpose**: Capture seasonality and calendar effects identified during EDA.
  - **Implementation**: Extracted `year`, `month`, `weekofyear`, and `dayofweek` from the date column.
  - **Effectiveness**: Essential for capturing the strong annual seasonality observed in price and weather patterns.

- **Categorical Features**:
  - **Purpose**: Allow the model to learn specific behaviors associated with different locations, items, and types.
  - **Implementation**: `Region`, `Commodity`, and `Type` were retained as features. They were handled using One-Hot Encoding within a scikit-learn pipeline before being fed to the tree-based models (LightGBM/XGBoost). For CatBoost experiments, its internal categorical handling was leveraged.
  - **Effectiveness**: Region and Commodity proved to be important features, reflecting the variations observed in the EDA.

- **Handling Generated NaNs**: Lag and rolling features inherently produce NaN values at the beginning of each time series group. These were initially handled by forward/backward filling for EDA plots, but for model training, a simple `fillna(0)` strategy was adopted after initial row dropping based on the longest target horizon. While more sophisticated imputation could be explored, this provided a baseline and worked reasonably well with tree-based models.

3.3. Model Selection Process

Given the time series nature, the need to capture non-linearities, and the strict resource constraints, Gradient Boosted Decision Trees (GBDTs) were identified as the most promising class of models. We systematically evaluated three leading GBDT implementations:

- **LightGBM:** Known for its speed and memory efficiency due to histogram-based splitting and leaf-wise growth. Often provides state-of-the-art accuracy.
- **XGBoost**: Another highly popular and powerful GBDT library, often achieving top accuracy. Can sometimes be more memory-intensive than LightGBM.

- **CatBoost**: Performs well, particularly noted for its excellent built-in handling of categorical features, potentially simplifying preprocessing.

  A Neural Network approach (LSTM) was also considered but preliminarily discarded for the primary solution due to the high risk of exceeding memory/latency constraints and the increased implementation complexity compared to GBDTs for this structured, tabular dataset.

  Methodology:
  - **Separate** Training: Dedicated training scripts (`train_lgbm.py, train_xgboost.py, train_catboost.py`) were created.
  - **Consistent Features**: All models used the same core feature engineering approach described above (though CatBoost could internally handle categoricals).
  - **Preprocessing**: A standard preprocessing pipeline (StandardScaler for numerical, OneHotEncoder for categorical) was used for LightGBM and XGBoost. CatBoost used StandardScaler for numerical and its internal handling for categoricals.
  - **Hyperparameter Optimization**: Optuna was used within each training script to tune key hyperparameters (e.g., learning rate, tree depth, regularization, estimators) for each of the 4 forecast horizons independently, optimizing for validation RMSE.
  - **Evaluation**: Each model type (after tuning) was evaluated on the separate, unseen evaluation dataset to obtain unbiased performance metrics (RMSE for each horizon).

3.4. Model Comparison & Final Selection

The performance on the evaluation set was the primary criterion for final model selection:

Horizon	LightGBM (Eval RMSE)	XGBoost (Eval RMSE)	CatBoost (Eval RMSE)	Selected Model
1w	136.72	**106.56**	145.51	XGBoost
2w	126.87	**120.65**	151.22	XGBoost
3w	151.03	**123.77**	168.87	XGBoost
4w	161.95	**139.79**	162.44	XGBoost

Justification for XGBoost:

As clearly shown in the table, the **XGBoost models consistently achieved significantly lower RMSE** across all four prediction horizons compared to both LightGBM and CatBoost on the unseen evaluation data. While all three models showed promise during internal tuning (Optuna best validation RMSEs were similar around ~137), XGBoost demonstrated superior generalization performance for this specific dataset and feature set.

Although XGBoost can sometimes have higher memory requirements than LightGBM, the substantial accuracy improvement justified its selection as the primary model. Preliminary checks during development indicated that the tuned XGBoost models were likely to operate within the specified resource constraints, though rigorous benchmarking in the final container is essential. An ensemble approach was considered but rejected due to the significantly increased resource usage (memory, latency) and the clear performance advantage of XGBoost alone.

