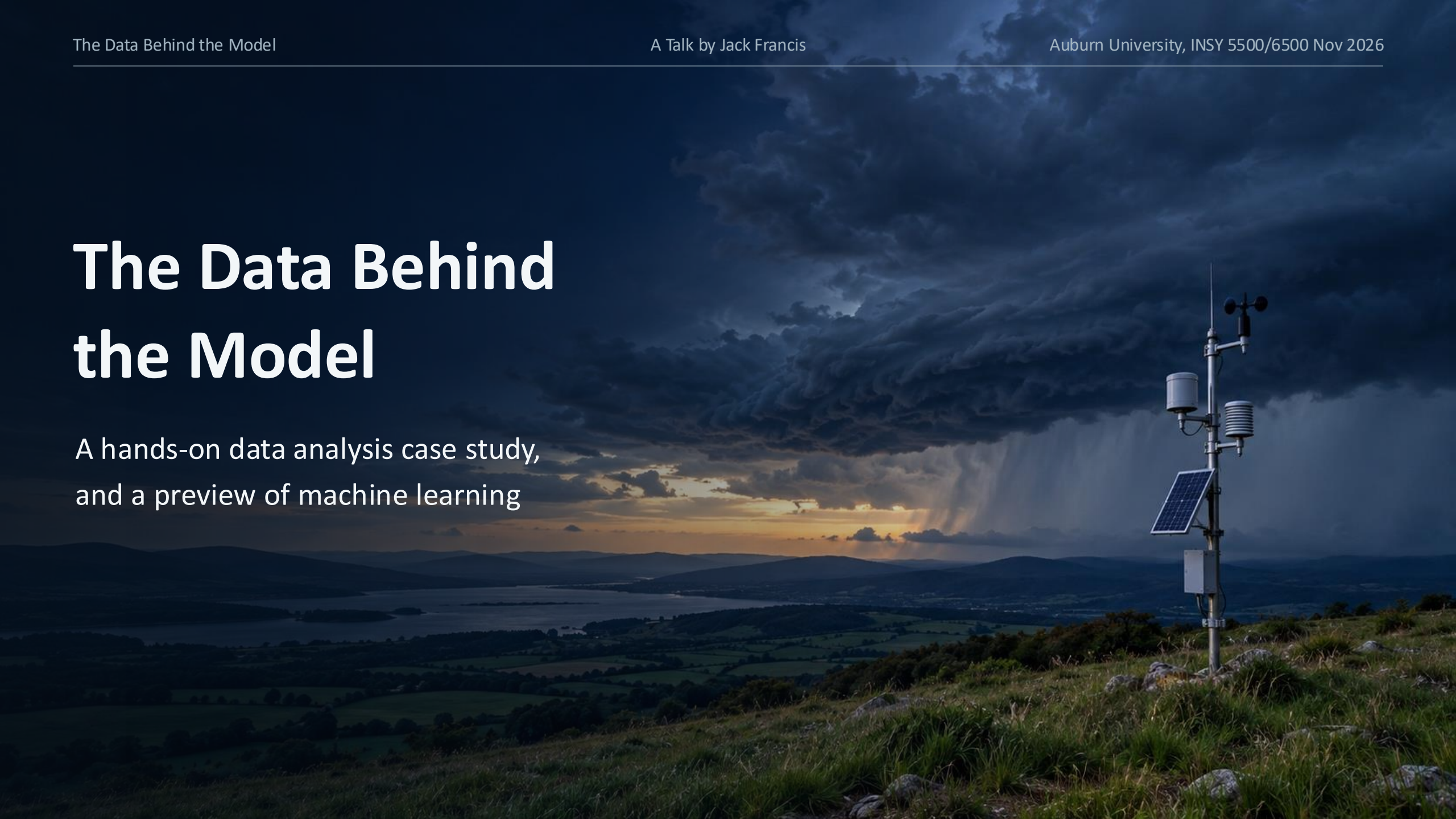
The Data Behind the Model

A hands-on data analysis case study,
and a preview of machine learning



Jack Francis



Blue Ghost Eevee

