1. Introduction: What drives fuel burn at interval level?

For the PRC 2025 challenge we predict **fuel burn in kg/min** for fixed time intervals along a flight trajectory. Conceptually, we assumed that fuel burn in a given interval is determined by three main groups of factors:

1. **Aircraft and route characteristics**
    - Aircraft type, size and performance (e.g. MTOW, engine count).
    - Route geometry (great-circle distance, origin/destination elevations).
    - Overall flight duration and where along the flight the interval lies.
2. **Kinematics and flight phase**
    - Altitude level and vertical profile (climb, cruise, descent).
    - Ground speed and heading/track.
    - Time since takeoff and time to landing (i.e. “progress along the flight”).
3. **Environment and context**
    - Weather at the aircraft’s position and altitude (winds, temperature, pressure, convection).
    - Broader context such as date, time of day, possibly macroeconomic proxies (experimentally).

From this mental model we defined which data we considered reasonable and feasible to collect:

- **Flight-level tables** with departure/arrival times, airports, aircraft type and the interval fuel labels.
- **Trajectory data** per flight, containing timestamped position and motion (lat/lon, altitude, ground speed, vertical rate, track).
- **Airport metadata** (positions, elevations) to derive route-level variables.
- **Weather reanalysis/forecast** on pressure levels from an external API, sampled at interval midpoints.
- **External time series** like oil prices and MSCI World index (as a speculative experiment, to test whether they might proxy for unobserved operational regimes).

All modelling is done at **interval level**, but many features are defined at **flight level** and then broadcast to intervals.

---

2. Methodology

2.1 Overall pipeline

The full pipeline consists of three conceptual stages:

1. **Data preparation and feature construction**
    - Merge flight-level and interval-level information.

- Construct trajectory-based interval point files and derive altitude/heading features.
- Sample weather and external time series.
- Build incremental feature tables features_intervals_v0 … features_intervals_v6 and the corresponding submission datasets.

2. **Modelling**
   - Start from an initial **R XGBoost baseline**.
   - Port the modelling pipeline to **Python** and extend it:
     - robust preprocessing,
     - sparse one-hot encoding,
     - Bayesian optimisation for hyperparameters,
     - group-wise cross-validation,
     - test-set and ensemble evaluation.
3. **Inference and submission**
   - Apply the final model to the chosen submission feature set.
   - Treat taxi-in/out intervals separately (using a specialised model) and recombine results.
   - Write the final predictions to a parquet file and upload them to the OpenSky/PRC MinIO bucket.

Below we focus on the modelling script and the incremental feature sets.

2.2 Modelling scripts (R baseline → Python PRC2025_V9.py)

R baseline (conceptual)

We created an initial **R-based XGBoost pipeline**, which we used as a starting point:

- Load a feature table (features_intervals.csv) with one row per interval.
- Select a subset of numerical and categorical variables.
- Train an **XGBoost regression model** on interval-level fuel_kg_min.
- Apply the model to the submission feature table and export predictions as a parquet file in the required format.

This gave us:

- a reference performance level,
- a proven set of core features,
- and the expected parquet schema for submissions.

Python model script PRC2025_V9.py

We then implemented an extended and fully Python-based training and inference script in PRC2025_V9.py.
This script replaces the R code while following the same conceptual structure, with several enhancements.

Data loading and filtering

- The script reads a consolidated feature table features_intervals.csv:
    - One row per interval.
    - Includes all engineered features from the latest feature version.
- It derives a **status** variable from pct_elapsed_mid:
    - pct_elapsed_mid < 0 → taxi_out
    - pct_elapsed_mid > 100 → taxi_in
    - otherwise → inflight
- For the main inflight model:
    - All taxi_in and taxi_out intervals are dropped.
    - Rows with missing fuel_kg_min are dropped (labels required).

Feature selection

- The script removes:
    - Columns that are constant or entirely NA.
    - Pure identifiers and time stamps (e.g. idx, flight_id, start, end, flight_date, takeoff, landed).
    - Some high-cardinality/textual categorical columns that do not add predictive value (e.g. weather_code_text, origin_icao, dest_icao, origin_region, dest_region).
- The remaining columns form the input feature set. fuel_kg_min is the target.

