A Framework for Trade Verification in Power and Gas Markets

Mathis Laurent
EDF UK – Curve Desk

July 28, 2025

Abstract

This paper presents a tool-based framework developed for EDF UK's Curve Desk to facilitate daily, weekly, and monthly trade analysis across power and gas markets. The framework focuses on trade flagging, anomaly detection, mispriced trades flagging and backtesting, supporting enhanced verification of trading performance.

1 Introduction

In the context of high-frequency energy trading at EDF UK's Curve Desk, the need for a robust and precise verification tool is essential. With over 5,000 trades executed daily by power traders and approximately 200 by gas traders, the likelihood of typographical or mispricing errors is non-negligible. The primary objective of this project is to design and implement an end-to-end tool that identifies such anomalies reliably by comparing internal trade data with EDFT reference prices.

This tool operates in two main stages. First, it ingests raw trade logs exported from the ALigne ETRM system either on a daily, weekly, or monthly aggregated basis focusing on power and gas transactions. Each trade is indexed by a unique Trade ID and includes key parameters such as volume, traded price, hedge value, and delivery period. The system initially filters out irrelevant entries, including prompt trades and trades specifically flagged for exclusion by the trading team. It then proceeds to clean and refurbish the dataset: standardizing date formats, removing redundant columns, and ensuring delivery windows are formatted correctly for comparison.

Each trade is then labeled using custom logic that assigns a `ProductLabel` (e.g., month, quarter, season) and a `LoadShape` (e.g., baseload, peakload, offpeak). This standardization enables apples-to-apples comparisons between traded prices and internal valuations. Once this preprocessing is complete, the system queries a dynamic Snowflake database to retrieve EDFT internal price curves for the exact same date and delivery specifications. These prices are merged with the cleaned dataset, allowing for side-by-side comparison between traded and reference prices.

The output of this first stage is stored locally in `.csv` format, ready for further inspection and flagging. In a future version of the tool, this pipeline could be automated end-to-end from Aligne export to Snowflake ingestion to visual dashboard output.

The second stage of the process consists of a statistical flagging model. It takes the cleaned and enriched dataset and applies multiple anomaly detection methods to identify potential mispriced trades. These methods include standard Z-score thresholds, robust statistical filters based on median absolute deviation (MAD), and machine learning-based techniques such as Isolation Forest. Any trade where the price deviation exceeds method-specific thresholds is flagged.

Finally, the flagged trades are aggregated and visualized through a Tableau dashboard, where users can filter results by product type, load shape, trade date, and more. The dashboard highlights repetitive mispricing patterns across certain trade types or delivery periods. Ultimately, traders can isolate trades with significant price deviations, often caused by manual input errors—and take corrective actions on their orders based on Trade ID.

This framework provides a scalable, reliable, and trader-friendly interface for validating thousands of trades daily, enhancing overall market accuracy and minimizing operational risk.

2 System Architecture and Tools Used

This section outlines the technological ecosystem behind the trade verification framework, from data access to analysis and reporting. The entire process integrates external databases, market tools, and programming libraries to enable a scalable and automated analysis of energy trades.

2.1 Industry Tools and Platforms

- **Snowflake**: Snowflake is the cloud-based data warehouse used to access internal pricing data, particularly EDFT daily prices. Access requires specific user roles, typically granted to members of the Curve Desk or analytics teams. The connection to Snowflake is established via secure authentication using the `snowflake.connector` module in Python.
- **Visual Studio Code (VSC) interconnection**: It interfaces with Snowflake using Python scripts and connector-based authentication.
- **ALigne (ETRM Tool)**: ALigne is the enterprise trading and risk management system used to record all daily power and gas trades. These trades are exported as `.csv` files for external analysis.
- **Excel and .CSV**: Trade logs and filtered data outputs are handled in `.csv` format, allowing compatibility with Excel and Python's `pandas` module.
- **SQL Queries**: SQL is used to extract relevant EDFT pricing data from Snowflake. These queries are embedded directly within the Python scripts to automate data loading and filtering. Additionally, small SQL scripts can be written manually for one-off trade validations.

2.2 Python Libraries and Functions

The following Python libraries are used throughout the project for data manipulation, analysis, filtering, anomaly detection, and output formatting:

- `pandas`: For reading, cleaning, transforming, and exporting `.csv` data files.
- `numpy`: For numerical operations and array handling.
- `datetime` and `pandas.tseries.offsets`: For time-based filtering, date shifting, and delivery window validation.
- `matplotlib` and `seaborn`: For visualization, including distribution plots, boxplots, and diagnostics.
- `os`: For accessing the local file system, validating paths, and saving output files.
- `snowflake.connector`: For securely querying Snowflake to retrieve EDFT pricing data.
- `openpyxl`: For enhanced Excel operations (if needed), including formatting and color highlighting.

- `statsmodels.robust.scale`: Specifically for Median Absolute Deviation (MAD)-based outlier detection.
- `sklearn.ensemble.IsolationForest`: For machine learning-based anomaly detection.
- `PatternFill` and `Font`: Formatting tools from `openpyxl.styles` for Excel customization.

These libraries enable complex workflows including:

- Trade data ingestion and preprocessing
- SQL-based price matching from Snowflake
- Trade labeling based on delivery characteristics
- Anomaly and outlier flagging via multiple statistical methods
- Export of verified and flagged trades to Tableau-compatible files

The script is designed to be modular and extendable, with options for future integration into daily batch processes or dashboarding systems.

3 User Guide and Tool Execution

3.1 Inputs and Setup

- **Input Date Format:** The user is prompted for a trade date in the format `DDMMMYY` (e.g., `25JUN25`).
- **File Structure:** Input trade logs must be placed in the directory `R:\Project 1\Output Trades (Tableau)`.
- **Output Path:** Final cleaned and flagged trade data is uploaded to Snowflake under the following path: `DB_WMS_PRD` → `MDR_SANDBOX` → `Tables` → `POWER_GAS_TRADE_FLAG`.
- **Dependencies:** The system requires packages such as `pandas`, `snowflake.connector`, `openpyxl`, `sklearn`, and `statsmodels`.

3.2 Execution Flow

1. **Download trade logs manually from FIS Aligne** and save them to the designated local folder (e.g., `R:\ZAINET_CLIENT\ODBS`).
2. **Run the labelling and filtering script** (`flag_trade_labeller.py`).
   (a) Enter the start and end trade dates when prompted (format: `DDMMMYY`).
   (b) The script will load local trade files, apply validation, compute internal metrics, label delivery periods and load shapes, fetch EDFT prices from Snowflake, and merge pricing.
   (c) Filtered and enriched outputs are saved as CSV files to `R:\Project 1\Trades Labelling & Filtering`.
3. **Run the flagging script** (`flag_trade_outliers.py`).
   (a) The script reads the labelled CSVs, applies Z-score, Robust Z-score, and Isolation Forest methods to identify anomalous trades.
   (b) Flagged results are uploaded to the Snowflake table `POWER_GAS_TRADE_FLAG`.
4. **Access Tableau dashboards** connected to the Snowflake output table to review flagged trades, aggregated PnL, and statistical anomalies.

