Flight anomaly detection model

I want to create a system that detects anomalies in real time in flights above Israel, I paid fr24 the 90 dollars subscription so I can have the data and I know I need a server on run pod and need to get at least a year back of flights for training and testing, I'm just not sure what is the best approach to choosing a model help me

Short answer: don't start by "picking *one* magic model". For what you want, the best approach is a **2–3 layer system** where a simple, well-understood baseline runs first, and a sequence model (LSTM/Transformer autoencoder) handles the hard stuff.

I'll walk you through **what to detect, how to represent the data, and which models for each layer**, with a concrete recommendation for a first version on RunPod.

---

1. First decide: what is "anomaly" for you?

Over Israel, you probably care about things like:

1. **Geometric / kinematic anomalies**
   - Very **low altitude** in regions where no flights are normally that low.
   - **Sharp heading changes** (e.g. > X° in Y seconds) at unusual places.
   - Abnormal **speed profile** (too fast/too slow for phase of flight/location).
   - **Weird climb/descent**: crazy vertical speed, oscillations, "yo-yo".
2. **Route / airspace anomalies**
   - Flight crossing **restricted zones / danger areas**.
   - Flight path that doesn't match typical patterns for that origin/destination.
   - Entering Israeli FIR from unusual entry points / directions.
3. **Behavioral anomalies over time**
   - A flight that is "almost normal" point-by-point, but its **whole trajectory** is off (odd path around the country, loitering, zigzag segments where normally routes are straight).

These map to **different modeling levels**, and that's how you should think about choosing models.

---

2. How to represent the data (this is key before model choice)

You'll be pulling FR24 historic + live data. For each **flight track**:

- You already have (roughly):
  - `timestamp`
  - `lat, lon`
  - `alt` (ft)
  - `gspeed` (knots)
  - `vspeed` (ft/min or similar)
  - `heading` (°)
  - `callsign / flight_id`
- You can add **derived features**:
  - `Δlat, Δlon`, distance between points.
  - `turn_rate = Δheading / Δt`.
  - `vertical_accel approx = Δvspeed / Δt`.
  - Distance to nearest airport, distance to FIR boundary, distance to restricted polygon (precomputed with shapely).
  - Flight phase estimate: climb / cruise / descent / approach / pattern.

Then think in **three granularities**:

1. **Point-level features** (one row per timestamp).
2. **Window-level sequences** (e.g. last 30–60 points for a given flight).
3. **Trajectory-level summary** (e.g. full flight or big chunks).

Different models shine at different levels.

---

3. Model layers – what's appropriate for you

Layer 1 – Fast rules + statistics (no ML, just sanity)

Purpose: instantly catch obvious craziness and give you a baseline even before training.

Examples:

- **Altitude rule**: If altitude < X ft inside restricted polygon or sea area where nothing normally is → alert.
- **Turn rule**: If heading change > 45° in < N seconds at cruise altitude → suspect.

- **Speed rule**: If speed outside [min, max] for phase (eg. > 550 knots below 10k ft, < 90 knots at FL300) → suspect.

These are:

- **Explainable**.
- Cheap (pure Python / NumPy).
- Work even when ML fails.

You should absolutely have this, even when you add fancy models.

Layer 2 – Point / window anomaly model (classical ML)

Here you treat each time step (or very short window) as a vector of features and train an **unsupervised anomaly model** on *normal* data.

Good choices:

- **Isolation Forest** (sklearn)
- **Local Outlier Factor (LOF)** – more sensitive but heavier.
- Simple **robust Z-scores** per feature + combination.

Why this layer is good for you:

- No labels needed; a year of "mostly normal" flights is enough (attacks are rare → treated as anomalies).
- Fast to train and very fast to run in real time.
- Easy to scale: you can even train **separate models** for:
  - En-route vs terminal area,
  - North vs South sectors,
  - Civil vs military corridors (if you separate them).

How it works:

1. Build dataset from 1 year:
   - For each in-air point in Israel FIR, create a vector:

```
text

[normalized_lat, normalized_lon,
 normalized_alt, normalized_speed,
 normalized_vspeed, normalized_turn_rate,
 context_features...]
```

- Option A: one big model for all.
- Option B: separate by phase / sector.

2. Train, for example, an **IsolationForest**:
   - It learns what “normal” density in feature space looks like.
   - Each new point gets an anomaly score (0–1).
3. Real time:
   - For each incoming FR24 update: compute features, feed to model, get score.
   - If score > threshold (e.g. 0.99) → point-wise anomaly.

This layer will catch, e.g.:

- Very low altitude in region where no training points had that alt.
- Speed/heading combination that’s never seen in that location/phase.

It **won’t** fully understand “weird full trajectory”, only local behavior.

Layer 3 – Sequence model (LSTM / Transformer autoencoder)

This is where your previous LSTM autoencoder idea actually makes sense.

Goal: detect flights whose **temporal pattern** is unusual, not just a single point.

Best fit for you:

- **LSTM (or GRU) autoencoder** on sequences of fixed length, e.g. last 30–60 samples (5–10 minutes).
- You can later switch to a light **Transformer** (e.g. small encoder) if you want.