AI Solutions Director

Davidson Technologies



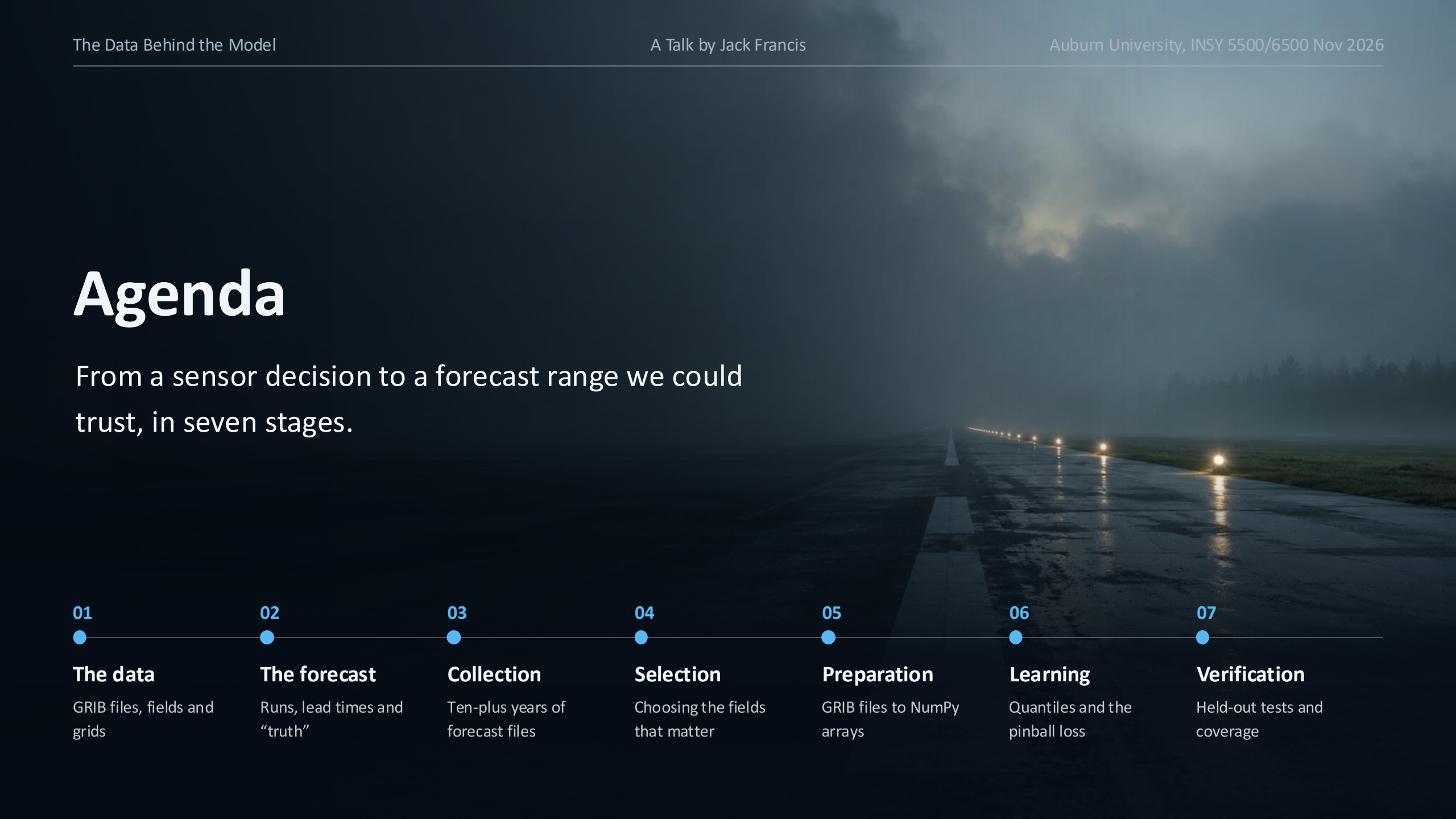
Rome



Tokyo

Agenda

From a sensor decision to a forecast range we could trust, in seven stages.