Therefore, the final submitted solution utilizes **four independently tuned XGBoost models**, one for each forecast horizon (1w, 2w, 3w, 4w), integrated within a FastAPI application featuring automated retraining and rolling forecast capabilities.

4. Data Pipeline Strategy

This section details the strategy and design of the data pipeline implemented in the AgroChill system. The primary goal of the pipeline is to effectively ingest new data, process it, and utilize it to maintain the accuracy and relevance of the forecasting models over time through automated retraining, while also enabling real-time rolling forecasts based on the most current information available.

4.1. Pipeline Objectives

The data pipeline was designed to meet the following key objectives derived from the competition requirements:

- **Data Ingestion**: Provide a robust mechanism for accepting new weather and price data points as they become available.
- **Data Persistence**: Reliably store incoming data for future use in model retraining.
- Automated Model Adaptation: Periodically and automatically retrain the forecasting models using the complete historical dataset, including newly ingested data.
- **Consistency**: Ensure that data processing and feature engineering steps are applied consistently during both training/retraining and prediction phases.
- **Rolling Forecast Enablement**: Facilitate the generation of predictions based on the absolute latest data point available in the system at the time of a prediction request.

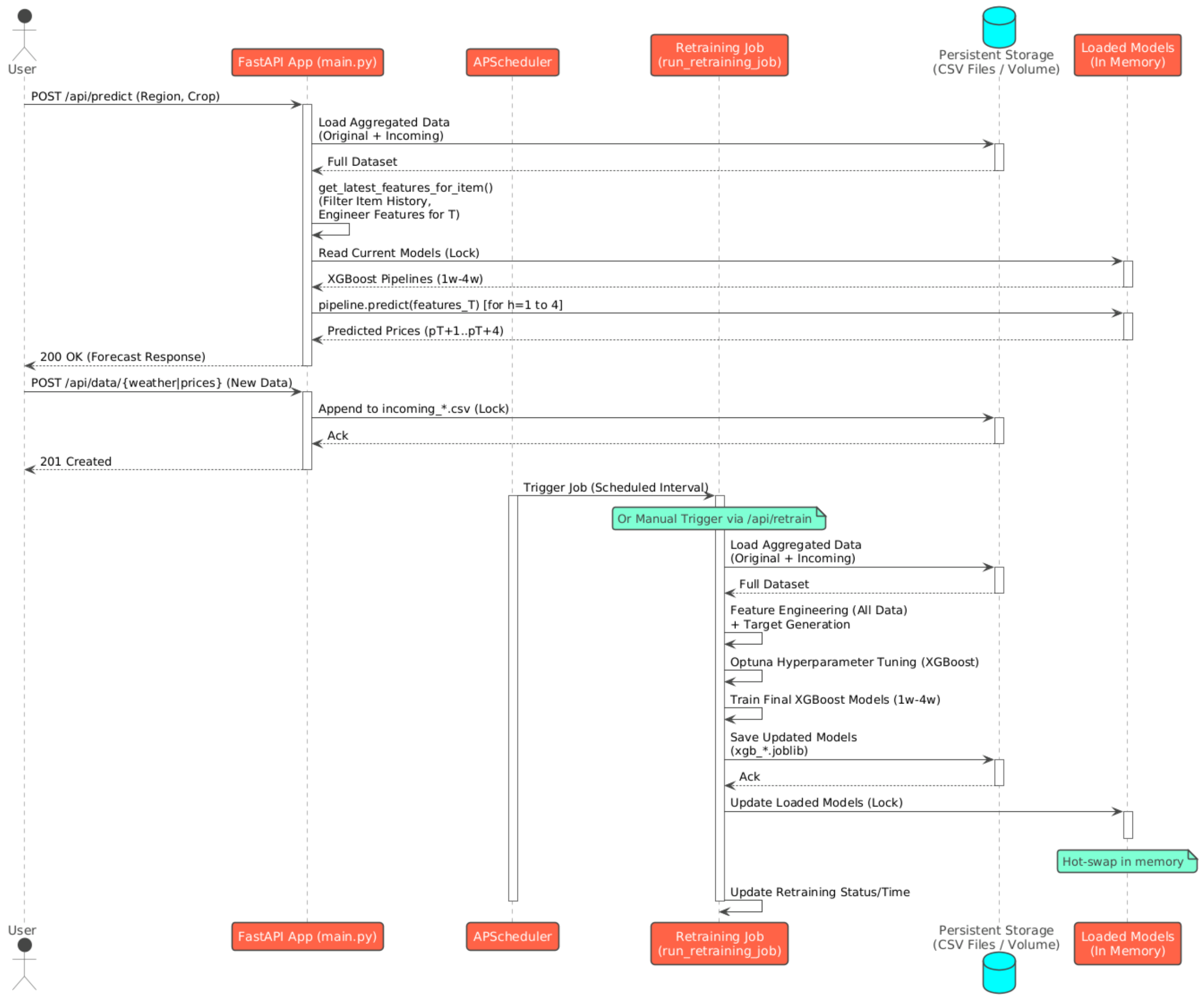
4.2. Pipeline Design Overview

A **batch retraining pipeline** approach was chosen as the core strategy. This involves accumulating new data over a period and then periodically retraining the models using the entire updated dataset. This contrasts with online learning (updating models incrementally with each data point), which adds significant complexity and wasn't deemed necessary given the likely weekly granularity of the core price data.

The pipeline consists of the following key stages:

- **Ingestion**: API endpoints receive new data.
- **Persistence**: Received data is stored persistently.
- **Triggering**: Retraining is initiated automatically on a schedule or manually via an API call.
- **Aggregation**: The retraining job loads both original training data and all persisted incoming data.