Why LSTM autoencoder is a good choice here:

- You have **rich time series** per flight.
- You don’t have labels for anomalies.
- Autoencoder trained on normal sequences learns to reconstruct typical motion. Abnormal sequences → high reconstruction error.

Feature sequence example (per time step):

```text
[lat_norm, lon_norm,
 alt_norm, speed_norm,
 vspeed_norm, sin(heading), cos(heading),
```

```
distance_to_airport_norm, distance_to_restricted_norm,
phase_one_hot...]
```

Training:

1. From your 1 year dataset:
   - Group by `callsign` / `fid`.
   - Sort by time.
   - Break each flight into overlapping windows of length `T` (e.g. 30 steps).
   - Use mostly "clean" flights (remove obvious outliers by rules/IF first).
2. Train LSTM autoencoder to minimize reconstruction error on those windows.
3. Compute reconstruction error distribution on validation set and choose a threshold (like 99th percentile) → anomaly cutoff.

Real time:

- For each flight being tracked, maintain a sliding window of last `T` samples.
- Every new point:
  - Update window.
  - Run through autoencoder → reconstruction error.
  - If error > threshold persistently (e.g. 3 windows in a row) → trajectory anomaly.

This layer will catch things like:

- Flight that deviates from typical corridor and loiters or zigzags.
- Approaches to Israeli airports with strange path shape.
- Drone/missile-like profiles mixed into civilian flows (assuming training on civilians).

---

4. How to choose *for now* – concrete recommendation

Given what you told me (RunPod server, FR24, 1 year of data, POC but serious):

Step 0 – Data plumbing (you're kind of already there)

- Use FR24 historical / archive API to pull:
  - All flights intersecting **Israel FIR bounding polygon** for ~1 year.
- Store as:
  - `parquet` by day (`YYYY-MM-DD_israel.parquet`) to S3/minio/disk.
- Preprocess:
  - Filter only **in-air** points (FR24 flags + altitude > X ft).

- Deduplicate, resample to fixed interval (e.g. every 10 seconds).

(I know you already work with bounding boxes & WKT; reuse that.)

Step 1 – Build the Layer 1 + Layer 2 baseline (no GPU needed)

On RunPod you can even start with a CPU instance:

1. Implement:
   - Rule engine (Python + shapely + some config JSON).
2. Train **IsolationForest** on last N months of point-level data.
3. Serve with a **FastAPI** service:
   - `/score_point` → returns:
     - `rule_alerts`
     - `iforest_score`
   - Persistence of per-flight last window (for Layer 3 later).

You'll immediately get:

- A working anomaly detector.
- Metric: % of points/Flights flagged on historical data.
- Feedback: where rules overfire / underfire → tune thresholds.

Step 2 – Add LSTM autoencoder (GPU helpful but not mandatory)

Once you have the baseline:

1. Spin up **RunPod GPU** (cheap 8–12GB VRAM is enough).
2. Use PyTorch / TensorFlow to:
   - Build LSTM autoencoder (2–3 layers, modest hidden size).
   - Train on sequences sampled from your 1-year dataset.
3. Export the trained model (e.g. TorchScript or just `.pt`) and run **inference** even on CPU for real time (LSTM with small sizes is cheap).
4. Extend your API:
   - `/score_sequence` or just `/score_flight_state` that:
     - Maintains sliding window per `callsign`.
     - Returns `sequence_anomaly_score`.

Step 3 – Fusion and UX

Create a final per-flight anomaly verdict:

- Combine:
  - `max_point_iforest_score` last N seconds,
  - `sequence_recon_error`,
  - `rule_hits_count`.
- Map to:
  - `NORMAL / SUSPICIOUS / ANOMALOUS`
  - - explanation: “low altitude over area X”, “unusual turn at FL340”, etc.

This fusion can be as simple as:

```
text


if any_critical_rule_hit:
    label = "ANOMALOUS"
elif seq_error > high_threshold or point_score > hi_point_thresh:
    label = "SUSPICIOUS"
else:
    label = "NORMAL"
```

---

5. So… which model should you pick?

If I have to answer your question in one sentence:

> **Use a hybrid: rules + IsolationForest for local anomalies, and an LSTM autoencoder for trajectory-level anomalies.**

- Start with **IsolationForest** (very fast to get value, no GPU).
- Once data plumbing is stable, **add LSTM autoencoder** for richer behavior.
- Don’t rely only on fancy deep learning; the rules and IF give robustness, explainability, and quick wins.

If you want, in the next step I can:

- Sketch a **minimal architecture diagram** (ingestion → feature store → models → API).
- Or give you **concrete Python skeletons**:

- `train_iforest.py`
- `train_lstm_autoencoder.py`
- `realtime_scoring_service.py` with FastAPI.

What is feature store?

A **feature store** is simply a structured place where you **compute, store, and serve the features** (inputs) that your ML models need — especially when you must use **the exact same features** during *training* and *real-time inference*.

It prevents *"training one thing, predicting another"* — the most common reason anomaly systems fail.

Let me explain it in your context with flights.

🔥 Simple Definition

A **feature store** is a system (or even just a clean folder structure + code) where you keep:

1. **Feature definitions** (the logic for calculating each feature)
2. **Historical feature values** (for training)
3. **Real-time feature calculation** (for inference)
4. **Consistency guarantees** (same math in both training & real time)

It ensures the model always receives **the same inputs**, no matter when or where the data is coming from.