01 **The data**
GRIB files, fields and grids

02 **The forecast**
Runs, lead times and “truth”

03 **Collection**
Ten-plus years of forecast files

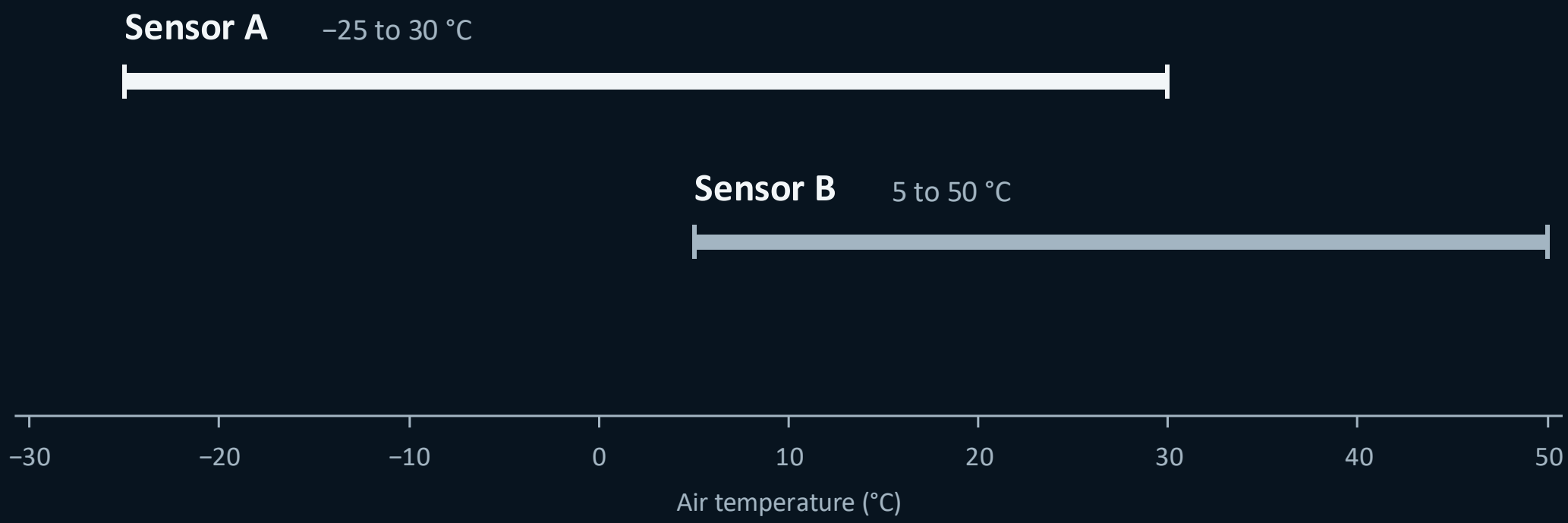
04 **Selection**
Choosing the fields that matter