3.3 Understanding the Trade Date Logic in the Script

In this script, the user is required to **input the trade date**,that is, the day on which trades were executed. This input must be a valid **business day** (i.e., Monday to Friday).

For example, if the user inputs:

```
04JUL25
```

which corresponds to a Friday, the code will automatically **add 1 calendar day** to this input, resulting in:

```
05JUL25
```

This incremented date represents the report date, the day on which trade files are generated and published.

Here's how the logic works step-by-step:

- The **user inputs the trade date**, e.g., `04JUL25`.
- The code then computes the corresponding report date by **adding one calendar day** to the trade date. This is used to **fetch the correct trade file**, such as:

  `MDR_ELEC_TRADES_05JUL25.csv`

- Despite fetching the file using the incremented date, the **original input date** is still used throughout the script to:
  - Query EDFT market prices for that trade day.
  - Filter trades based on their actual execution date.
  - Label each trade with the correct week, month, or quarter.
- In essence, the input date is the anchor of the analysis. The increment is only used for file access, while the rest of the pipeline operates relative to the trade date itself.

To summarize: *We input the trade date, and the code automatically computes the report date to fetch the corresponding trade file. The trade date remains the reference point for all price comparisons, labelling, and analysis.*

4 Methodology

This section outlines the complete methodology used for processing, validating, and preparing power and gas trade datasets for anomaly detection and dashboard visualization. The pipeline consists of two principal components: the Trade Filter, Labelling, and Merging Module, and the Statistical Flagging Tool. Each step is implemented using modular and modifiable Python code, allowing users to adapt parameters depending on their specific workflow needs.

4.1 Trade Filter, Labeling, and Merging Module

User Interface: The system begins with a user prompt requesting a trade date in the format `DDMMMYY`" (e.g., `20JUN25`"). This date is used to locate the corresponding power and gas trade files and to fetch reference prices from the EDFT database. The Snowflake connection is established via external browser authentication, requiring a valid user account and access credentials.

Data Ingestion: Trade files for power and gas are loaded using the `pandas.read_csv()` method. Corresponding EDFT reference prices are fetched using SQL queries executed through the `snowflake.connector` library.

Preprocessing and Cleaning: The tool checks for invalid or prompt trades based on the following:

- Zero Volume or Hedge Value (excluded).
- Trades with Delivery Date more than 2 days before the selected trade date.
- Prompt trades where Volume equals Forward Valuation and `DaysToDelivery` < 2.

Price Computation: Traded and internal prices are computed per MWh. Metrics like price differences (`PriceDiff`, `AbsPriceDiff`), and `WeightedImpact_£` are calculated for further analysis.

Labelling: Each trade is labeled with:

- `ProductLabel` — such as `202406`, `2024Q3`, or seasonal labels `2024WIN`, `2024SUM`.
- `LoadShape` — using start and end time logic: `BASELOAD`, `PEAKLOAD`, or `OFFPEAK`.
- `WeekLabel`, `MonthLabel`, `QuarterLabel` — derived from delivery period.

Merging EDFT Prices: An exact match is attempted on `ProductLabel` and `LoadShape`. If unavailable, the system defaults to EDFT `BASELOAD` prices. Final columns include `EDFT_Price`, `EDFT_Diff`, and `EDFT_Flag`.

Export: After cleaning and enrichment, power and gas datasets are exported to CSV files with a subset of relevant columns for Tableau integration. Directory paths and selected columns can be modified directly in the export block.

Modifiable Parameters:

- Thresholds for `AbsPriceDiff` (currently set to `> 3`).
- Minimum volume for flagging (set to `> 10000`).
- Threshold for EDFT deviation (fixed at 2).
- Export file location and date formatting.

Summary Diagnostics: Trade counts, suspicious trade ratios, prompt trade flags, and missing EDFT prices are reported at the end for both power and gas datasets.

4.2 Flagging Tool

This part of the system applies statistical methods to detect outliers among the cleaned and labeled trade data. The flagged trades are later visualized in a dashboard to assist traders in identifying mispriced or erroneously recorded transactions. The description and code for this module are presented in the following section.

The second phase of the pipeline applies multiple statistical methods to detect potential anomalies or mispriced trades based on deviations between traded and internal valuations (`EDFT_Diff`). Each trade is flagged based on three complementary approaches:

4.3 Z-Score Method (Standard Score)

Purpose: Identifies individual trades with large deviations from the mean. Best suited for detecting specific mispriced trades per `TradeID`.

Formula:

$$Z = \frac{x - \mu}{\sigma}$$

Where:

- x is the trade's price difference
- μ is the mean of all price differences for the trade day or week (depending on User selection)
- σ is the standard deviation:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

where:

  - x_i is the price difference (`EDFT_Diff`) of trade i,
  - n is the total number of trades considered.

Flagging Rule: $|Z| > 2.5$

Interpretation:

- Sensitive to outliers and assumes normality
- Effective for pinpointing strong anomalies on a per-trade basis

4.4 Robust Z-Score Method (Median + MAD)

Purpose: More stable under non-normal distributions. Reduces sensitivity to extreme values by relying on medians.

Formula:

$$RZ = \frac{x - \text{median}(x)}{1.4826 \cdot \text{MAD}}$$

Where:

- $\text{MAD} = \text{median}(|x - \text{median}(x)|)$
- 1.4826 is a constant to normalize for Gaussian-like behavior

Flagging Rule: $|RZ| > 2.5$

Interpretation:

- Less influenced by outliers
- Suitable for distributions with heavy tails or skewed data

4.5 Isolation Forest (Machine Learning)

Purpose: An unsupervised anomaly detection algorithm using tree-based isolation. Detects groups of abnormal trades by structure or rarity.

Mechanism:

- Builds random decision trees and isolates samples
- Outliers are isolated faster and scored lower
- Trade is flagged if model prediction returns `-1`

Interpretation:

- Captures unusual trade patterns or clusters
- Stronger when applied to aggregated datasets

In the implementation:

- We use `IsolationForest` from `sklearn.ensemble`.

- We set a contamination rate of 5% to assume that roughly 5% of the trades could be anomalous.
- The model assigns each trade a label: `-1` for anomalous, `1` for normal.
- A column `Flag_IForest` is created where trades labeled as `-1` are flagged.