Train–test split and group structure

- Flights are split into **train and test sets** at the flight ID level:
    - 80% of flights → training set.
    - 20% of flights → internal test set.
- This ensures that **all intervals of a flight** are always in the same split.
- Flight IDs, aircraft types and flight durations are also kept for building group-wise cross-validation folds.

Preprocessing and sparse design matrix

- The script normalises columns by type:
    - Numeric columns stay numeric (NaN allowed).
    - Boolean columns are converted to 0/1.

    - Other columns (categoricals, strings) are converted to string with an __NA__ sentinel for missing values.

- A custom **one-hot encoder** creates a sparse matrix:
    - Numeric columns → dense numeric block.
    - Categorical columns → sparse one-hot block (get_dummies with sparse output).
    - Both blocks are horizontally stacked into a single CSR matrix.
- A helper aligns column spaces between train and test/submission matrices, dropping unknown columns and zero-filling any columns that appear in training but not submission.

Validation split and DMatrix construction

- Within the training set, 10% of intervals are randomly selected as a **validation set**.
- Two DMatrices are created:
    - dtrain – training rows.
    - dvalid – validation rows.
- A third DMatrix is used for full cross-validation (dtrain_full), containing all training rows.

Bayesian optimisation (hyperparameters)

- If the bayes_opt package is available, Bayesian optimisation (BO) is run over key hyperparameters:
    - eta (learning rate), max_depth, min_child_weight, subsample, colsample_bytree, gamma,
    - reg_lambda, reg_alpha, max_leaves, grow_policy.
- For each candidate parameter set:
    - XGBoost **cross-validation** is run with:
        - 5 folds grouped by flight_id (main score).
        - Optionally 3 folds grouped by aircraft_type.
        - Optionally 3 folds grouped by duration buckets.
    - This encourages robustness across flights, aircraft types and distance regimes.
    - The score combines average RMSE and worst-case RMSE across these folds.
- If BO is not available, a sensible default parameter set is used.

Final model training

- With the final hyperparameters fixed:
    - XGBoost is trained on the training subset with **early stopping** on the validation set.
    - GPU acceleration is requested; if CUDA is not available, the script falls back to CPU.

- The best iteration and validation RMSE are extracted robustly from the model and its evaluation history.

Evaluation

- The model is evaluated on the internal test set:
    - RMSE on fuel_kg_min (kg/min).
    - RMSE on fuel **per interval in kg**, by multiplying predictions and truth by interval_min.
- The script also trains a **10-model ensemble**:
    - Repeated training with different seeds and slightly shrunk subsample/colsample.
    - Predictions are averaged across models.
    - Ensemble performance is reported in the same metrics.

Feature importance and diagnostics

- Feature importances (gain) are extracted from the final model and saved to importance_matrix.csv.
- A detailed **"predicted vs actual" table** for the test set is written, including:
    - Actual and predicted fuel per interval,
    - Absolute and percentage errors,
    - All original features for deeper error analysis.

Submission generation

- The script then reads submission_intervals.csv and recomputes status in the same way as for training.
- It splits the submission data into:
    - Inflight intervals → to be scored by the main model.
    - Taxi-out/in intervals → to be filled from a separate taxi model.
- Taxi predictions are taken from submission_intervals_v4_scored.csv (already run by a specialised taxi model), converted to fuel_kg, and merged back.
- The inflight subset is normalised and one-hot encoded with **exactly the same pipeline** as training:
    - Missing training variables are added as all-NA columns.
    - Sparse matrix is aligned to the training column order.
- The model predicts fuel burn rate (fuel_kg_min); this is multiplied by interval_min to obtain **fuel mass per interval** fuel_kg.
- Taxi and inflight results are concatenated and written to a parquet file in the required schema (idx, flight_id, start, end, fuel_kg), and uploaded to the MinIO/S3 bucket.

---

3. Incremental feature sets V0–V6

Our feature engineering followed an incremental versioning scheme. Each version starts from the previous one and adds new columns. In the end, we trained several models on different versions to assess the marginal benefit of each feature block.

