.
Generate clean, modular, production-ready Python code for the following task.

----
1. High-level goal

Build a complete pipeline that:

1. Loads multiple monthly CSV files(in the Enhanced folder) containing matched
   ADS-B + noise data.
2. Segments the data into individual flights using distance & time rules.
3. Groups flights into “flows” by arrival/departure (A/D) and Runway.
4. Preprocesses each flight trajectory:
   - sort by timestamp
   - clean
   - smooth
   - resample to a fixed number of points
5. Clusters the trajectories within each flow to identify backbone tracks.
6. Computes backbone and side tracks (e.g. p10/p50/p90 envelopes).
7. Writes results to disk as CSV files, and optionally some basic plots.
8. Is driven by a YAML config file (no hard-coded paths or hyperparameters).
9. Supports a TEST MODE that runs the entire pipeline end-to-end on a small subset
   of the data to validate that everything works, but is fast enough for quick checks.

Code should be readable, well-documented, and easy to extend.

----
2. Input data description

Each monthly CSV has the same schema. Important columns:

- timestamp — timestamp of the ADS-B sample (string; parse as datetime)
- latitude
- longitude
- altitude
- geoaltitude
- groundspeed
- vertical_rate
- track
- icao24
- callsign
- MP — measuring point ID (do NOT use this for flight identity)
- t_ref — reference time used during matching (can be ignored for segmentation)
- aircraft_type_noise
- aircraft_type_adsb
- aircraft_type_match
- dist_to_airport_m — distance from airport centre in meters (critical)
- onground
- alert
- spi
- squawk
- last_position
- hour
- firstseen — first detection timestamp for the flight
- origin
- lastseen
- destination
- day
- A/D — "A" for arrival, "D" for departure
- Runway — e.g. "09R", "09L", "27L", "27R", etc.

Notes:

- Data is already filtered to within ~12 km of the airport:
  - dist_to_airport_m ranges roughly from 0 up to 12000 m.
- Within a single flight, dist_to_airport_m is monotonic (increasing or decreasing,
  depending on how matching was done).

- Different flights can have overlapping distance values, e.g. one flight ending around 11977 and another starting around 11989, so distance similarity alone cannot identify flight boundaries.

----

3. Requirements for the codebase

3.1 General

- Use pandas, numpy, PyYAML and scikit-learn.
- Optional: matplotlib for a few simple debug plots (protected by a config flag).
- Use type hints everywhere.
- Use logging instead of print.
- Add concise but meaningful docstrings to all public functions and classes.
- Structure the code as a small package with multiple modules.

3.2 Project structure

Generate code as if the project layout is:

```
backbone_tracks/
    __init__.py
    config.py
    io.py
    segmentation.py
    preprocessing.py
    clustering.py
    backbone.py
    plots.py
    cli.py
```

I don’t need setup.py or packaging; just the code for these modules.

----

4. Configuration via YAML (including test mode)

All important parameters must be in a YAML config file.
Write a loader that reads something like:

```
# config/backbone.yaml

input:
  csv_glob: "data/matched_*.csv"
  parse_dates: ["timestamp", "firstseen", "lastseen"]
  timezone: "UTC"  # assume; no need to convert for now

segmentation:
  time_gap_sec: 60           # Rule A threshold
  distance_jump_m: 600       # Rule C threshold
  min_points_per_flight: 10  # discard super-short flights

flows:
  # columns that define a flow (traffic class)
  flow_keys: ["A/D", "Runway"]

  # which flows to include (optional filter). If empty, use all unique combos.
  include:
    - ["A", "09R"]
    - ["A", "27L"]
    - ["D", "09R"]
    - ["D", "27L"]

preprocessing:
  smoothing:
    enabled: true
    window_length: 7
    polyorder: 2
    columns: ["latitude", "longitude", "altitude"]
  resampling:
    n_points: 40
    method: "time"   # "time" or "index" (allow easily switching strategies)

clustering:
  method: "optics"   # "optics" or "kmeans"
  optics:
    min_samples: 5
    xi: 0.05
    min_cluster_size: 0.05
  kmeans:
    n_clusters: 4
  random_state: 42

backbone:
  percentiles: [10, 50, 90]   # p10/p50/p90
  min_flights_per_cluster: 5  # ignore tiny clusters

testing:
  enabled: true               # if true, run in TEST MODE
  max_rows_total: 50000       # global cap on total rows loaded
  max_flights_per_flow: 20    # cap flights per (A/D, Runway) flow
  sample_flows_only: true     # if true, restrict to flows.include above
  max_clusters_per_flow: 3    # optional: cap number of clusters per flow for test runs

output:
  dir: "output"
  save_preprocessed: true
  save_flight_metadata: true
  save_backbones: true
  save_plots: true
```

You don’t need to validate every field rigorously, but write the code assuming these sections exist and use sensible defaults where missing.