This method is particularly useful for detecting clustered or patterned anomalies that may not be obvious using statistical thresholds like Z-score. However, it can be more sensitive and may capture borderline or slightly irregular trades as potential anomalies.

Outcome and data overview for Power trades:

Figure 1: Overview of trade flagging agglomeration and pattern over a week of Power trades. All flagged trades here include ML method, Robust Z score and Z score flagged trades.

Outcome and data overview for Power trades:

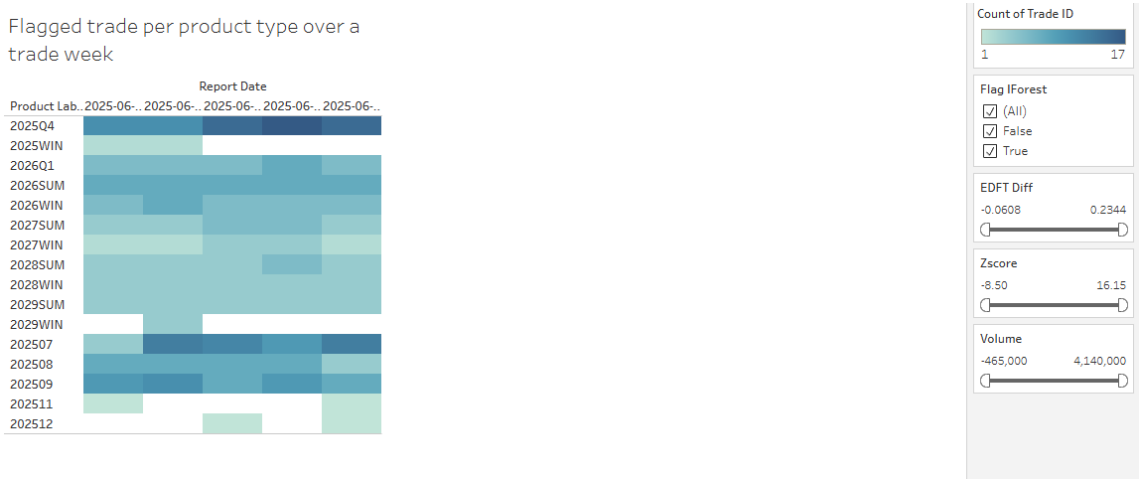

Figure 2: Overview of trade flagging agglomeration and pattern over a week of Gas trades. All flagged trades here include ML method, Robust Z score and Z score flagged trades.

4.6 Flag Aggregation and Highlighting

Each method returns a Boolean flag per trade. A total `Flag_Count` column aggregates the number of methods that flagged a given trade (range 0 to 3). If at least two methods flag the same trade, it is considered a strong anomaly and highlighted in red in the Excel output.

4.7 Comparison Summary

Method	Robustness	Sensitivity	Best Use Case
Z-Score	Low	Low	Detecting individual trade-level mispricings in normally distributed datasets
Robust Z-Score	High	Medium	General-purpose flagging that is more resilient to outliers and skewed data
Isolation Forest	Adaptive	High	Detecting group anomalies or pattern-based anomalies (ideal for clustering-based heatmaps)

Table 1: Comparison of Trade Flagging Techniques

Flagging overview:

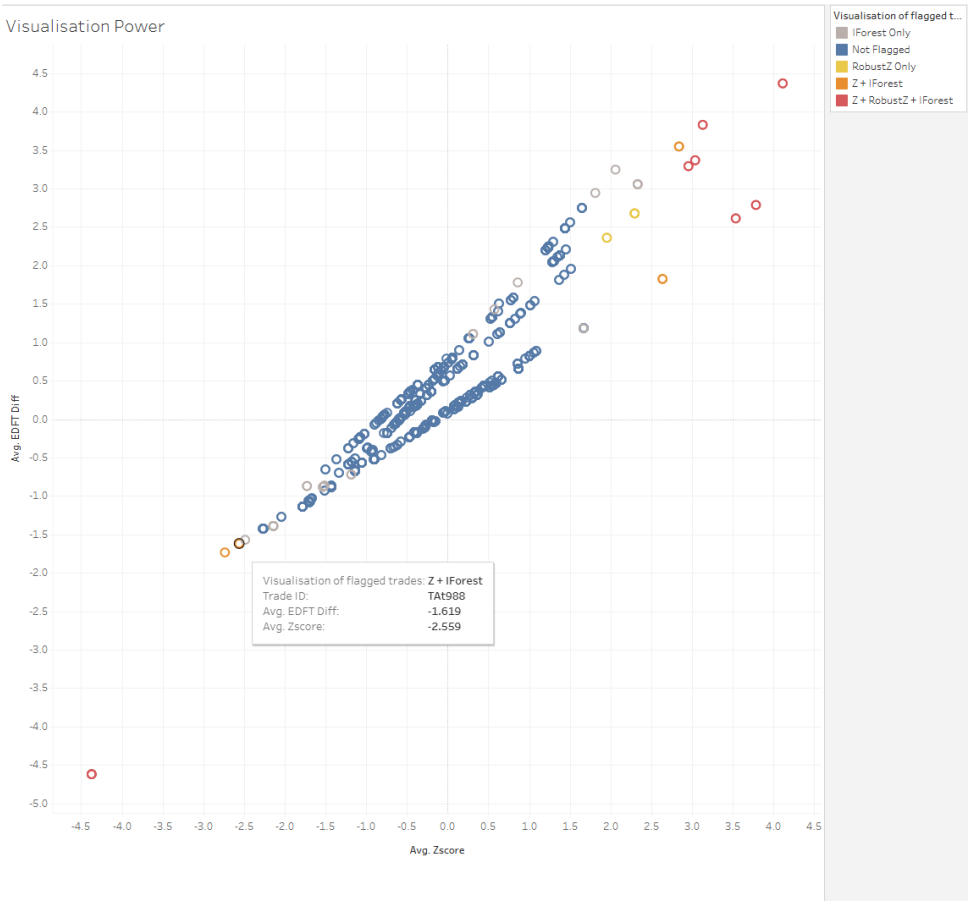

Figure 3: Overview of price data per trade. Colours present different flagging methods.

5 Data Structure and Sources

5.1 Trade Data Sources

The system retrieves two main categories of trade data on a daily basis:

- **Power Trades:** Exported from the Aligne ETRM system into daily CSV files prefixed with `MDR_ELEC_TRADES_{date}.csv`.
- **Gas Trades:** Similarly exported with filenames like `MDR_GAS_TRADES_{date}.csv`.

These files are stored locally and accessed using Python's `os.path` and `pandas.read_csv()` methods. The user inputs a trade date (`DDMMMYY` format), which is automatically incremented by one business day to locate the report file (as Aligne reports for `D` are saved as `D+1`).