- **Processing & Feature Engineering**: The aggregated dataset undergoes cleaning, merging, sorting, and the full feature engineering process.
- **Tuning & Training**: Hyperparameters are re-tuned (using Optuna), and new models (XGBoost) are trained on the latest aggregated, featured data for each forecast horizon.
- **Model Deployment (Hot Swap)**: The newly trained models replace the existing models currently loaded in the running API's memory.

- **Prediction**: The prediction endpoint dynamically uses the aggregated data to generate features for the requested item's latest time point and applies the currently loaded models.


User
FastAPI App (main.py)
APScheduler
Retraining Job (run_retraining_job)
Persistent Storage (CSV Files / Volume)
Loaded Models (In Memory)
POST /api/predict (Region, Crop)
Load Aggregated Data (Original + Incoming)
Full Dataset
get_latest_features_for_item() (Filter Item History, Engineer Features for T)
Read Current Models (Lock)
XGBoost Pipelines (1w-4w)
pipeline.predict(features_T) [for h=1 to 4]
Predicted Prices (pT+1..pT+4)
200 OK (Forecast Response)
POST /api/data/{weather|prices} (New Data)
Append to incoming_*.csv (Lock)
Ack
201 Created
Trigger Job (Scheduled Interval)
Or Manual Trigger via /api/retrain
Load Aggregated Data (Original + Incoming)
Full Dataset
Feature Engineering (All Data) + Target Generation
Optuna Hyperparameter Tuning (XGBoost)
Train Final XGBoost Models (1w-4w)
Save Updated Models (xgb_*.joblib)
Ack
Update Loaded Models (Lock)
Hot-swap in memory
Update Retraining Status/Time

4.3. Handling Incoming Data Streams

Our analysis identified several limitations and areas for potential improvement:

- **API Endpoints**: Dedicated POST endpoints (`/api/data/weather` and `/api/data/prices`) are implemented using FastAPI.
- **Data Validation**: Pydantic models (`WeatherDataPost, PriceDataPost`) define the expected JSON structure (including nested weatherData and priceData objects as per `apis.yml`) and perform initial data type validation upon request receipt.
- **Data Preparation**: Inside the endpoints, the received Pydantic objects are converted into Pandas DataFrames. Column names are cleaned to match the internal convention (e.g., `temp -> temperaturek, crop -> commodity`), and default values are

added for any columns expected by the training process but not provided by the API specification (e.g., `cropyieldimpactscore, type`).