🛫 Why YOU need a feature store for flight anomaly detection (very important!)

You already have raw data:

```sql
lat, lon, alt, speed, vspeed, heading, timestamp
```

But the models we discussed require **derived features** like:

- turn_rate
- change_in_altitude
- normalized_speed
- distance to FIR boundary
- distance to restricted area
- flight_phase
- delta_lon, delta_lat
- vertical_acceleration
- sin(heading), cos(heading)

During **training**, you compute these from historical FR24 data.
During **real-time**, you compute the *same exact features* from live FR24 streams.

If training and real-time versions of these calculations diverge even a little — the model becomes useless.

That's why you need a **feature store**.

🧠 Example: small feature store for your use case

📁 Structure

```yaml
/features/
  definitions/
    flight_features.py
  historical/
    year2024/
      2024-01-01.parquet
      2024-01-02.parquet
      ...
    year2025/
      ...
  realtime/
    state_cache/
      flight_XYZ123.json
      flight_ELY341.json
```

✔️ What happens inside

1. Feature definitions (core logic)

A single python file:

```python
def compute_features(prev_point, curr_point):
    return {
        "lat": curr_point.lat,
        "lon": curr_point.lon,
        "alt_norm": curr_point.alt / 45000,
        "speed_norm": curr_point.gspeed / 600,
        "vspeed_norm": curr_point.vspeed / 6000,
        "turn_rate": angle_diff(curr_point.heading, prev_point.heading) / dt,
        "vertical_accel": (curr_point.vspeed - prev_point.vspeed) / dt,
        "sin_heading": np.sin(np.radians(curr_point.heading)),
        "cos_heading": np.cos(np.radians(curr_point.heading)),
        "dist_to_border": distance_to_fir(curr_point.lat, curr_point.lon),
        "phase": get_flight_phase(...)
    }
```

This file must be used in *both* training and live inference.

---

2. Historical feature store

You run a batch preprocessing on 1 year of FR24 raw tracks:

```python
features = compute_features(prev, curr)
```

Store them in Parquet files:

```bash
/features/historical/2024-05-14.parquet
```

These are used for:

- training IsolationForest
- training LSTM autoencoder

- analytics
- threshold tuning

---

3. Real-time feature store / cache

When a flight updates every ~10 seconds:

- You load the previous state from `/features/realtime/state_cache/`
- You compute new features
- You store the updated state back

This enables:

- sliding windows for LSTM
- turn-rate calculations
- vertical acceleration
- anomaly scoring in real time

🔥 Why it matters (critical reason)

If your LSTM autoencoder was trained using turn_rate = Δheading/Δt computed a certain way, but in real time you accidentally compute it differently (e.g., missing modulo 360 handling)…

your model breaks immediately.

Bad predictions, false alarms, missed anomalies.

The feature store eliminates this risk.

💡 What is the simplest form of feature store you should use?

You don’t need Redis, Feast, or Snowflake.

Start with:

- **One Python module** with all feature formulas
- **Parquet files** for historical features

- **A tiny JSON state cache** for real-time windows
- **A loader function** that both training and inference call

This is more than enough for your POC and actually used in real production systems.

🚀 Want me to build it for you?

I can generate:

✅ `feature_definitions.py` (all features you need)
✅ `build_feature_store.py` (extract 1 year into Parquet)
✅ `live_feature_service.py` (real-time feature updates)
✅ Model training code using the same feature definitions
✅ Architecture diagram of how the whole system connects on RunPod

Just tell me **"build it"** and I'll generate the complete system.

Do I need that for the lstm?

Short answer: **YES — you absolutely need a feature store (or at least unified feature logic) for the LSTM**.
Not because it's "nice to have", but because **LSTMs break instantly** if the features at training and inference are not **identical**.

Let me explain it clearly and practically.

✅ Why the LSTM *requires* consistent features

An LSTM autoencoder learns **patterns over time**—how altitude changes, how heading curves, how speed evolves, etc.

If the numbers you feed it in real time are even slightly different in scale or definition from the numbers you used during training, the model will:

- output huge reconstruction errors
- generate false positives
- miss real anomalies

- become useless

This is the **#1 reason anomaly detection LSTM systems fail in production.**

🧠 Example of why a feature store is required

Training data:

You compute turn rate like this:

```
python

turn_rate = (heading_curr - heading_prev) / dt
```

Real-time data:

You mistakenly compute it like this:

```
python

turn_rate = abs(heading_curr - heading_prev) / dt
```

Or you forget to normalize:

```
python

turn_rate /= 180
```

→ **Your model suddenly sees "nonsense" values.**

The model will think every flight is an anomaly.

❗ What if you feed the LSTM raw lat/lon/speed/alt only?

You *can*, but you will still need:

1. **normalization**
2. **consistent scaling**
3. **a sliding window buffer of past points**
4. **flight-phase logic or derived metrics if you want good performance**

Even with “raw” features, you still need the feature store logic:

Example:

Training:

```python
alt_norm = alt / 43000
speed_norm = speed / 600
```

Live:

```python
alt_norm = alt / 42000   # a tiny difference!
speed_norm = speed / 550
```

Your LSTM explodes.