5.2 Reference Price Data (EDFT)

Reference prices for validation are fetched dynamically from Snowflake using SQL queries through the `snowflake.connector` module. The system connects to the staging schema of the EDFT price publication database and queries two relevant tables:

- `POWER` table, filtering by `MARKET = 'UNITED KINGDOM'` and `PRICE_TYPE = 'MID'`.
- `GAS` table, filtering by `PUBLICATION_DATE`.

The relevant fields extracted include:

- `ProductLabel` – standardized delivery product label (e.g., 2024Q1, 202406, 20240603).
- `LoadShape` – trade profile such as BASELOAD, PEAKLOAD, or OFFPEAK.
- `EDFT_Price` – reference market price for validation.

Once fetched, the results are stored in two DataFrames: `df_edft` for power and `df_edft_gas` for gas. All column values are cleaned and standardized to ensure compatibility during merging.

```
# Connect to Snowflake
conn = snowflake.connector.connect(
    user='mathis.laurent@edfenergy.com',
    account='YSQZARO-WQ16105',
    warehouse='WH_PERSONA_WMS_PRD',
    database='FLK_DUB_DB_DATALAKE_PRD',
    schema='STAGING_WMS_EDFT',
    authenticator='externalbrowser'
)

# Query to fetch EDFT power prices
query_power = f"""
SELECT
    PRODUCT, LOAD_SHAPE, PRICE
FROM POWER
```

```
WHERE PUBLICATION_DATE = DATE '{edft_reference_date_str}'
  AND MARKET = 'UNITED KINGDOM'
  AND PRICE_TYPE = 'MID';
"""

cursor = conn.cursor()
cursor.execute(query_power)
df_edft = pd.DataFrame(cursor.fetchall(), columns=[col[0] for col in
↪  cursor.description])
df_edft.columns = ['ProductLabel', 'LoadShape', 'EDFT_Price']
df_edft['EDFT_Price'] = df_edft['EDFT_Price'].astype(float)
```

5.3 Data Indexing and Merging

Trade File Parsing

The loaded CSV files contain delivery start/end dates and times that are parsed and normalized into datetime and 24-hour string formats. Trade-level details are cleaned and mapped to unified column names such as `TradeID`, `TradeDate`, `StartDate`, `EndDate`, `StartTime`, `EndTime`, `Volume`, `HedgePrice`, and `ForwardValuation`.

Initial Filtering

Invalid trades are filtered out using the following logic:

- Zero-volume or zero-value trades are dropped.
- Trades where `Volume == ForwardValuation` and time to delivery is less than 2 days are flagged as *prompt* and excluded.

Label Assignment and Load Shape Inference

Each trade is labeled with a `ProductLabel`, `MonthLabel`, `QuarterLabel`, and `LoadShape` based on its delivery window and time. This labeling is essential for aligning trade data with the EDFT price structure.

Reference Price Matching

The merging process occurs in two stages:

1. **Primary Join:** Merges `df_power` and `df_edft` on exact `ProductLabel` + `LoadShape`.
2. **Fallback Join:** For unmatched rows, the tool defaults to EDFT prices for `BASELOAD` only, using the same `ProductLabel`.

The final EDFT price is computed as:

- `EDFT_Price` = `EDFT_Price_Matched` if available.
- Otherwise, fallback to `EDFT_Price_Fallback`.

A similar process is applied to gas trades, but only one match on `ProductLabel` is required.

Final Output

Once cleaned and matched, the power and gas DataFrames are saved locally as CSV files in a predefined output directory for further processing by the anomaly flagging tool and dashboard visualization pipeline.

6 Dashboard and Visualizations

The final dashboard is built using Tableau Online, which connects directly to the Snowflake table `DB_WMS_PRD.MDR_SANDBOX.POWER_GAS_TRADE_FLAG`. This table contains all flagged and enriched trades resulting from the labelling and anomaly detection pipeline.

The dashboard provides both granular and aggregated insights into the trade quality, PnL contributions, and anomaly scores across time and product types. Below are the key visualizations, each designed to serve a specific analytical purpose.

6.1 1. PnL per Energy Type vs. Trade Date

This line chart displays the aggregated profit or loss (`WeightedImpact_£`) by `TradeDate`, grouped by `EnergyType` (i.e., Power or Gas).

Purpose: Quickly identifies periods of high financial risk or unusual profitability, and distinguishes temporal performance patterns across energy products.