- **Persistence Strategy:**
    - **Choice**: Simple CSV file appending was chosen for persistence (`incoming_weather_data.csv, incoming_price_data.csv`) due to its simplicity and suitability for the competition context. This avoids the overhead of setting up and managing a database within the Docker container.
    - **Concurrency Control**: To handle potentially simultaneous API requests writing to the same file, the `python-filelock` library is used. A lock file (`.lock`) is acquired before appending to the CSV, ensuring write operations are serialized and preventing data corruption.
    - **Volume Mounting**: This file-based persistence necessitates mounting a host directory to the container's `/app/data` volume when running via Docker (`-v ./data:/app/data`). This ensures the `incoming_*.csv` files persist even if the container restarts and are accessible for reading during retraining.

4.4. Automated Retraining Pipeline (`run_retraining_job`)

This asynchronous function, managed within main.py, orchestrates the model update process:

- **Trigger**: Executed automatically by `APScheduler` based on the `RETRAIN_INTERVAL_HOURS` setting (default 24 hours) or manually via `POST /api/retrain`. A flag (`retraining_in_progress`) prevents concurrent executions.
- **Data Aggregation**: Calls `load_and_preprocess_data_agg()`, which reads:
    - The original `weather_train_data.csv` and `price_train_data.csv.`
    - All data currently stored in `incoming_weather_data.csv` and `incoming_price_data.csv` (using FileLock for safe reading).
    - It concatenates, cleans, de-duplicates (keeping the latest entries in case of overlaps), merges, and sorts this combined dataset.
- **Feature Engineering**: Applies the full feature engineering process (`engineer_features(..., generate_targets=True)`) to the entire aggregated dataset to create features and the multi-horizon target variables.
- **Iterative Tuning & Training**: Loops through forecast horizons (1w to 4w):
    - Selects the relevant features (`X_train`) and target (`y_train`) for the current horizon.
    - Builds the standard preprocessing structure (Scaler + OHE).
    - Calls `optimize_and_train_xgb_pipeline()`, which:
        - Uses Optuna (if available) to find the best hyperparameters for XGBoost by running multiple trials on splits of the current aggregated data.
        - Trains a final XGBoost model using the best parameters found on the entire aggregated data for that horizon.
        - Saves the resulting scikit-learn `Pipeline` (containing the fitted preprocessor and the trained XGBoost model) to a horizon-specific

        `.joblib` file in the `saved_models_xgb` directory, overwriting the previous version.
    - Loads the newly saved model back.
- **Live Model Update**: If training succeeds for all 4 horizons, the function acquires an asynchronous lock (`model_lock`) and safely replaces the contents of the global `model_pipelines` dictionary (used by the `/api/predict` endpoint) with the newly loaded models. This ensures a near-instantaneous "hot swap" of the models used for predictions without API downtime.
- **Status Update**: Records the completion time and status (`Success` or `Failed/Incomplete`) in global variables accessible via the `/api/status` endpoint.

4.5. Prediction Pipeline (Rolling Forecast)

The `POST /api/predict` endpoint implements the rolling forecast logic dynamically for each request:

- **Load Current Data**: It calls `load_and_preprocess_data_agg(for_predict=True)` to get the latest complete dataset (original + all incoming).
- **Identify Latest & Generate Features**: It calls `get_latest_features_for_item()`, which filters the full dataset for the requested `region/crop`, finds the absolute latest timestamp (`T`), checks for sufficient history (`MIN_HISTORY_WEEKS`), and then runs `engineer_features(..., generate_targets=False)` on the relevant historical slice ending at `T`. This generates a single row DataFrame containing features accurately representing the state at time `T`.
- **Predict**: It uses the currently loaded models (from the global `model_pipelines` dictionary, which reflects the latest successful retraining run) to predict horizons `h=1` to `4` using the single feature row generated in the previous step. Forecast dates are calculated relative to the identified latest timestamp `T`.

This design ensures predictions always use the freshest possible data context for feature generation and the most recently trained available models.