Normalization MUST be identical → feature store solves that.

🏗️ What is the *minimal* feature store required for LSTM?

You only need **3 things**:

1. One Python function shared by both training and live inference:

```python
from config import normalizers

def compute_features(prev, curr):
    dt = curr.timestamp - prev.timestamp

    return {
        "alt_norm": curr.alt / normalizers["alt"],
        "speed_norm": curr.speed / normalizers["speed"],
        "vspeed_norm": curr.vspeed / normalizers["vspeed"],
        "heading_sin": math.sin(math.radians(curr.heading)),
        "heading_cos": math.cos(math.radians(curr.heading)),
```

```
    "turn_rate": normalize_turn_rate(curr.heading, prev.heading, dt)
  }
```

Both training and real-time call **the same code**.

---

2. Normalization values saved in a file (`normalizers.json`)

```json
{
  "alt": 45000,
  "speed": 600,
  "vspeed": 6000
}
```

You load these for real-time AND training.

They MUST be identical.

---

3. State cache (for prev point + sliding window)

You keep per-flight last N minutes in memory or small JSON.

LSTM requires sequences → you must store them somewhere.

No sequence history = LSTM can't run.

🎯 What happens if you DON'T use a feature store?

Here is exactly what will happen:

- LSTM will detect "anomalies" in 99% of flights (false positives).
- Normal flights will look abnormal to the model.
- Reconstruction error thresholds become impossible to tune.
- Results vary depending on how the code is deployed (hell).
- You'll waste GPU time training a model that won't generalize.
- Debugging becomes extremely hard.

This is why **every professional anomaly detection pipeline** has a feature store, even a minimal one.

✔️ FINAL ANSWER

Yes — for an LSTM to work reliably on flight anomaly detection, you need a feature store OR at least one shared feature-definition module + consistent normalization + sequence caching.

It does not need to be complex.

The *minimum viable* feature store is:

1. `feature_definitions.py`
2. `normalizers.json`
3. Real-time state cache (per-flight window)

This guarantees:

- identical training & inference behavior
- stable LSTM predictions
- reproducible results
- far fewer false alarms

Want me to generate the exact minimal code for:

- `feature_definitions.py`
- `build_historical_features.py`
- `real_time_feature_cache.py`
- `train_lstm_autoencoder.py`
- `predict_lstm_autoencoder.py`

Just say **"build it"** and I will give you the full working system.

↳ 2. Normalization values saved in a file (normalizers.json)

Who decides those values

Great question — **who decides the normalization values?**
Short answer: **your data decides them**, not you.
Let me explain exactly how.

✅ What are normalization values?

These are numbers used to scale features into stable ranges for the LSTM, for example:

```json
{
  "alt": 45000,
  "speed": 600,
  "vspeed": 6000
}
```

You use them like:

```python
alt_norm = alt / 45000
speed_norm = speed / 600
vspeed_norm = vspeed / 6000
```

This keeps everything between **0 and ~1**, which is what LSTMs learn best.

🧠 So who decides the numbers?

✔️ You DO NOT guess them.

✔️ They come automatically from analyzing your 1-year dataset.

The process:

1. Load the historical data (1 year of FR24 flights across Israel).

2. Compute normalization stats automatically:

Examples:

- For altitude: take the 99th percentile of all altitudes
  → ~43,000 ft
- For speed: 99th percentile of all ground speeds
  → ~580 knots
- For vertical speed: 99th percentile of absolute rates
  → ~5,500 ft/min

These become your constants.

🧪 Example code: auto-generate normalization values from the dataset

```python
import pandas as pd
import json
import numpy as np

df = pd.read_parquet("israel_flights_2024.parquet")

normalizers = {
    "alt": np.percentile(df["alt_ft"].dropna(), 99),      # ~43000
    "speed": np.percentile(df["gspeed_knots"].dropna(), 99), # ~580
    "vspeed": np.percentile(abs(df["vspeed_ftmin"].dropna()), 99), # ~5500
}

with open("normalizers.json", "w") as f:
    json.dump(normalizers, f, indent=4)
```

That's it — the data decides.

📍 Why percentile and not max?

Because FR24 sometimes has:

- noise
- extreme spikes
- glitched readings (speed 1200 knots for 1 frame)
- bad barometric altitudes

Using the **99th percentile** removes these anomalies from your normalization values.

This makes the LSTM stable.

🧩 How this affects your system

During training:

You load `normalizers.json`.

During real-time inference:

You load the same `normalizers.json`.

Result:

Identical feature scaling → stable LSTM anomaly detection.

🎯 Summary

Who decides the normalization values?
➡ Your dataset does — automatically.

How?
➡ By computing percentiles of each feature over the full year of data.

Why is this important?
➡ So that training and inference stay consistent, and your LSTM recognizes what "normal flight behavior" looks like.

If you want, I can generate:

- code to auto-build `normalizers.json`
- full feature store setup
- full training pipeline for your LSTM autoencoder
- real-time inference server to deploy on RunPod

Just say **"build the pipeline"**.

Ok so I will give the why that I get the data and you will create me a class for get that normallization and one for the first layer another for the second that include predict and train, and a class that extracts the extra features from the data

Perfect — that's exactly the right architecture.