3.1 V0 – Initial data: Flight & interval meta

Scripts:
01_merge_to_features_csv, 01A_merge_to_submission.py, 01B_merge_to_final.py

What we added / calculated

V0 merges the core competition tables (flight list, interval labels, airport data) and adds basic route and timing information:

- **Route geometry and airports**
    - gc_distance_km: great-circle distance between origin and destination, computed via a haversine formula using airport lat/lon and Earth radius 6 371 km.
    - elev_delta_ft: difference between destination and origin airport elevations.
- **Flight-level timing**
    - flight_duration_min: gate-to-gate duration = (landed – takeoff) in minutes.
    - interval_min: duration of an interval = (end – start) in minutes.
    - t_since_takeoff_min: minutes from takeoff to the interval **start**.
    - t_to_landing_min: minutes from interval **end** to landing.
    - pct_elapsed_mid: percentage of the flight that has elapsed at the **interval midpoint**. This is later also used to classify taxi vs inflight intervals.
- **Time of day**
    - start_hour_utc, end_hour_utc: interval start and end hours in UTC, used to capture diurnal effects and operational patterns.
- **Coarse average speed**
    - avg_speed_km_per_min: average ground speed for the entire flight = gc_distance_km / flight_duration_min.
- **Target**
    - fuel_kg_min: fuel burn rate per interval, provided in the training dataset.
    - In submission/final datasets, this remains NA and is predicted by the model.

Why we added it

V0 encodes the **basic mission profile** at a coarse level:

- Distance and elevation difference give a rough idea of **mission energy requirements**.
- Flight duration and elapsed percentage place each interval along the route, distinguishing early climb, mid-flight cruise, and approach, even before we add detailed kinematics.

- Time-of-day can capture differences in typical routing or congestion.

These are inexpensive to compute and robustly available for all flights.

3.2 Trajectory extraction & interpolation (preparation for V1–V2–V4)

Before V1 and V2, we need clean **interval point files** from the trajectories.

Scripts:
02_extract_interval_points.py, 02A_extract_interval_points_submission.py, 02B_extract_interval_points_submission.py

What we do

For each interval (identified by idx, flight_id, start, end) we:

- Load the corresponding flight trajectory parquet from the "flights_final" dataset.
- Rename columns into a canonical set:

timestamp, flight_id, typecode, latitude, longitude, altitude, groundspeed, track, vertical_rate, mach, TAS, CAS, source

- Define **usable** trajectory rows as those with non-missing:

timestamp, latitude, longitude, altitude, groundspeed, track, vertical_rate

- Extract all usable rows whose timestamp lies within [start, end].
    - If there are no such rows, or the interval lies entirely outside the trajectory time span, we fall back to interpolation/extrapolation (see below).

Interpolation/extrapolation rules

We updated the extraction script to handle intervals that extend beyond the recorded trajectory:

1. **Interval fully inside flight span but no complete internal row**
    - Select **left anchor** = last valid row with timestamp ≤ start (or the first row if none).
    - Select **right anchor** = first valid row with timestamp ≥ end (or the last row if none).
    - Create three synthetic rows at start, midpoint and end:
        - Start row: copy alt/speed/track from left anchor.
        - End row: copy from right anchor.
        - Midpoint row: linear interpolate lat/lon/alt between anchors; average groundspeed and vertical_rate; compute circular mean for track.
        - Set source = "interpolation".
2. **Interval completely before the first or after the last datapoint**
    - Find the **nearest** valid trajectory row in time and copy its lat/lon/track.
    - Create three rows at start, midpoint, end:

        - Lat/lon/track from nearest; altitude, groundspeed and vertical_rate set to 0.0.
        - source = "extrapolation".

These interval point files in data/processed/Intervals[...] are used by V1 for altitude metrics and by V2/V4 for track and weather features.

Why we did this

- We wanted at least a **minimal, well-defined position and altitude** for every interval, even when the raw trajectory is incomplete.
- The three-row representation (start, mid, end) allows us to derive **mean values** and detect altitude changes with very little data.
- Explicit interpolation/extrapolation also makes it clear which intervals rely on synthetic information (source column).