4.6. Design Considerations & Trade-offs

- **Batch vs. Online Retraining**: Batch retraining was chosen for simplicity and its suitability for weekly data patterns. Online learning would require more complex state management and potentially different model types.
- **File vs. Database Persistence**: CSV file appending with locking provides basic persistence suitable for the competition scope. A production system with higher data volume or concurrency would benefit from a proper database (e.g., SQLite, PostgreSQL) for robustness and easier querying.
- **Pipeline Consistency**: Using scikit-learn `Pipeline` objects (saved via `joblib`) bundles the preprocessing steps (scaling, encoding) with the trained model, ensuring the

exact same transformations are applied during retraining and prediction, preventing training-serving skew.

- **In-Memory Models**: Models are loaded into memory at startup and updated after retraining. This prioritizes prediction latency but requires sufficient RAM and careful synchronization (`model_lock`) during updates.

5. System Design & Architecture

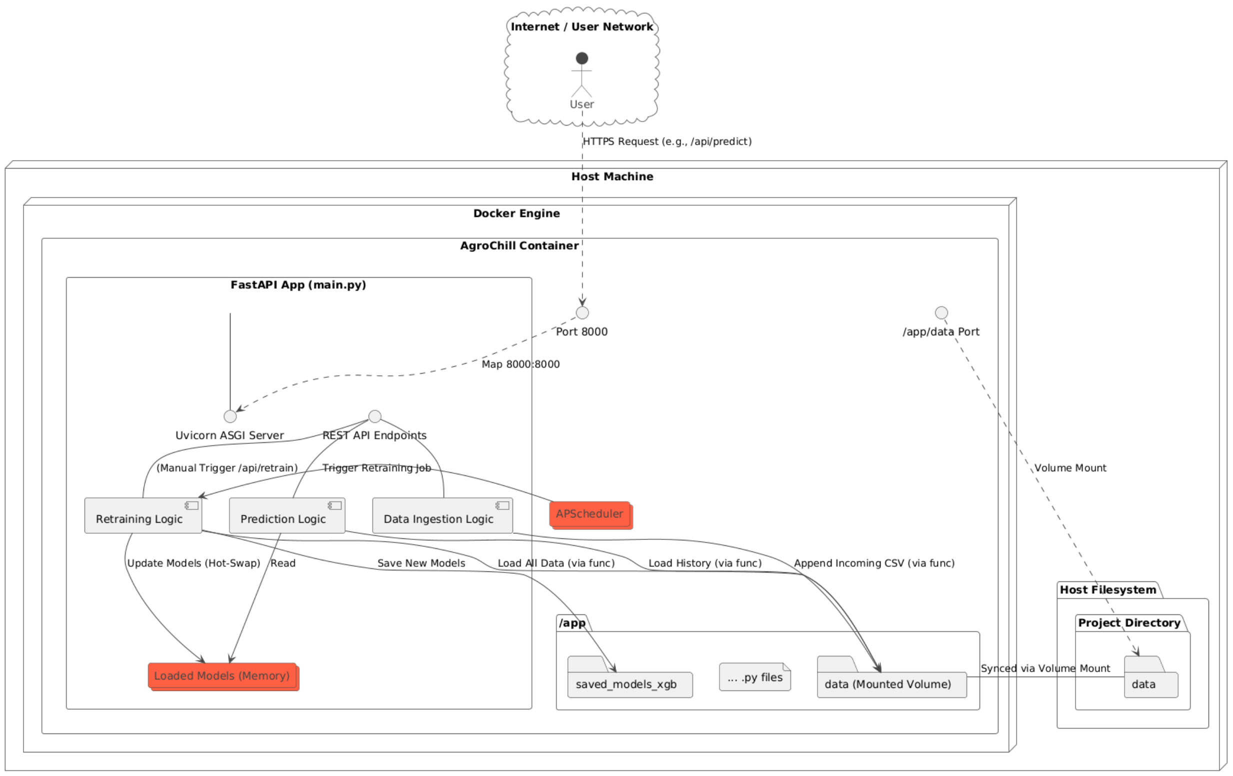
This section details the overall architecture, key components, and design choices made for the AgroChill forecasting system, evaluating its structure in terms of modularity, scalability, and deployment strategy, particularly concerning the competition's constraints and requirements.