Before I generate the final production-ready classes, I need **one thing from you**:

👉 Show me the way you get the data

(i.e., the exact format of the raw FR24 records that arrive in your system).

Because everything depends on this:

- feature extractor needs to know your fields
- normalizer needs to know your columns
- LSTM trainer needs your timestamp spacing
- point-level anomaly model needs your raw values

📌 Please paste one of these:

Option A — A single JSON/Dict example from your system

What your code receives for *each point*:

```json
{
  "timestamp": "2025-11-12T07:48:39Z",
  "lat": 55.4906,
  "lon": 37.12311,
  "alt": 8200,
  "speed": 243,
  "vspeed": 3200,
  "heading": 187,
  "callsign": "FDB8490",
  "source": "MLAT"
}
```

Option B — The CSV columns (if you load from CSV)

Option C — The object/Dict structure you receive from FR24 in your code

(e.g., the Python dict returned by `client.live.flight_positions.get_light(...)`)

❗ Why I need this

So I will produce code that matches your exact field names and avoids:

- mismatched keys
- wrong units (ft vs meters, knots vs km/h)
- wrong timestamp parsing
- missing vs wrong features

✔️ After you send me an example, I will generate:

📦 1. FeatureExtractor class

- computes:
  - turn rate
  - delta alt
  - vertical acceleration
  - heading sin/cos
  - distance to border
  - phase etc.
- robust missing-value handling
- requires no modification later

📦 2. NormalizerBuilder class (auto-normalization)

- scans your dataset (Parquet/CSV/list of dicts)
- computes normalization stats (99th percentile)
- writes `normalizers.json`
- provides `normalize()` and `denormalize()`

📦 3. FirstLayerModel class (rules + IsolationForest)

Includes:

- `.train(df_features)`
- `.predict(feature_vector)`
- `.save()` & `.load()`
- threshold tuning

📦 4. SecondLayerLSTM class

- builds LSTM/GRU autoencoder
- sliding window handling
- `.train(windows)`
- `.predict(sequence)` → returns anomaly score
- automatic sequence normalization
- `.save()` + `.load()`

📌 Final output:

A complete reusable ML pipeline:

```bash

/ml/
  feature_extractor.py
  normalizer.py
  first_layer.py
  second_layer_lstm.py
  common_utils.py
  normalizers.json
  models/
```

All classes with type hints, clean architecture, and ready to integrate into your RunPod service.

👉 **Send me ONE real data example now**, and I will generate the full system.

,timestamp,lat,lon,alt,gspeed,vspeed,track,squawk,callsign,source
0,2025-11-16T17:52:31Z,41.26366,28.74887,0,0,0,286,0656,,ADSB
1,2025-11-16T18:35:25Z,41.26361,28.74911,0,3,0,289,5376,,ADSB
2,2025-11-16T18:35:39Z,41.26357,28.74933,0,3,0,292,5376,,ADSB
3,2025-11-16T18:35:53Z,41.26353,28.74955,0,3,0,306,5376,,ADSB
4,2025-11-16T18:36:06Z,41.26348,28.74975,0,3,0,320,5376,,ADSB
5,2025-11-16T18:36:17Z,41.2634,28.74995,0,3,0,345,5376,,ADSB
6,2025-11-16T18:36:38Z,41.26318,28.75006,0,3,0,5,5376,,ADSB
7,2025-11-16T18:36:54Z,41.26297,28.75007,0,3,0,14,5376,,ADSB
8,2025-11-16T18:37:13Z,41.26275,28.75,0,3,0,16,5376,,ADSB
9,2025-11-16T18:37:31Z,41.26253,28.74991,0,3,0,8,5376,,ADSB
10,2025-11-16T18:37:47Z,41.26232,28.74983,0,3,0,2,5376,,ADSB

Nice, that CSV is exactly what I needed.
I'll give you a **clean, modular setup** with 4 core pieces:

1. `FlightFeatureExtractor` – builds extra features (turn rate, deltas, etc.)
2. `NormalizationStats` – computes & applies normalizers from historical data
3. `FirstLayerAnomalyModel` – IsolationForest (point-level anomalies)
4. `LSTMSequenceAnomalyModel` – LSTM autoencoder (sequence anomalies)

All written to work directly with data in the format you sent.

🔧 Dependencies

Put these (or similar) in your `requirements.txt`:

txt

```
pandas
numpy
scikit-learn
tensorflow
```

1️⃣ Feature extraction class

python

```python
# features.py
import math
from typing import List, Dict, Optional

import numpy as np
import pandas as pd


class FlightFeatureExtractor:
    """
    Extracts extra kinematic features from FR24-like data.

    Expected columns in df:
    ['timestamp','lat','lon','alt','gspeed','vspeed','track','squawk','callsign','source']
    """

    def __init__(self):
        pass

    @staticmethod
    def _angle_diff_deg(a: float, b: float) -> float:
        """
        Smallest signed difference between two headings in degrees.
        """
        diff = (a - b + 180) % 360 - 180
        return diff

    def add_features(self, df: pd.DataFrame) -> pd.DataFrame:
        df = df.copy()
```

```
        # Ensure required columns exist
        required = ["timestamp", "lat", "lon", "alt", "gspeed", "vspeed", "track", "callsign"]
        missing = [c for c in required if c not in df.columns]
        if missing:
            raise ValueError(f"Missing required columns: {missing}")

        # Parse timestamps and sort
        df["timestamp"] = pd.to_datetime(df["timestamp"], utc=True)
        df["callsign"] = df["callsign"].fillna("").replace("", "UNKNOWN")

        df = df.sort_values(["callsign", "timestamp"]).reset_index(drop=True)

        # Time delta per flight (seconds)
        df["dt_sec"] = (
            df.groupby("callsign")["timestamp"].diff().dt.total_seconds().fillna(0.0)
        )
        # Avoid zero dt (for first point): set to median or small value
        df.loc[df["dt_sec"] <= 0, "dt_sec"] = df["dt_sec"][df["dt_sec"] > 0].median() or 1.0

        # Deltas
        df["d_alt"] = df.groupby("callsign")["alt"].diff().fillna(0.0)
        df["d_speed"] = df.groupby("callsign")["gspeed"].diff().fillna(0.0)

        # Turn rate (deg/sec)
        prev_track = df.groupby("callsign")["track"].shift(1)
        prev_track = prev_track.fillna(df["track"])  # first point of flight
        angle_diff_vec = np.vectorize(self._angle_diff_deg)
        df["turn_deg"] = angle_diff_vec(df["track"].values, prev_track.values)
        df["turn_rate_deg_per_s"] = df["turn_deg"] / df["dt_sec"]

        # Heading sin/cos
        df["heading_sin"] = np.sin(np.deg2rad(df["track"]))
        df["heading_cos"] = np.cos(np.deg2rad(df["track"]))

        # Absolute vspeed
        df["vspeed_abs"] = df["vspeed"].abs()

        # You can add more features here (distance to border, phase, etc.)

        return df

    @staticmethod
```

```
    def default_feature_columns() -> List[str]:
        """
        Which columns to feed into models by default.
        """
        return [
            "lat",
            "lon",
            "alt",
            "gspeed",
            "vspeed",
            "d_alt",
            "d_speed",
            "turn_rate_deg_per_s",
            "heading_sin",
            "heading_cos",
            "vspeed_abs",
        ]
```

> Note: `"turn_rate_deg_per_s"` is derived from `"turn_deg"` and `"dt_sec"`—I created it but also kept `"turn_deg"`. You can remove `"turn_deg"` from the DataFrame if you want it cleaner.

2️⃣ Normalization class (auto from historical data)

python

```python
# normalization.py
import json
from typing import Dict, List

import numpy as np
import pandas as pd


class NormalizationStats:
    """
    Holds per-feature normalization values and applies them consistently.
    Typically: value_norm = value / normalizer[feature].
    """

    def __init__(self, normalizers: Dict[str, float]):
```

```
        self.normalizers = normalizers

    @classmethod
    def from_dataframe(
        cls,
        df: pd.DataFrame,
        feature_cols: List[str],
        percentile: float = 99.0,
    ) -> "NormalizationStats":
        normalizers: Dict[str, float] = {}
        for col in feature_cols:
            if col not in df.columns:
                raise ValueError(f"Column {col} not found in dataframe for normalization.")
            vals = df[col].replace([np.inf, -np.inf], np.nan).dropna()
            if vals.empty:
                # fallback to 1.0 to avoid division by zero
                normalizers[col] = 1.0
            else:
                normalizers[col] = float(np.percentile(np.abs(vals), percentile))
                if normalizers[col] == 0.0:
                    normalizers[col] = 1.0
        return cls(normalizers)

    def save(self, path: str) -> None:
        with open(path, "w") as f:
            json.dump(self.normalizers, f, indent=4)

    @classmethod
    def load(cls, path: str) -> "NormalizationStats":
        with open(path, "r") as f:
            normalizers = json.load(f)
        return cls(normalizers)

    def normalize_df(self, df: pd.DataFrame, feature_cols: List[str]) -> np.ndarray:
        """
        Returns numpy array X_normalized of shape (n_samples, n_features).
        """
        X_list = []
        for col in feature_cols:
            if col not in df.columns:
                raise ValueError(f"Column {col} not found in dataframe for normalization.")
            base = df[col].to_numpy(dtype=float)
```

```
            norm_val = self.normalizers.get(col, 1.0)
            X_list.append(base / norm_val)
        X = np.stack(X_list, axis=1)
        return X

    def normalize_array(self, x: np.ndarray, feature_cols: List[str]) -> np.ndarray:
        """
        For real-time single point or batch, x shape: (n_samples, n_features)
        in the same order as feature_cols.
        """
        x = np.asarray(x, dtype=float)
        x_norm = x.copy()
        for idx, col in enumerate(feature_cols):
            norm_val = self.normalizers.get(col, 1.0)
            x_norm[:, idx] = x_norm[:, idx] / norm_val
        return x_norm
```