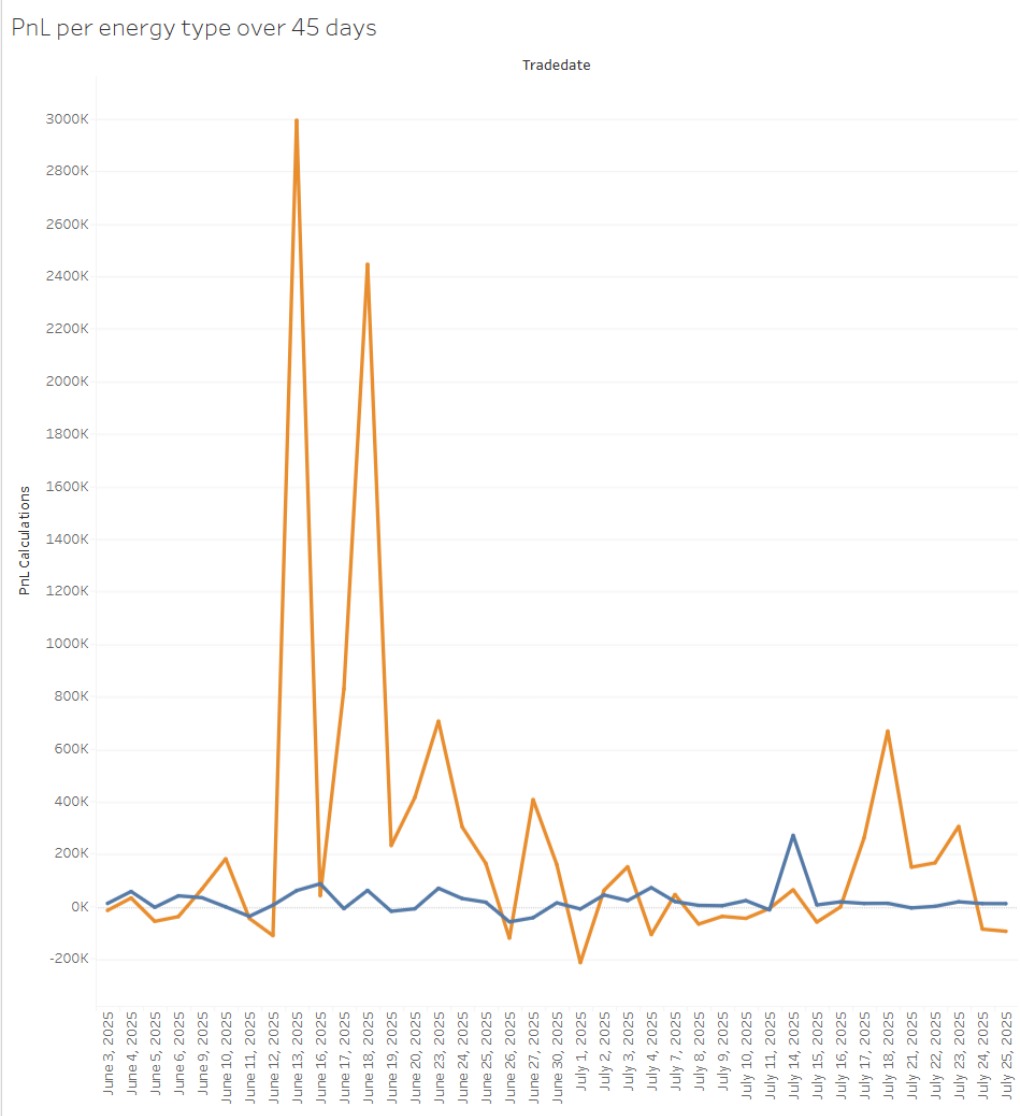

Figure 4: PnL per Energy Type over Time

Market Context Behind PnL Spikes

The two significant spikes observed in the PnL dashboard correspond to the trading days of **13th June** and **18th June**, respectively. These dates align with a major geopolitical event in the Middle East: on June 13th, Israel launched a bombing campaign targeting Iranian nuclear facilities. In retaliation, Iran responded on June 18th with a full-day barrage of ballistic missile strikes.

These escalating military actions raised serious concerns among global commodities investors, particularly around the potential closure or disruption of the **Strait of Hormuz**—a critical maritime corridor through which a significant share of the world's oil and liquefied natural gas (LNG) is transported.

Fears of restricted tanker movement and LNG flow from the Gulf region led to immediate surges in oil prices, which in turn impacted gas markets due to their strong correlation. Given the tight coupling between gas and power markets in Europe, this resulted in a substantial upward movement in power prices as well. Additionally, traders anticipated increased demand for alternative gas supplies, such as U.S. LNG and Norwegian pipeline gas, further contributing to price volatility.

The spike in power PnL during these days thus reflects not just market reaction to geopolitical tension, but also the broader systemic interdependence between oil, gas, and electricity markets under stress conditions.

Later on, the spikes and volatility around the **18th July** and **23rd July** was mainly due to the following:

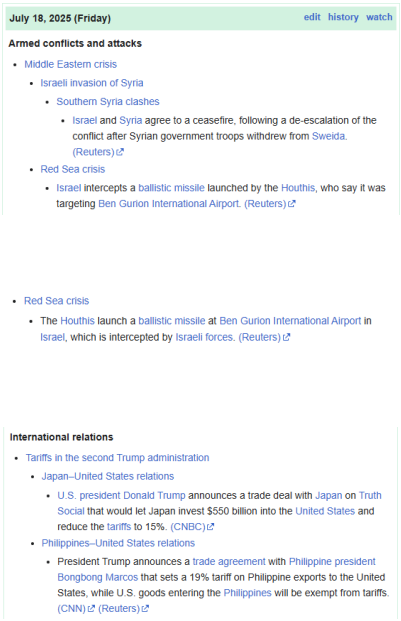

Figure 5: News source regrouped on wikipedia

6.2 2. PnL per Energy Type vs. Product Label

This bar chart aggregates PnL by `ProductLabel` (e.g., `2024Q4`, `20250701`), grouped by `EnergyType`.

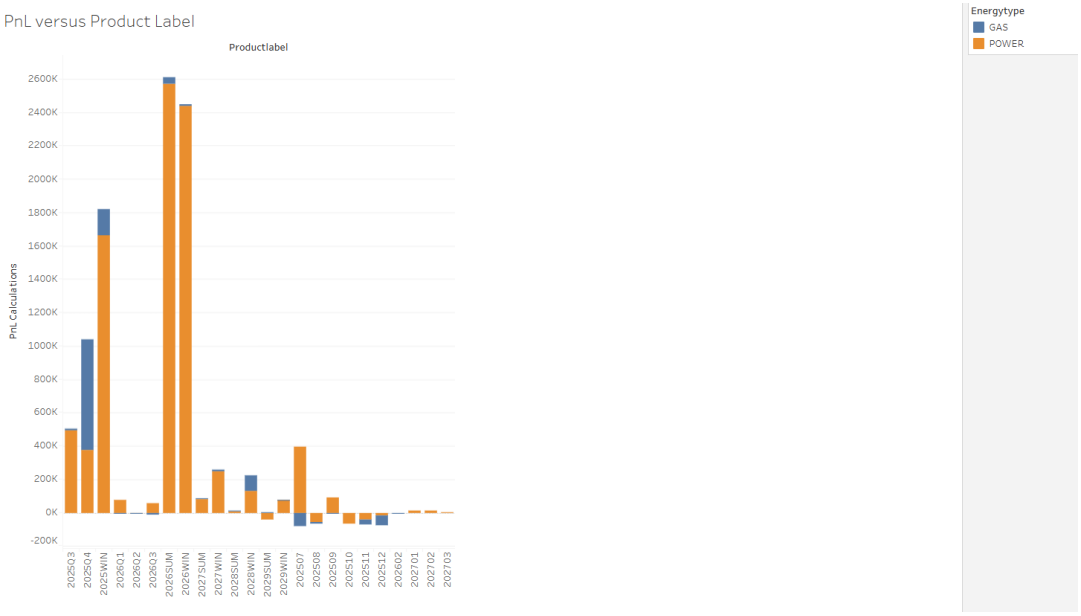

Figure 6: PnL by Product Label and Energy Type

Purpose: Highlights which delivery periods are most profitable or risky, helping to identify structural pricing issues with specific products.

6.3 3. Flagged Trade Count per Product Label

This chart shows the number of trades flagged (Flag Count ≥ 1) per `ProductLabel`.

Purpose: Detects delivery products with frequent anomalies, allowing investigation into mispricing or trade entry issues.