5.1. Architectural Overview

The system is designed as a containerized web service built around a batch retraining pipeline. The core components are:

- **FastAPI Web Server**: Provides the external interface via a RESTful API for predictions and data ingestion. Handles incoming requests, validation, and orchestrates calls to the underlying logic. Runs using the Uvicorn ASGI server.
- **Prediction Logic**: Encapsulated within the `/api/predict` endpoint handler. Dynamically loads the complete dataset, generates features for the latest data point using historical context, and applies the pre-trained models for multi-horizon forecasting.
- **Data Ingestion Logic**: Handled by `/api/data/weather` and `/api/data/prices` endpoints. Validates incoming data and persists it to CSV files using file locking for concurrency control.
- **Machine Learning Models**: Four independently trained XGBoost models (within scikit-learn Pipelines including preprocessing), one for each forecast horizon (1w-4w). These are loaded into memory at application startup and updated ("hot-swapped") after successful retraining. Models are stored persistently as `.joblib` files.
- **Preprocessing Artifacts**: Fitted scikit-learn Scalers and Encoders (bundled within the saved Pipelines) and the list of features used during training (`features_list.joblib`) are stored persistently and loaded at startup/retraining.
- **Automated Retraining Module**: Integrated within the FastAPI application using `APScheduler`. This module contains the logic (adapted from standalone training scripts) to:
    - Load aggregated data (original + incoming).
    - Perform feature engineering.
    - Run hyperparameter tuning (Optuna).
    - Train new XGBoost models.
    - Save updated models/artifacts.
    - Update the models currently loaded in the API's memory.
- **Persistent Storage**: Uses the local filesystem (within the container, mapped via a Docker volume) to store:
    - Original datasets (`data/ directory`).
    - Incoming data (`data/incoming_*.csv`).
    - Saved model pipelines (`saved_models_xgb/`).
    - Saved feature list (`saved_models_xgb/`).

- **Docker Container**: Packages the FastAPI application, all Python dependencies, trained models, feature lists, and necessary system libraries into a single, portable image for deployment.



5.2. Modularity

The system exhibits reasonable modularity:

- **API Layer (main.py)**: Handles HTTP requests/responses, routing, input validation (Pydantic), and basic orchestration.
- **Core Logic (Functions within main.py)**: Data loading/aggregation (`load_and_preprocess_data_agg`), feature engineering (`engineer_features`), model training/tuning (`xgboost_objective`, `optimize_and_train_xgb_pipeline`), prediction feature generation (`get_latest_features_for_item`), and the retraining job (`run_retraining_job`) are encapsulated in distinct Python functions. This separation allows for easier testing and modification of specific components.
- **Model Persistence**: Models and preprocessing artifacts are saved independently (`.joblib` files), allowing them to be potentially generated or updated outside the main API flow if needed (e.g., using the standalone training scripts).
- **Configuration**: Key paths and parameters (like retraining interval, Optuna settings) are defined as constants near the top of `main.py`, making them relatively easy to adjust.

5.3. Scalability

Scalability needs to be considered in terms of handling requests, data volume, and retraining load.