3️⃣ First layer – IsolationForest (point-level anomalies)

python

```
# first_layer.py
from typing import List, Optional, Tuple

import joblib
import numpy as np
import pandas as pd
from sklearn.ensemble import IsolationForest

from normalization import NormalizationStats


class FirstLayerAnomalyModel:
    """
    Point-level anomaly detector using IsolationForest on normalized features.
    """

    def __init__(
        self,
        feature_cols: List[str],
        contamination: float = 0.01,
```

```
        random_state: int = 42,
    ):
        self.feature_cols = feature_cols
        self.contamination = contamination
        self.random_state = random_state
        self.iforest: Optional[IsolationForest] = None
        self.normalizer: Optional[NormalizationStats] = None

    def fit(
        self,
        df_features: pd.DataFrame,
        normalizer: NormalizationStats,
    ) -> None:
        """
        Train IsolationForest on historical feature dataframe.
        """
        self.normalizer = normalizer
        X = self.normalizer.normalize_df(df_features, self.feature_cols)

        self.iforest = IsolationForest(
            n_estimators=200,
            contamination=self.contamination,
            random_state=self.random_state,
            n_jobs=-1,
        )
        self.iforest.fit(X)

    def predict_scores(self, df_features: pd.DataFrame) -> np.ndarray:
        """
        Returns anomaly scores (the higher, the more anomalous).
        """
        if self.iforest is None or self.normalizer is None:
            raise RuntimeError("Model not fitted yet.")
        X = self.normalizer.normalize_df(df_features, self.feature_cols)
        # IsolationForest gives negative scores for anomalies; invert for intuition
        raw_scores = self.iforest.score_samples(X)  # higher = more normal
        scores = -raw_scores  # higher = more anomalous
        return scores

    def predict_labels(self, df_features: pd.DataFrame, threshold: Optional[float] = None) ->
np.ndarray:
        """
```

```
        Returns binary labels (1 = anomalous, 0 = normal).
        If threshold is None, uses 0 as cutoff on inverted scores (approx).
        """
        scores = self.predict_scores(df_features)
        if threshold is None:
            threshold = np.percentile(scores, 100 * (1 - self.contamination))
        labels = (scores >= threshold).astype(int)
        return labels

    def save(self, model_path: str, normalizer_path: str) -> None:
        if self.iforest is None or self.normalizer is None:
            raise RuntimeError("Model or normalizer not set.")
        joblib.dump(self.iforest, model_path)
        self.normalizer.save(normalizer_path)

    def load(self, model_path: str, normalizer_path: str) -> None:
        self.iforest = joblib.load(model_path)
        self.normalizer = NormalizationStats.load(normalizer_path)
```

4️⃣ Second layer – LSTM autoencoder (sequence anomalies)

Here I'll use `tensorflow.keras`. You train on windows of sequences per flight.

python

```
# second_layer_lstm.py
from typing import List, Optional, Tuple, Dict

import numpy as np
import pandas as pd
import joblib
import tensorflow as tf
from tensorflow.keras import layers, models

from normalization import NormalizationStats


class LSTMSequenceAnomalyModel:
    """
    LSTM autoencoder for sequence-level anomaly detection.
    Works on sequences of shape (seq_len, n_features).
```

```
    """

    def __init__(
        self,
        feature_cols: List[str],
        seq_len: int = 30,
        latent_dim: int = 32,
    ):
        self.feature_cols = feature_cols
        self.seq_len = seq_len
        self.latent_dim = latent_dim

        self.model: Optional[tf.keras.Model] = None
        self.normalizer: Optional[NormalizationStats] = None

    def _build_model(self, n_features: int) -> tf.keras.Model:
        inputs = layers.Input(shape=(self.seq_len, n_features))

        # Encoder
        x = layers.LSTM(64, return_sequences=True)(inputs)
        x = layers.LSTM(self.latent_dim, return_sequences=False)(x)

        # Bottleneck
        encoded = layers.Dense(self.latent_dim, activation="relu")(x)

        # Decoder
        x = layers.RepeatVector(self.seq_len)(encoded)
        x = layers.LSTM(self.latent_dim, return_sequences=True)(x)
        x = layers.LSTM(64, return_sequences=True)(x)
        outputs = layers.TimeDistributed(layers.Dense(n_features))(x)

        model = models.Model(inputs, outputs)
        model.compile(optimizer="adam", loss="mse")
        return model

    @staticmethod
    def _df_to_sequences(
        df: pd.DataFrame,
        feature_cols: List[str],
        seq_len: int,
    ) -> np.ndarray:
        """
```

```
    Build sequences per callsign from a feature dataframe.
    """
    sequences = []
    for callsign, group in df.groupby("callsign"):
        group = group.sort_values("timestamp")
        X = group[feature_cols].to_numpy(dtype=float)
        if len(X) < seq_len:
            continue
        for i in range(len(X) - seq_len + 1):
            seq = X[i : i + seq_len]
            sequences.append(seq)
    if not sequences:
        return np.empty((0, seq_len, len(feature_cols)), dtype=float)
    return np.stack(sequences, axis=0)

def fit(
    self,
    df_features: pd.DataFrame,
    normalizer: NormalizationStats,
    batch_size: int = 64,
    epochs: int = 20,
    validation_split: float = 0.1,
) -> None:
    """
    Train the LSTM autoencoder on normal data.
    """
    self.normalizer = normalizer

    # normalize df
    X_norm = normalizer.normalize_df(df_features, self.feature_cols)
    df_norm = df_features.copy()
    for i, col in enumerate(self.feature_cols):
        df_norm[col] = X_norm[:, i]

    # build sequences
    seqs = self._df_to_sequences(df_norm, self.feature_cols, self.seq_len)
    if seqs.shape[0] == 0:
        raise ValueError("No sequences could be built from the data.")

    n_features = seqs.shape[2]
    self.model = self._build_model(n_features)
```

```
        self.model.fit(
            seqs,
            seqs,
            epochs=epochs,
            batch_size=batch_size,
            validation_split=validation_split,
            shuffle=True,
            verbose=1,
        )

    def sequence_errors(self, seqs: np.ndarray) -> np.ndarray:
        """
        Given normalized sequences, compute reconstruction errors per sequence.
        """
        if self.model is None:
            raise RuntimeError("Model not trained.")
        recon = self.model.predict(seqs, verbose=0)
        errors = np.mean((seqs - recon) ** 2, axis=(1, 2))
        return errors

    def flight_sequence_errors(
        self,
        df_features: pd.DataFrame,
    ) -> Dict[str, np.ndarray]:
        """
        For each callsign, compute reconstruction error per sliding window.
        Returns dict: callsign -> array of errors.
        """
        if self.normalizer is None:
            raise RuntimeError("Normalizer not set.")
        # normalize df
        X_norm = self.normalizer.normalize_df(df_features, self.feature_cols)
        df_norm = df_features.copy()
        for i, col in enumerate(self.feature_cols):
            df_norm[col] = X_norm[:, i]

        errors_by_flight: Dict[str, np.ndarray] = {}
        for callsign, group in df_norm.groupby("callsign"):
            group = group.sort_values("timestamp")
            X = group[self.feature_cols].to_numpy(dtype=float)
            if len(X) < self.seq_len:
                errors_by_flight[callsign] = np.array([])
```

```
            continue
        seqs = []
        for i in range(len(X) - self.seq_len + 1):
            seqs.append(X[i : i + self.seq_len])
        seqs = np.stack(seqs, axis=0)
        errs = self.sequence_errors(seqs)
        errors_by_flight[callsign] = errs
    return errors_by_flight

def save(self, model_path: str, normalizer_path: str) -> None:
    if self.model is None or self.normalizer is None:
        raise RuntimeError("Model or normalizer not set.")
    self.model.save(model_path)
    self.normalizer.save(normalizer_path)

def load(self, model_path: str, normalizer_path: str) -> None:
    self.model = tf.keras.models.load_model(model_path)
    self.normalizer = NormalizationStats.load(normalizer_path)
```