3.3 V1 – Altitude difference & vertical speed

Scripts:
05_compute_alt_diff.py, 05A_compute_alt_diff_submission.py, 05B_compute_alt_diff_submission.py

What we added / calculated

Using the interval point files from V0/02, we derive three altitude-related features:

- points_file_exists (0/1): whether a {idx}_flight_data.csv is available for the interval.
- alt_diff:
    - Difference between the last and first non-null altitude in the interval point file:
        - alt_diff = altitude_last - altitude_first (feet).
- alt_mean_ft:
    - Mean altitude over the interval:
        - Streaming average of all non-null altitude values in the point file.
- vs_mean_fpm:
    - Mean vertical speed in ft/min:
        - vs_mean_fpm = alt_diff / interval_min.

Why we added it

- alt_diff and vs_mean_fpm directly capture whether the interval is **climb, descent, or level flight**, which is strongly tied to fuel burn.
- alt_mean_ft distinguishes lower-level segments (e.g. initial climb, terminal area) from cruise altitudes, again with clear fuel implications.
- These features provide a **physics-aligned signal** from trajectories without building a full phase classification model.

3.4 V2 – Track / heading

Scripts:
08_add_heading.py, 08A_add_heading_submission.py, 08B_add_heading_submission.py

What we added / calculated

From each interval point file we extract a representative heading:

- Choose the best available heading-like column in the point file:
    - Prefer track, otherwise heading.
- Extract all non-null values and compute their **median**:
    - This median is written as track in the interval feature table.

Why we added it

- Track encodes the **direction of travel**, which later allows us to compute weather-relative features such as headwind and crosswind.
- Even without explicit wind, heading can correlate with operational behaviour (e.g. typical routing patterns).
- Using the median (instead of just one point) reduces sensitivity to local noise in the trajectory.

3.5 V3 – Regions (origin/destination region labels)

Scripts:
09_map_regions.py, 09A_map_regions_submission.py, 09B_map_regions_submission.py

What we added / calculated

Based on origin_icao and dest_icao, we map airports to higher-level regions:

- origin_region, dest_region:
    - Look up each ICAO code in a region mapping table (e.g. continent or macro-region).
    - Assign a categorical label such as “Europe”, “North America”, etc.

Why we added it

- Region labels capture **large-scale geographic patterns**, which might reflect:
    - Differences in typical routing and altitudes,
    - Different weather regimes,
    - Different operational practices.
- They also allow us to see if the model’s errors systematically differ by region.

In the final Python script we treat these as optional; they can be dropped if they do not improve performance or if they risk overfitting.

3.6 V4 – Weather at interval midpoints

Scripts:
06_prepare_weather_dataset.py, 10_construct_weather_dataset.py, 10A_construct_weather_dataset_submission.py, 15_merge_weather_data.py, 15A_merge_weather_data_submission.py

What we added / calculated

Using the interval point files (with correctly interpolated midpoints) we query the Open-Meteo Historical Forecast API to attach weather at the interval midpoint:

1. **Determine sampling point**
    - Use the **midpoint** row from the interval point file:
        - timestamp, latitude, longitude, altitude.
    - Convert altitude to meters, infer best-matching **pressure level** from altitude.
2. **Query Open-Meteo**
    - Request historical forecast for the latitude/longitude and the date range around the interval midpoint.
    - Retrieve hourly time series for the chosen pressure level and surface variables.
3. **Find nearest model time**
    - Convert model times to UTC and find the forecast time closest to the interval midpoint.
4. **Attach weather variables**, e.g.:
    - At the chosen pressure level:
        - temperature_[level]hPa (°C),
        - relative_humidity_[level]hPa (%),
        - wind_speed_[level]hPa (km/h),
        - wind_direction_[level]hPa (deg),
        - geopotential_height_[level]hPa (m).
    - At the surface:
        - surface_pressure (hPa),
        - precipitation, rain, showers, snowfall,
        - weather_code, weather_code_text.
    - Derived:
        - wind_cardinal (N, NE, E, …).
5. **Merge back**

    - Aggregate these into a weather dataset keyed by idx.
    - Merge into the interval feature tables as features_intervals_v4 and submission_intervals_v4.