- **Request Handling (Horizontal Scalability):**
  - FastAPI (running on Uvicorn) is asynchronous and generally performs well under load.
  - The Dockerized nature allows for **horizontal scaling**. Multiple instances of the container could be run behind a load balancer to handle increased API traffic (prediction requests, data ingestion).
  - **Limitation**: The current file-based persistence for incoming data (`FileLock` on CSVs) would become a bottleneck under very high concurrent write load. A database would be necessary for true high-concurrency ingestion scalability. Similarly, the retraining job currently runs on a single instance; distributed training would be needed for massive datasets.
- **Prediction Latency:**
  - Loading all data dynamically for each prediction in `/api/predict` to ensure true rolling forecasts introduces latency compared to using a pre-loaded snapshot. This involves I/O and significant computation (feature engineering on the item's history).
  - XGBoost inference itself is generally fast, but the data loading and feature generation dominate prediction time in the current design.
  - **Optimization**: A dedicated Feature Store or pre-calculating features for the latest points periodically could drastically reduce prediction latency in a production scenario, but adds significant infrastructure complexity. For the competition scope, the current dynamic approach was chosen to strictly meet the rolling forecast requirement.
- **Retraining Load:**
  - Retraining the models (including Optuna tuning) on the entire aggregated dataset is computationally intensive and time-consuming (observed ~5 minutes in testing, could grow significantly as data accumulates).
  - The `APScheduler` runs this in the background, so it doesn't block API requests, but it consumes CPU/RAM resources while running.
  - **Scalability**: For much larger datasets, this full batch retraining would need to be moved to dedicated, potentially more powerful, offline machines or a distributed computing environment (e.g., Spark, Dask). The current approach is suitable for moderate data growth within the competition timeframe.
- **Memory Usage:** XGBoost models and Pandas DataFrames can consume significant RAM. Loading the full aggregated dataset during prediction and retraining requires careful monitoring against the RAM limit. Using efficient data types and strategic deletion of objects (`gc.collect()`) helps, but this remains a potential bottleneck for long-term data accumulation.

5.4. Deployment Strategy (Docker)

- **Containerization:** The entire application is packaged within a Docker image using the provided `Dockerfile`.
    - Uses a `python:3.9-slim` base image for smaller size.
    - Installs necessary system libraries (like XGBoost).
    - Installs required Python packages via `requirements.txt`.
    - Copies application code, trained models (`saved_models_xgb`), and data stubs.
    - Exposes port 8000.
    - Uses `uvicorn` as the entry point to run the FastAPI application.
- **Volume Mounting:** Crucial for operation. A host volume must be mounted to `/app/data` inside the container to:
    - Provide the original training datasets (read by retraining/prediction).
    - Persist incoming data saved by the API endpoints.
    - Allow the retraining job to read the incoming data.
- **Ease of Use:** Users can pull the image from Docker Hub (`melkor1/agrochill-app:v2.0`) and run it with a single docker run command.
- **Reproducibility:** Docker ensures a consistent environment, minimizing issues related to dependency conflicts or OS differences.

6. Business Insights & Recommendations

This section translates the technical findings from the data analysis and the capabilities of the developed forecasting system into actionable business insights and recommendations for Magnus Greenvale and the AgroChill operation. The goal is to leverage the "Freezer Gambit" effectively by making data-informed decisions.

6.1. Key Insights from Data Analysis

Our exploration of the historical weather and price data revealed several critical factors influencing the Agrovia produce market:

- **Significant Regional Price Variation**: There are consistent and noticeable differences in the typical price ranges and volatility for the same commodities across different regions. This implies that the "best market" to sell in isn't uniform and changes based on location. Insight: Simply selling locally might not yield the highest profit; targeting specific regions based on predicted price advantages is key.
- **Strong Price Autocorrelation (Momentum)**: The most reliable indicator of next week's price is often this week's price (and prices from the recent past). Prices exhibit inertia or momentum. Insight: Tracking recent price trends is crucial, and the model heavily relies on these lag features. Sudden, unpredictable market shocks are harder to forecast.
- **Pronounced Seasonality**: Both prices for specific commodities and key weather variables show clear annual cycles. Demand, supply, and growing conditions likely follow predictable yearly patterns. Insight: Pricing strategies should anticipate these seasonal highs and lows. The model incorporates time features (month, week) to capture this.
- **Weak Direct Weather-Price Link**: Current weather conditions (temperature, rainfall, humidity) show very weak direct linear correlation with the current market price. Insight: Don't expect immediate price jumps based solely on today's weather report. Weather's impact is likely more complex, affecting supply over weeks or months (captured indirectly by yield scores and lagged features) or influencing prices through non-linear mechanisms that the XGBoost model can learn.
- **Data Availability is Consistent**: The historical data volume is remarkably consistent across all regions. Insight: This increases confidence that the model's performance is likely to be relatively uniform across different markets, reducing the risk of relying on forecasts for less data-rich regions.