05 **Preparation**
GRIB files to NumPy arrays

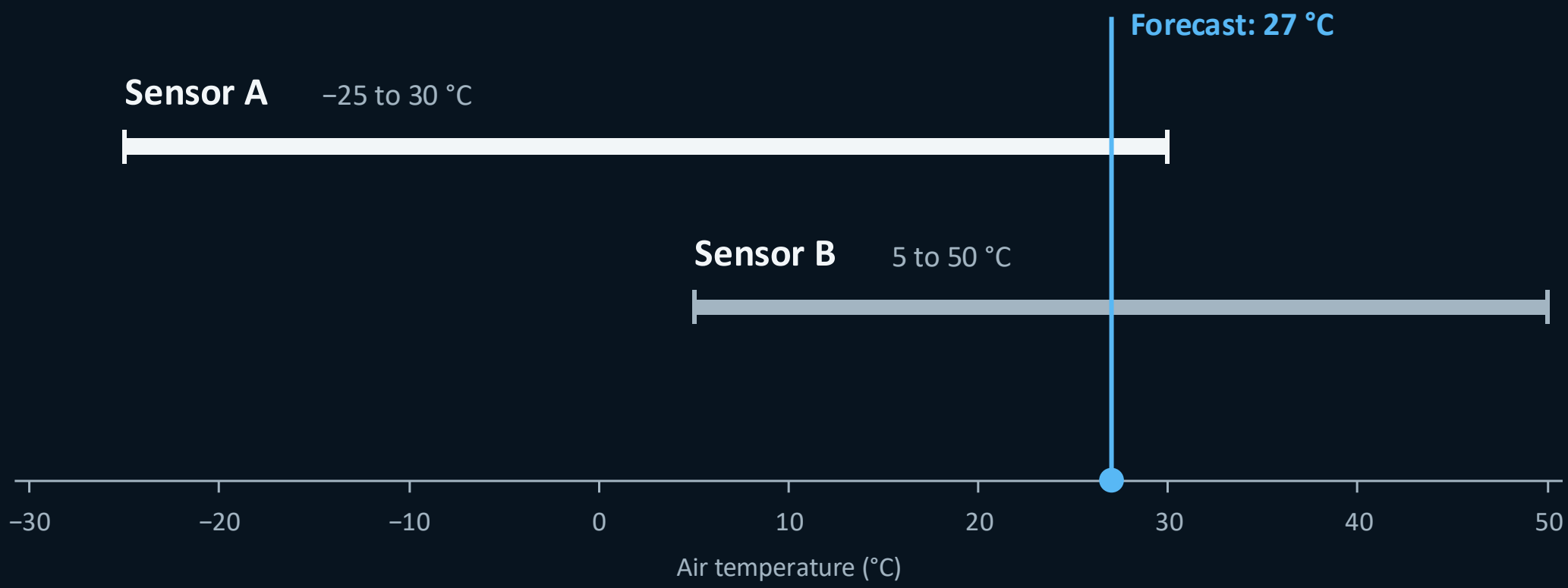
06 **Learning**
Quantiles and the pinball loss