We also experimented with **derived variables** such as Delta T ISA (difference between actual temperature and standard atmosphere) and splitting wind into **headwind / crosswind** components using track from V2.

Why we added it

- Weather is a natural candidate driver for fuel burn:
    - Headwinds vs tailwinds change effective groundspeed and required thrust.
    - Temperature and density affect engine performance.
    - Convective activity may induce reroutes, level-offs and speed changes.
- By sampling weather at the aircraft's location and altitude at the correct time, we attempted to encode these effects explicitly.

Empirical outcome

- In the XGBoost feature importance matrix, most weather variables (and their derivatives) received **very low importance scores** and did not consistently improve RMSE.
- We therefore treat the V4 weather block as an **exploratory extension** rather than a core component of the final model.

3.7 V5 – New categoricals: aircraft grouping and phase

Scripts:
Custom feature engineering based on existing columns.

What we added / calculated

V5 adds categorical variables derived from existing continuous features:

- **Aircraft grouping**
    - Group individual aircraft types into coarser families or classes:
        - e.g. narrow-body vs wide-body, or manufacturer families.
    - Derived from aircraft_type and static aircraft specs (MTOW, engines, etc.).
- **Phase categorisation**
    - Convert the continuous vs_mean_fpm (from V1) into a discrete **phase** label:
        - climb if vs_mean_fpm above a positive threshold,
        - descent if below a negative threshold,
        - cruise/level otherwise.
    - Optionally refined with alt_mean_ft and distance to departure/arrival.

These categorical variables are then one-hot encoded in the modelling script.

Why we added it

- Although vs_mean_fpm and alt_mean_ft are already informative, an explicit phase classification:
    - Provides a **high-level indicator** that the model can pick up even if the continuous values are noisy.
    - Allows us to see feature importance and errors **by phase**, which is interpretable for domain experts.
- Aircraft grouping helps generalise behaviour across similar types, particularly where individual types are rare in the data.

3.8 V6 – Oil price and MSCI World

Scripts:
Custom ETL for external time series; merge into interval table by date.

What we added / calculated

V6 brings in simple macroeconomic/context variables:

- oil_price: daily oil price (e.g. Brent/WTI) in USD.
- msci_world: daily closing value of the MSCI World equity index.

These variables are merged at the **flight-date level** and broadcast to all intervals of a flight.

Why we added it

- This was an exploratory idea:
    - Higher oil prices might correlate with different **operational choices** (e.g. fuel-saving procedures).
    - Equity indices might loosely capture different **traffic demand regimes**.
- We wanted to test whether such context variables can capture long-term patterns not explicitly present in the flight data.

Empirical outcome

- In practice, both oil_price and msci_world ended up with **very low feature importance**.
- They did not improve the validation or test RMSE in a reliable way.
- They are therefore best seen as a **research experiment** rather than part of the final production feature set.

---

4. Summary of feature versions and final choice

- V0–V2–V5 provide a **strong, physically interpretable core**:
    - Route geometry, flight timing, altitude and phase, track/heading, aircraft grouping.

- V3 (regions) adds some interpretable geography, but is not critical.
- V4 (weather) and V6 (macro variables) did not add measurable value and increased complexity and external dependencies.
- The final features_intervals.csv used by PRC2025_V9.py is therefore a **curated version**:
    - It includes the most useful features from V0–V5.
    - It may drop or ignore variables that were found to be constant, unhelpful or overly noisy.
    - The modelling script further prunes ID/time-only columns and low-value categoricals.

Combined with the Python model script (BayesOpt + ensemble XGBoost), this feature set yields a model that:

- Is **competitive in RMSE** on interval fuel burn.
- Maintains **physical plausibility** across climb/cruise/descent and across aircraft classes.
- Is implemented in a **reproducible and extensible** codebase, with clear separation between data preparation, feature engineering and modelling.