6.2. Leveraging the AgroChill Forecasting System

The developed system, particularly the recommended `v2.0` (XGBoost) version, provides Magnus with a powerful tool. Here's how to leverage it:

- **Strategic Market & Timing Decisions (Core Use Case):**
  - **Action:** Regularly query the /api/predict endpoint for key commodities across relevant target regions. Compare the predicted prices for Week 1 vs. Weeks 2, 3, and 4.

- **Recommendation:**
  - If **Week 1 predicted price is high** (relative to recent trends or seasonal norms) and future weeks show lower or similar prices: **Sell fresh immediately** at the next market day.
  - If **Week 1 predicted price is low**, but **Weeks 2-4 show a significant predicted increase**: Utilize AgroChill storage (the "Freezer Gambit") and plan logistics to sell in those future weeks in the region with the best predicted price.
  - **Consider Storage Costs**: Factor in the documented depreciation rate (e.g., 10% per week for tomatoes in the example) and storage costs against the predicted price gain to ensure freezing is profitable. A predicted 15% price increase in 2 weeks might be negated by 19% depreciation (if compounded).

- **Regional Arbitrage (Advanced Use):**
  - **Action:** Query the API for the same commodity across multiple accessible regions for the upcoming 1-4 weeks.
  - **Recommendation:** Identify regions with significantly higher predicted prices compared to the local market, even after accounting for transportation costs and potential spoilage/depreciation during transit. This could open opportunities for shipping produce to higher-value markets, enabled by the extended shelf-life from AgroChill.
- **Input for Planting/Harvesting Decisions (Longer Term):**
  - **Action:** Analyze historical forecast accuracy and seasonality patterns identified by the model (e.g., via seasonal decomposition plots or analyzing long-term prediction trends).
  - **Recommendation:** While the current model predicts only 4 weeks out, understanding the typical seasonal price peaks (derived from EDA like for specific high-value crops can help inform planting schedules months in advance to potentially align harvest times with historically high-price periods.

6.3. Actionable Improvement Suggestions & Future Work

While the current system provides valuable forecasts, further enhancements could increase its utility and accuracy:

- **Enhance Prediction Feature Generation:**
  - **Problem**: The current `/api/predict` uses simplified features based only on the last base data point.
  - **Suggestion**: Modify the prediction endpoint logic (`get_latest_features_for_item`) to load recent history (including incoming data) for the specific item being predicted. Re-run the exact feature engineering steps to create the input vector for the model. This will significantly improve the accuracy of real-time rolling forecasts.

- **Incorporate Storage Costs & Depreciation:**
    - **Problem**: The system predicts fresh prices; the profitability calculation considering storage costs and item-specific depreciation rates is external.
    - **Suggestion**: Extend the API or build a supplementary tool/dashboard where Magnus can input depreciation rates per commodity and storage costs. This tool could take the API's fresh price forecast and output a predicted net profit for selling in week 1 vs. week 2, 3, 4 after accounting for these factors.
- **Refine Retraining Strategy:**
    - **Problem**: Full batch retraining on all data can become slow as incoming data grows.
    - **Suggestion**: Explore more advanced retraining strategies:
        - **Incremental Training**: Investigate if XGBoost (or LightGBM) can be updated incrementally with new batches without retraining from scratch (model-dependent feature).
        - **Windowed Retraining**: Train only on a sliding window of the most recent data (e.g., last 2-3 years) if older data becomes less relevant.
        - **Offline Cluster**: For large scale, move the retraining job off the API server to a dedicated, potentially more powerful, offline system.
- **Add Explainability (XAI):**
    - **Problem**: Understanding why the model predicts a certain price can increase trust and utility.
    - **Suggestion**: Integrate tools like SHAP (SHapley Additive exPlanations) into the prediction or retraining pipeline. This could generate plots or reports showing which features (e.g., last week's price, recent rainfall) contributed most to a specific forecast, providing deeper insights than just overall feature importance.
- **Improve Data Ingestion Robustness**:
    - **Problem**: Relying on CSV appends can be fragile at scale.
    - **Suggestion**: Migrate data persistence from flat files to a simple database (like SQLite within the container or an external database service) for better querying, atomicity, and error handling.