🧬 How it all connects (typical training flow)

python

```
# train_pipeline_example.py
import pandas as pd

from features import FlightFeatureExtractor
from normalization import NormalizationStats
from first_layer import FirstLayerAnomalyModel
from second_layer_lstm import LSTMSequenceAnomalyModel

# 1. Load historical CSVs (1 year)
df = pd.read_csv("all_israel_flights.csv")

# 2. Extract features
feat_extractor = FlightFeatureExtractor()
df_feat = feat_extractor.add_features(df)

feature_cols = feat_extractor.default_feature_columns()

# 3. Build normalization from historical data
```

```
normalizer = NormalizationStats.from_dataframe(df_feat, feature_cols, percentile=99.0)
normalizer.save("normalizers.json")

# 4. Train first layer (IsolationForest)
first_layer = FirstLayerAnomalyModel(feature_cols=feature_cols, contamination=0.01)
first_layer.fit(df_feat, normalizer)
first_layer.save("iforest_model.joblib", "normalizers.json")

# 5. Train second layer (LSTM autoencoder)
lstm_model = LSTMSequenceAnomalyModel(feature_cols=feature_cols, seq_len=30,
latent_dim=32)
lstm_model.fit(df_feat, normalizer, batch_size=64, epochs=20)
lstm_model.save("lstm_autoencoder.h5", "normalizers.json")
```

🛰️ Real-time scoring sketch

In your realtime service:

python

```
# realtime_example.py
import pandas as pd
from features import FlightFeatureExtractor
from normalization import NormalizationStats
from first_layer import FirstLayerAnomalyModel
from second_layer_lstm import LSTMSequenceAnomalyModel

# Load models & normalizer once
feat_extractor = FlightFeatureExtractor()
feature_cols = feat_extractor.default_feature_columns()

normalizer = NormalizationStats.load("normalizers.json")

first_layer = FirstLayerAnomalyModel(feature_cols=feature_cols)
first_layer.load("iforest_model.joblib", "normalizers.json")

lstm_model = LSTMSequenceAnomalyModel(feature_cols=feature_cols, seq_len=30,
latent_dim=32)
lstm_model.load("lstm_autoencoder.h5", "normalizers.json")

def score_realtime_points(df_new_raw: pd.DataFrame):
```

```
    # df_new_raw has the same columns as your CSV example

    df_feat = feat_extractor.add_features(df_new_raw)

    # Point-level scores
    scores_point = first_layer.predict_scores(df_feat)

    # Sequence-level: you'd typically maintain a rolling history per callsign
    # For a simple example on this batch only:
    errors_by_flight = lstm_model.flight_sequence_errors(df_feat)

    return scores_point, errors_by_flight
```

If you want, next step I can:

- Add a **small FastAPI service** around these classes.
- Or adapt this to exactly how your RunPod container is structured (folders, entrypoint, etc.).