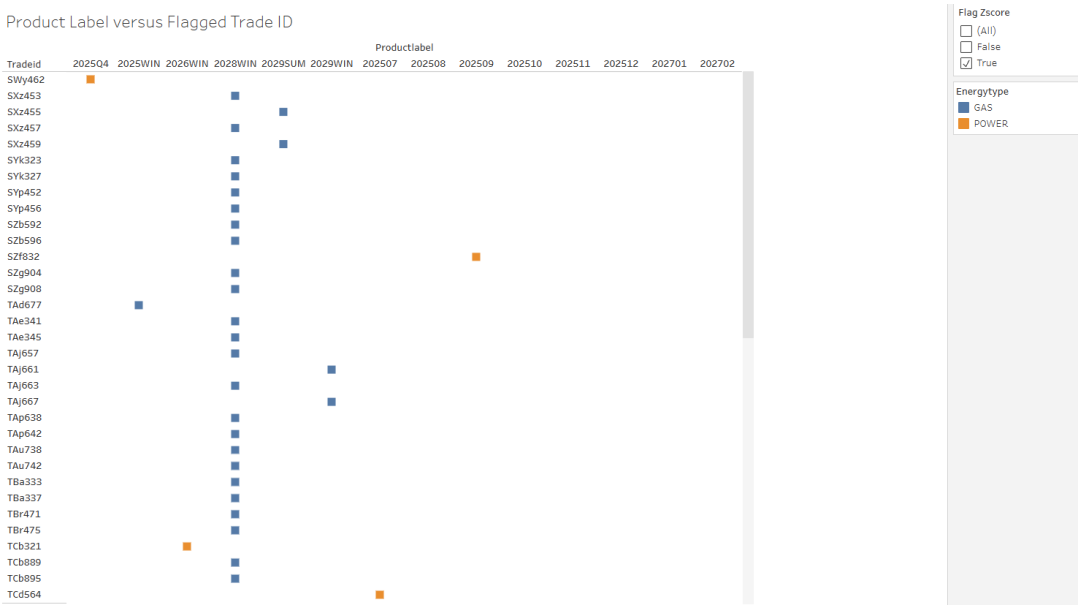

Figure 7: Number of Flagged Trades per Product Label

6.4 4. Trade Count per Energy Type vs. Product Label

This count plot groups the total number of trades by `ProductLabel` and segments by `EnergyType`.

Purpose: Reveals trading intensity per product and detects if certain delivery periods are overly or under-traded.

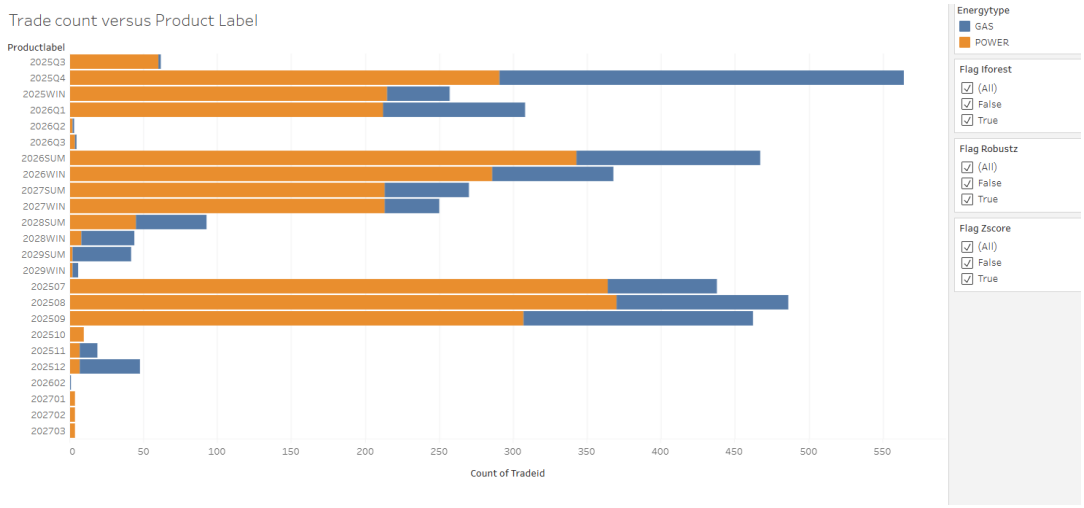

Figure 8: Total Trades by Product Label and Energy Type

6.5 5. Heatmap: Energy Type vs. Product Label with Z-Score Intensity

This heatmap uses `AVG(Zscore)` as the color intensity, with `EnergyType` and `ProductLabel` on the axes.

Purpose: Visualizes the average deviation from EDFT pricing benchmarks, helping identify systemic pricing discrepancies across energy and product types.

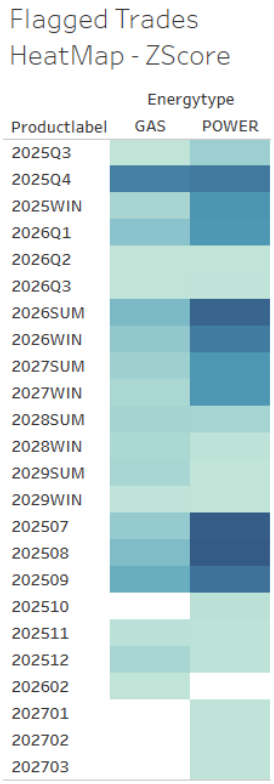

Figure 9: Heatmap of Z-Score Intensity by Product and Energy Type

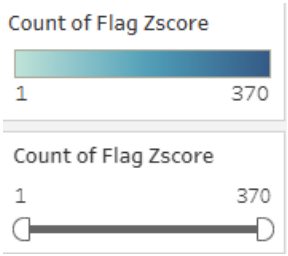

Figure 10: Parameters

6.6 6. Global Trade Overview Dashboard

This dashboard combines multiple metrics over time:

- **PnL per TradeDate**
- **Percentage of Flagged Trades**
- **Trade Count**
- **Average Z-Score**
- **Average Flag Count**

Purpose: Provides a holistic, time-series driven view of trade performance, statistical anomaly severity, and overall operational health.