The TEST MODE is very important:

- When testing.enabled is true, the pipeline should still execute ALL steps (load, segment, preprocess, cluster, backbone, output) but on a reduced dataset:
  - truncate after max_rows_total rows right after loading
  - after segmentation, for each flow only keep up to max_flights_per_flow unique flight_ids
  - if sample_flows_only is true, only process flows that appear in flows.include
- TEST MODE must be implemented in cli.main so that a user can validate the end-to-end behaviour quickly (e.g., on a laptop) before running on the full dataset.
- The main logs should clearly state whether test mode is active and how many rows/flights were retained at each step.

----

5. Modules and their responsibilities

5.1 config.py

- Provide load_config(path: str) -> dict.
- Use yaml.safe_load.
- Optionally provide small helper functions to get nested values with defaults.

5.2 io.py

Functions:

1) load_monthly_csvs(csv_glob: str, parse_dates: list[str], max_rows_total: int | None) -> pd.DataFrame
   - Use glob.glob.
   - Read all matching CSVs.
   - Parse date columns given in the config.
   - Concatenate and return a single DataFrame.
   - If max_rows_total is not None, truncate the concatenated DataFrame to that many rows.

2) ensure_required_columns(df: pd.DataFrame) -> pd.DataFrame
   - Check that all required columns are present:
     - timestamp, latitude, longitude, altitude,
       dist_to_airport_m, A/D, Runway

   - If missing, raise a ValueError with a clear message.

3) save_dataframe(df: pd.DataFrame, path: str) -> None

5.3 segmentation.py

Goal: add a flight_id column to the DataFrame.

The algorithm is multi-rule.

Let df be sorted by ["A/D", "Runway", "timestamp"].

Define:

Rule A — Time gap rule
- If timestamp[i] - timestamp[i-1] > time_gap_sec, start a new flight.

Rule B — Direction reset rule
- Consider dist_to_airport_m. Compute differences:
  prev_diff = dist[i-1] - dist[i-2] and curr_diff = dist[i] - dist[i-1].
- If sign(curr_diff) != sign(prev_diff) and curr_diff != 0, start a new flight.

Rule C — Distance jump rule
- If abs(dist[i] - dist[i-1]) > distance_jump_m, start a new flight.

Implement a function:

```
def segment_flights(
    df: pd.DataFrame,
    time_gap_sec: float,
    distance_jump_m: float,
    min_points_per_flight: int,
) -> pd.DataFrame:
    """
    Adds a 'flight_id' column to df using Rules A, B, and C.
    Segmentation is applied per (A/D, Runway) flow.
    Flights with fewer than min_points_per_flight points are removed.
    """
```

Steps:

1) Sort the input frame by ["A/D", "Runway", "timestamp"].
2) For each (A/D, Runway) group:
   - Apply the segmentation logic iterating row-by-row.
   - Assign integer flight_id values local to the whole dataset
     (e.g., simply increment a counter across all flows).
3) After segmentation, drop flights (i.e., flight_ids) with fewer than
   min_points_per_flight rows.
4) Return the augmented DataFrame.

If testing.enabled is true and testing.max_flights_per_flow is set in the config,
add a helper in this module (or preprocessing) that restricts to the first
max_flights_per_flow flights per flow for further processing.

5.4 preprocessing.py

Responsible for per-flight trajectory cleaning & resampling.

Implement:

```
def preprocess_flights(
    df: pd.DataFrame,
    smoothing_cfg: dict,
    resampling_cfg: dict,
) -> pd.DataFrame:
    """
    For each (flight_id, flow), sort by timestamp, optionally smooth
    latitude/longitude/altitude, and resample to a fixed number of points.
    Returns a DataFrame where each row is a resampled trajectory point.
    """
```

Details:

- Group by ["A/D", "Runway", "flight_id"].
- For each group:
  1) Sort by timestamp.
  2) Optionally apply smoothing on the specified numeric columns (per config).
     - Prefer Savitzky–Golay (scipy.signal.savgol_filter) if SciPy is available.
       If not, implement a simple moving average fallback.
  3) Resampling:
     - Use resampling_cfg["n_points"] to interpolate trajectories to exactly N points.
     - Use one of two strategies:
       - "time": normalize time between 0 and 1 and interpolate lat/lon/alt on a regular grid.
       - "index": normalize index between 0 and 1 and interpolate similarly.
  4) Return a DataFrame with columns:
     - A/D, Runway, flight_id, step (0 to N-1),
       latitude, longitude, altitude, dist_to_airport_m,
       plus optional metadata (e.g., icao24, aircraft_type_match) taken from the first row of the flight.

- Concatenate all groups and return.

Write convenience helpers:
- smooth_trajectory(...)
- resample_trajectory(...)

5.5 clustering.py

Perform clustering per flow on resampled trajectories.