07 **Verification**
Held-out tests and coverage

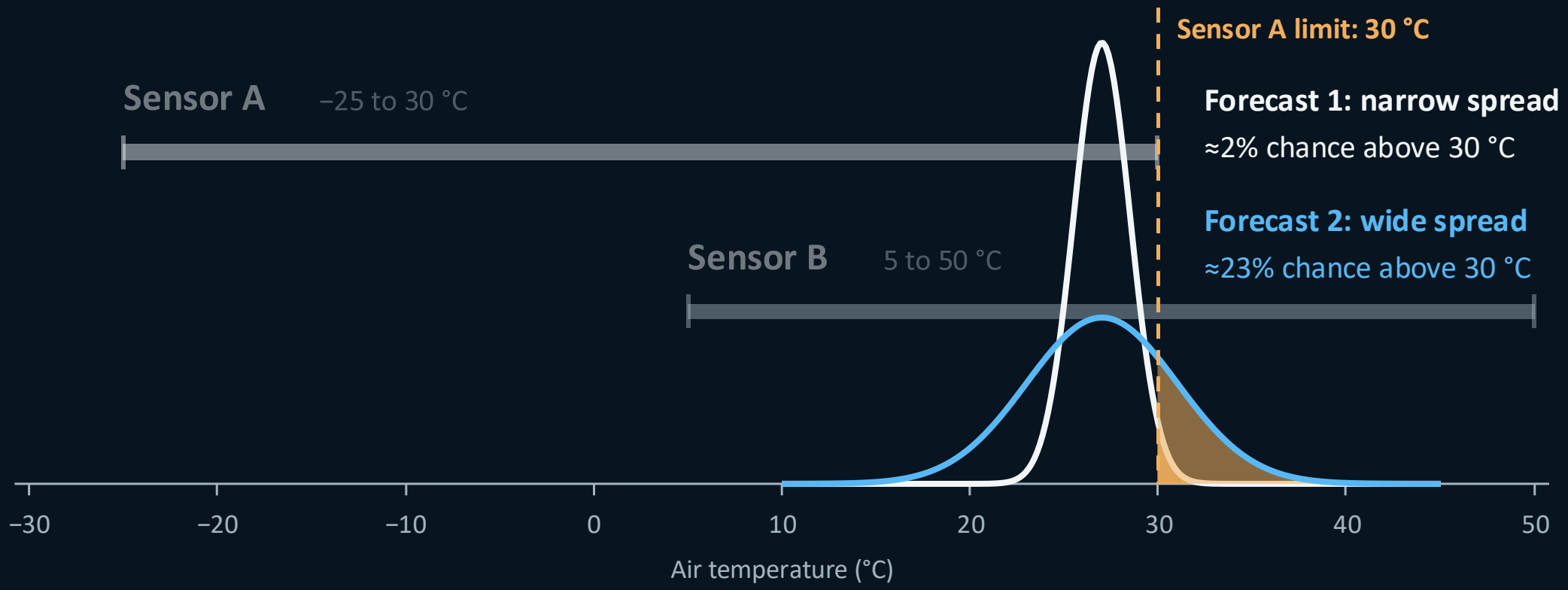
Two sensors, two operating ranges



One predicted value



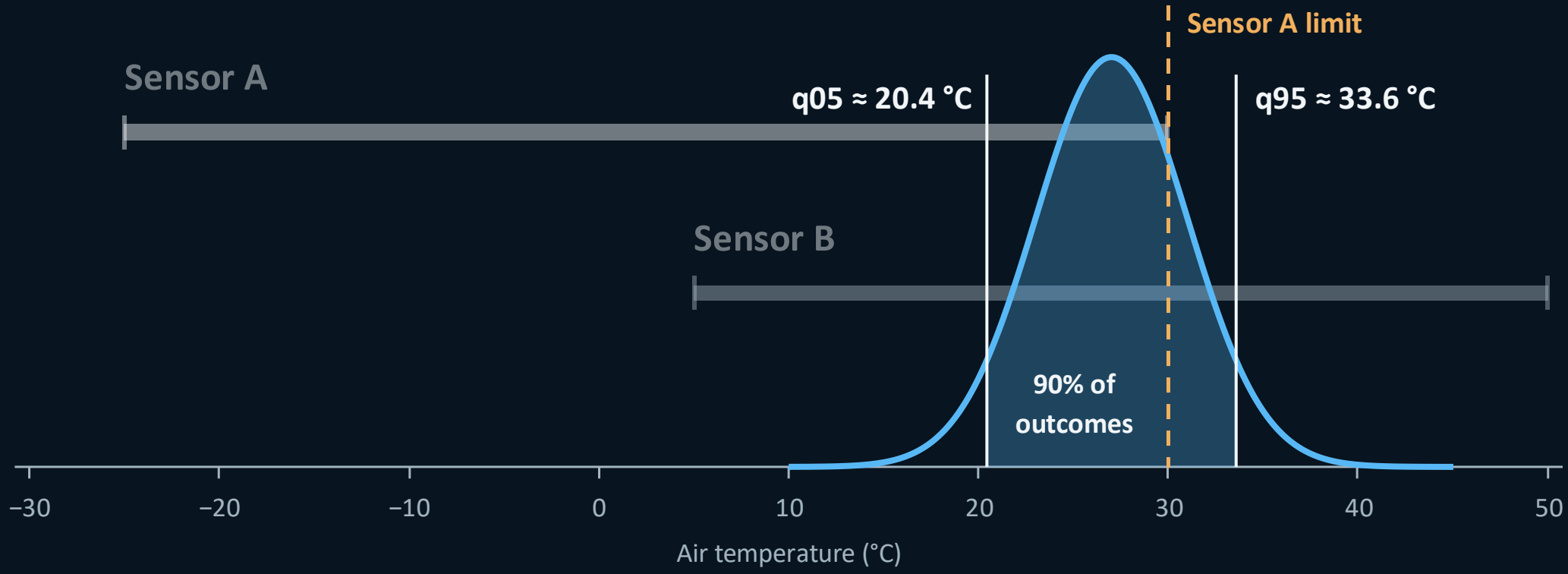
Same forecast, different risk



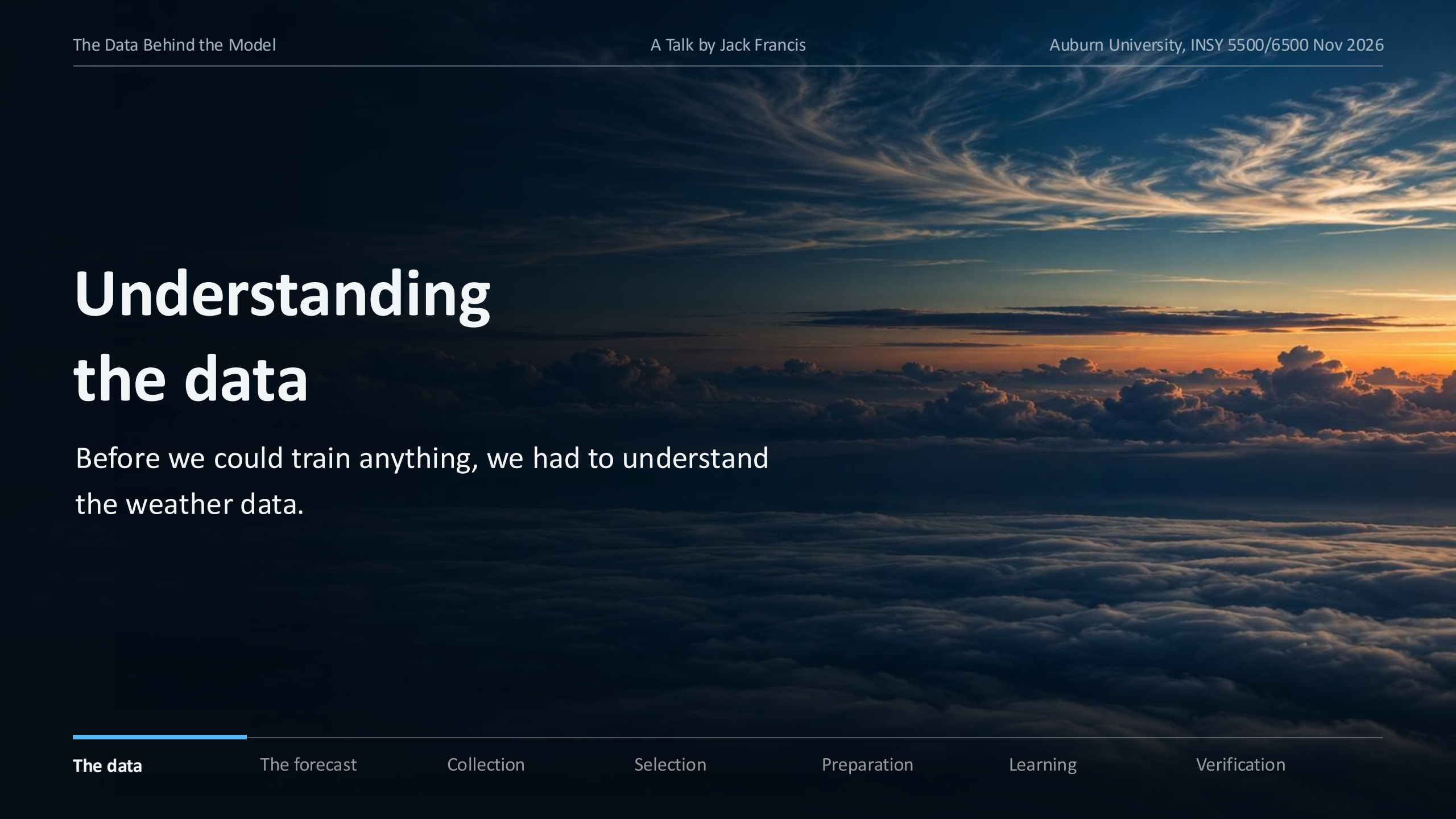
Illustrative normal distributions

A range of plausible conditions

A forecast gives one deterministic value for each place and time. We wanted the 5th and 95th percentiles around it: a range that should contain the outcome 90% of the time.