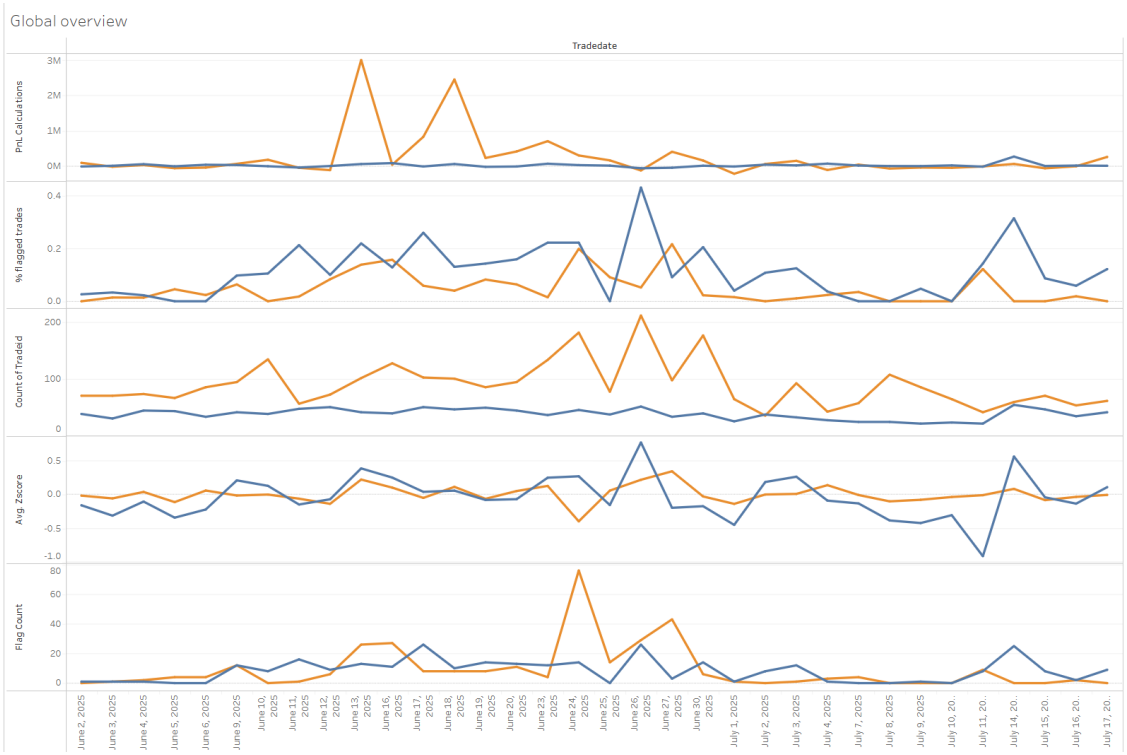

Figure 11: Global Trade Overview Dashboard

- Overview and comparison of different metrics and parameters over the same period (Trade date: 01 July - 17 July 2025). Useful to simultaneously visualise correlation between PnL, ratio of flagged trades, count of trades, Z score value...

7 Warning and potential issues

Even though the tool is relatively reliable and effective it has some potential inefficiency in terms of matching if ran and not maintained year round. Other issues regarding parameters and flagging can be checked in the conclusion below in **section 7**.

Attention

This code does not take the transition from Summer Time to Winter Time into consideration.
Potential issues that may arise:

- **Incorrect LoadShape classification** — trades near DST transitions may be labeled as PEAKLOAD instead of BASELOAD, or vice versa (this does not matter if we are ignoring prompt trades).
- **Misaligned delivery windows** — especially for trades spanning 05:00–05:00, which may span 23 or 25 real hours.
- **ProductLabel assignment errors** — date-based or intraday labels may be mismatched when time offsets shift.
- **Incorrect PnL impact analysis** — due to mismatches between local time and EDFT price publication times (which may use UTC).
- **Data merging issues** — especially when EDFT prices are timestamped in UTC and trade data is not timezone-aware.

Potential code fix to add:
- Add this at line 211 at: " Convert dates to datetime for Power"

```
from zoneinfo import ZoneInfo  # already mentioned

tz_london = ZoneInfo("Europe/London")

# Apply timezone awareness to delivery windows
df_power['StartDate'] = df_power['StartDate'].dt.tz_localize(
    tz_london, ambiguous='NaT', nonexistent='NaT'
)
df_power['EndDate'] = df_power['EndDate'].dt.tz_localize(
    tz_london, ambiguous='NaT', nonexistent='NaT'
)
```

- Later on add this code snippet just after the previous one

```
def combine_datetime(row):
    try:
        date = row['StartDate'].date()  # Already tz-aware
        time = datetime.strptime(row['StartTime'], '%H:%M').time()
        dt = datetime.combine(date, time)
        return dt.replace(tzinfo=ZoneInfo('Europe/London'))
    except:
```

```
        return pd.NaT

df_power['StartDateTime'] = df_power.apply(combine_datetime, axis=1)
```

- And finally the last code snippet to add just after the previous code:

```
df_power['StartDateTime_UTC'] =
↪  df_power['StartDateTime'].dt.tz_convert('UTC')
```

Attention

Leap year (bissextile) handling may not be fully covered — especially for February 28th–29th edge cases.
Potential issues to watch for:

- **Incorrect range detection** — logic assuming February has only 28 days may skip valid trades or dates in leap years.
- **Delivery period misclassification** — trades spanning February might not be labeled correctly by month or quarter.
- **Incorrect end-of-month detection** — functions like `detect_full_month()` may fail if not leap-year aware.
- **Silent errors in time series offsets** — operations using `pd.Timedelta` or `BDay` might skip over February 29 if not explicitly handled.

8 Conclusion and Next Steps

This tool provides a **robust and scalable framework** for validating power and gas trades through statistical and a machine learning-driven anomaly detection filter. Its modular architecture enables seamless ingestion, transformation, and flagging of trading data—making it an effective daily control system to reduce pricing errors and support trader decisions at EDF UK's Curve Desk. Parameters are quite modular within the code. Parameters to label by shape load or porduct label, parameters for pre-flagging and finally to final flagging can easily be removed or modified.

The use of **multiple flagging techniques**—standard Z-Score, Robust Z-Score, and Isolation Forest—ensures that the tool captures both **individual trade outliers** and **emerging patterns of mispricing** across load shapes and product periods. This diversity in detection logic allows the system to adapt to varying market conditions and datasets with skewed or non-normal distributions.

However, a few **considerations and potential limitations** should be acknowledged:

- **False positives or borderline trades** flagged by Isolation Forest may require human validation, particularly in high-volume or irregular trading weeks. Indeed Robust Z and isolation Forest flagging method are quite vulnerable and might flag un-problematic trades. These flagging filters are more here to identify pattern in terms of intensity for the flagged trades.
- **Timing mismatches** or **labeling discrepancies** between EDFT price publications and trade logs could introduce mismatches that must be handled via fallback logic or manual overrides.
- The current tool requires **manual execution steps**, including date input, local file management, and Snowflake authentication, which could introduce operational delays or inconsistencies.

8.1 Model Calibration and Feedback

To enhance accuracy over time, a validation loop could be established:

- Store trader feedback on each flagged trade (confirmed vs false positive).
- Track model precision, recall, and update flagging thresholds or contamination rate accordingly.
- Use a Snowflake-hosted feedback table for central accessibility.