Implement:

```
def build_feature_matrix(df: pd.DataFrame, n_points: int) -> tuple[np.ndarray, pd.DataFrame]:
    """
    Build a 2D feature matrix (n_flights, n_features) for clustering.
    Feature vector should be a concatenation of (latitude, longitude) at each step:
    [lat_0, lon_0, lat_1, lon_1, ..., lat_{N-1}, lon_{N-1}].
    Returns (X, metadata_df) where metadata_df has one row per (A/D, Runway, flight_id).
    """

def cluster_flow(
    df_flow: pd.DataFrame,
    method: str,
    cfg: dict,
) -> pd.DataFrame:
    """
    Cluster a single flow (single A/D, Runway).
    Adds a 'cluster_id' to each flight in df_flow (per flight_id).
    Uses either OPTICS (default) or KMeans, depending on 'method'.
    """
```

Details:

- cluster_flow takes a subset of preprocessed trajectories for a single
  (A/D, Runway) combination.

- Use scikit-learn:
  - OPTICS: sklearn.cluster.OPTICS with params from config.
  - KMeans: sklearn.cluster.KMeans with n_clusters from config.

- Clustering is at flight level, not point level:
  - Each row of X corresponds to a full flight (flattened trajectory).
  - Assign the same cluster_id to all rows of that flight in the output DataFrame.

- Filter out flights with cluster_id == -1 when computing backbones (but you can
  keep them in metadata for completeness if desired).

- In TEST MODE, if testing.max_clusters_per_flow is set, it is acceptable
  to post-process cluster_id values to keep only the first N clusters
  and mark the rest as noise or ignore them for backbone calculation,
  so that test runs are small and fast.

5.6 backbone.py

Compute backbone and side tracks.

Implement:

```
def compute_backbones(
    df: pd.DataFrame,
    percentiles: list[int],
    min_flights_per_cluster: int,
) -> pd.DataFrame:
    """
    For each (A/D, Runway, cluster_id), compute per-step percentiles for trajectory variables.
    Typically percentiles = [10, 50, 90].
    Returns a DataFrame where each row corresponds to a (flow, cluster_id, step, percentile).
    """
```

Details:

- df is the preprocessed dataframe with flight_id, cluster_id, and step.

- For each (A/D, Runway, cluster_id):
  - If number of unique flight_id < min_flights_per_cluster, skip this cluster.
  - For each step from 0 to N-1:
    - For each desired percentile P in percentiles:
      - Compute percentile for latitude, longitude, altitude, dist_to_airport_m.
  - Save rows with columns:
    - A/D, Runway, cluster_id, step, percentile,
      latitude, longitude, altitude, dist_to_airport_m,
      plus n_flights in that cluster.

5.7 plots.py (optional)

Simple plotting utilities, used only if output.save_plots is true.

Example functions:

- plot_flight_segmentation(df, ad, runway, output_path)
  - Plot dist_to_airport_m vs timestamp with different colors for flight_id.

- plot_backbone(df_backbone, ad, runway, cluster_id, output_path)
  - Plot backbone and side tracks in map view (lat vs lon, color-coded by percentile).

Use matplotlib; keep plots basic and robust (no fancy styling required).

5.8 cli.py

Provide a simple command-line entry point:

```
def main(config_path: str = "config/backbone.yaml") -> None:
    """
    Orchestrate the entire pipeline:
    - load config
    - load data
    - segment flights
    - (optionally) apply test-mode restrictions
    - preprocess trajectories
    - cluster per flow
    - compute backbones
    - save outputs
    """

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-c", "--config",
        default="config/backbone.yaml",
        help="Path to YAML config file."
    )
    args = parser.parse_args()
    main(args.config)
```

Within main:

1) Load config.
2) Determine whether TEST MODE is enabled from config["testing"]["enabled"].
3) Load all CSVs using csv_glob, optionally truncating to testing.max_rows_total if in test mode.
4) Ensure required columns.
5) Segment flights → add flight_id.
6) If TEST MODE, restrict flights per flow:
   - Keep only flows listed in flows.include if testing.sample_flows_only is true.
   - For each remaining flow, keep at most testing.max_flights_per_flow unique flight_ids.
7) Preprocess → smoothing + resampling.
8) For each flow (A/D, Runway):
   - Cluster flights (respecting clustering.method and clustering params).
   - Save cluster assignments / flight metadata to output.dir.
9) Compute backbones for all clusters in all flows (respecting backbone.min_flights_per_cluster).
10) Save backbone CSV(s) to output.dir.
11) Optionally, generate diagnostic plots if output.save_plots is true.

Use logging to print progress stage by stage and clearly indicate when test mode is active,
how many rows were loaded, how many flows were processed, how many flights per flow, etc.

----

6. Code style and quality

- Use PEP8-compliant style.
- Use type hints and docstrings.
- In each module, put a short module-level docstring.
- Avoid global state; pass config dictionaries explicitly.
- Do not overcomplicate with classes; use functions unless a small class is clearly helpful.
- Avoid external dependencies beyond pandas, numpy, sklearn, yaml, (optional) matplotlib, and optionally scipy for Savitzky–Golay. If SciPy is not available, implement a simple moving average smoother instead.

----

7. Output expectations

When you finish, output the code for all the modules in a single response, using clear
separators like:

```
# backbone_tracks/config.py
<code>

# backbone_tracks/io.py
<code>

...
```

Do not include explanations in between — just the code files, in order.