8.2 Future Work and Automation Pipeline

To increase the efficiency and maintainability of the tool, the following improvements are proposed:

- **ETRM Integration**: Automate the end-to-end pipeline by directly linking with the ALigne ETRM database instead of relying on CSV exports. This would remove the need for local file handling and improve data freshness.
- **Snowflake Write Access**: Deploy a dedicated Snowflake schema to **store historical enriched trade logs** and flagging results, indexed by trade date and product type. This would allow:

  - Instant fetches for analytics
  - Efficient backtesting over rolling windows
  - Scalable integration with Tableau or Python-based dashboards

- **Live Monitoring Interface**: Integrate a **real-time flagging and diagnostics dashboard** (e.g., using Streamlit or Power BI) that can pull from Snowflake directly, giving traders and risk managers a live view of suspicious trades as they are executed.

- **Performance Tracking and Model Calibration**: Over time, a feedback loop could be introduced to measure the accuracy of flagged trades (true/-false positives) and recalibrate thresholds or contamination rates used by the machine learning components.

In summary, the tool offers a solid foundation for trade verification in high-frequency energy markets. By addressing data pipeline inefficiencies and improving real-time integration between the trading desk and data infrastructure, it can evolve into a **fully automated and intelligent anomaly detection system** tailored for EDF UK's trading operations.

9 Appendix and Code

Project Directory Structure on GitHub:

```
trade_analysis_project/
│
├── main.py                    # Main pipeline: fetch, clean, label, compare, export
├── config.py                  # Configurations: file paths, thresholds, credentials
│
├── utils/                     # Core utility logic
│   ├── date_utils.py          # Parse trade date, compute filename/reporting dates
│   ├── time_utils.py          # Format delivery start/end time
│   ├── file_paths.py          # Compute file paths for trade files ← moved here
│   ├── labeling.py            # Add Quarter/Month/Week/ProductLabel columns
│   ├── loadshape.py           # Infer load shape (BASELOAD, PEAKLOAD, etc.)
│   ├── integrity_checks.py    # Final summary, export, diagnostic reporting
│   ├── snowflake_utils.py     # Fetch EDFT prices from Snowflake DB
│   ├── trade_filtering.py     # Load, filter, and clean trade files
│   ├── edft_merging.py        # Merge trades with EDFT price data (with fallback logic)
│   └── price_computation.py   # Compute traded/internal prices and weighted price deltas
│
├── flagging/                  # Trade anomaly detection (modular outlier flagging)
│   ├── main_flagging.py       # Entry point: applies Z, robust Z, IForest
│   ├── flagging_methods.py    # Implements flagging logic (Z-score, MAD, Isolation Forest
│   └── export_results.py      # Highlight and export flagged trades to Excel
│
├── data/
│   └── output/                # Optional subfolder for exported results (CSV or XLSX)
│
└── README.md                  # Project overview, instructions, dependencies
```

Labelling code (for product label):

```python
# Generate EDFT Label Components (Week, Month, Quarter)
def generate_edft_labels(row):
    start = row['StartDate']
    end = row['EndDate']

    start_year, start_month, start_week = start.year, start.month, start.isocalendar().week
    end_year, end_month, end_week = end.year, end.month, end.isocalendar().week

    def get_quarter_label(year, month):
        if month <= 3:
            return f"{year}Q1"
        elif month <= 6:
            return f"{year}Q2"
        elif month <= 9:
            return f"{year}Q3"
        else:
            return f"{year}Q4"

    start_quarter = get_quarter_label(start_year, start_month)
    end_quarter = get_quarter_label(end_year, end_month)
```

```
    start_month_label = f"{start_year}{start_month:02d}"
    end_month_label = f"{end_year}{end_month:02d}"
    start_week_label = f"{start_year}W{start_week:02d}"
    end_week_label = f"{end_year}W{end_week:02d}"

    is_single_quarter = start_quarter == end_quarter
    is_single_month = start_month_label == end_month_label
    is_single_week = start_week_label == end_week_label

    return pd.Series({
        'QuarterLabel': start_quarter if is_single_quarter else
        ↪  f"{start_quarter}{{end_quarter}}",
        'MonthLabel': start_month_label if is_single_month else
        ↪  f"{start_month_label}{{end_month_label}}",
        'WeekLabel': start_week_label if is_single_week else
        ↪  f"{start_week_label}{{end_week_label}}",
        'LabelType': (
            'SingleQuarter' if is_single_quarter else
            'SingleMonth' if is_single_month else
            'MultiPeriod'
        )
    })

# Apply labeling function
df_power = df_power.join(df_power.apply(generate_edft_labels, axis=1))

# Detect full seasonal delivery (Winter or Summer)
def detect_full_season(start_date, end_date):
    if (start_date.month == 10 and start_date.day == 1 and
        end_date.month == 3 and end_date.day == 31 and
        end_date.year == start_date.year + 1):
        return f"{start_date.year}WIN"
    elif (start_date.month == 4 and start_date.day == 1 and
          end_date.month == 9 and end_date.day == 30 and
          start_date.year == end_date.year):
        return f"{start_date.year}SUM"
    return None

# Check full month
def detect_full_month(start_date, end_date):
    next_month = start_date.replace(day=28) + pd.Timedelta(days=4)
    last_day = (next_month - pd.Timedelta(days=next_month.day)).day
    return start_date.day == 1 and end_date.day == last_day and
    ↪  start_date.month == end_date.month and start_date.year ==
    ↪  end_date.year

# Check full ISO week
def detect_full_week(start_date, end_date):
    return start_date.weekday() == 0 and end_date.weekday() == 6 and
    ↪  (end_date - start_date).days == 6

# Check full quarter
def detect_full_quarter(start_date, end_date):
```

```
    q1 = (start_date.month == 1 and start_date.day == 1 and end_date.month
    ↪  == 3 and end_date.day == 31)
    q2 = (start_date.month == 4 and start_date.day == 1 and end_date.month
    ↪  == 6 and end_date.day == 30)
    q3 = (start_date.month == 7 and start_date.day == 1 and end_date.month
    ↪  == 9 and end_date.day == 30)
    q4 = (start_date.month == 10 and start_date.day == 1 and
    ↪  end_date.month == 12 and end_date.day == 31)
    return (start_date.year == end_date.year) and (q1 or q2 or q3 or q4)

# Main ProductLabel generator
def get_product_label(row):
    # First check for full seasonal blocks
    season_label = detect_full_season(row['StartDate'], row['EndDate'])
    if season_label:
        return season_label

    if detect_full_quarter(row['StartDate'], row['EndDate']):
        return generate_edft_labels(row)['QuarterLabel']
    if detect_full_month(row['StartDate'], row['EndDate']):
        return generate_edft_labels(row)['MonthLabel']
    if detect_full_week(row['StartDate'], row['EndDate']):
        return generate_edft_labels(row)['WeekLabel']

    # Else mark as MULTI-period
    return "MULTI"
```