Two quantiles mark the edges of the range. They do not describe the full shape of the distribution.



Understanding the data

Before we could train anything, we had to understand the weather data.

Inside a GRIB file

A GRIB2 file is a sequence of messages. Each message holds one field: one parameter, at one level, on one grid, for one time.

One public forecast file: 743 messages

⋮

461	HGT 850 mb
462	**TMP 850 mb**
463	RH 850 mb
464	TCDC 850 mb
465	SPFH 850 mb
466	VVEL 850 mb
467	DZDT 850 mb
468	UGRD 850 mb
469	VGRD 850 mb

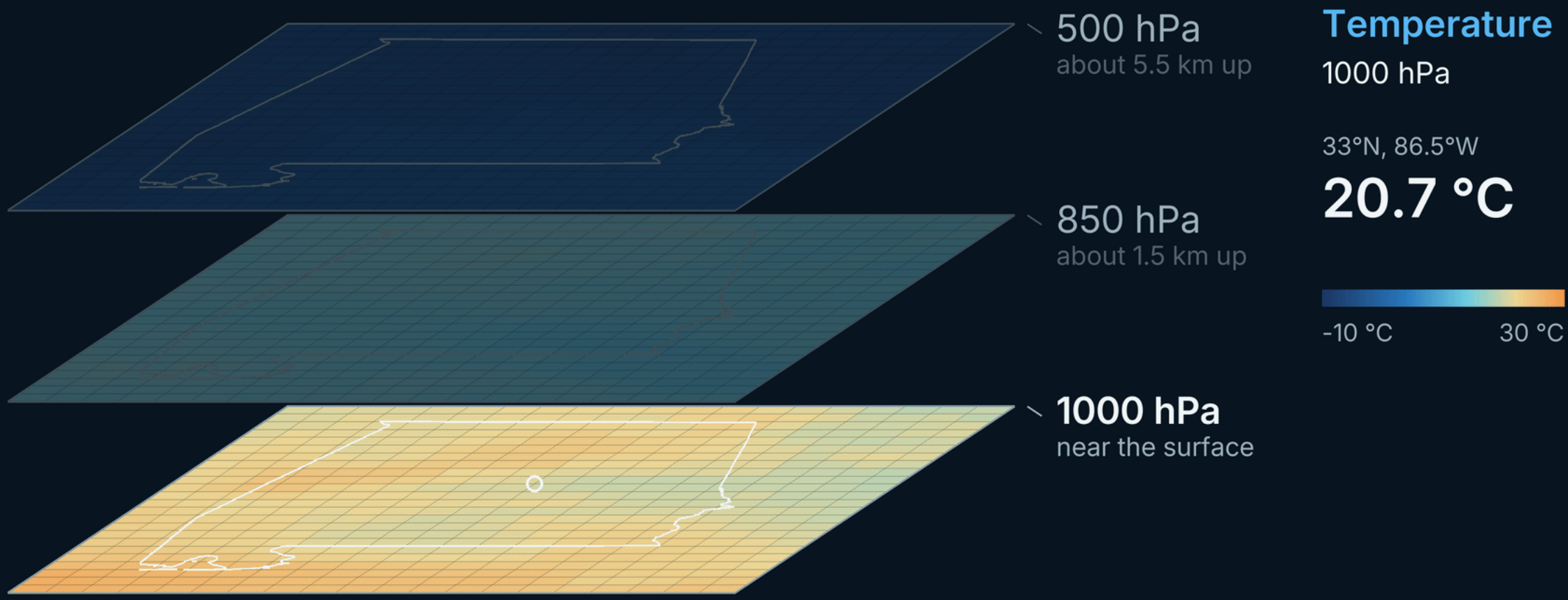
⋮

Message 462

Parameter	Temperature (TMP), in kelvin
Level	850 hPa pressure surface
Grid	0.25° latitude–longitude, 1,440 × 721 points
Time	Run 00 UTC 1 Oct 2025, forecast +6 h
Values	1,038,240 numbers, packed to save space

Fields stacked by pressure level

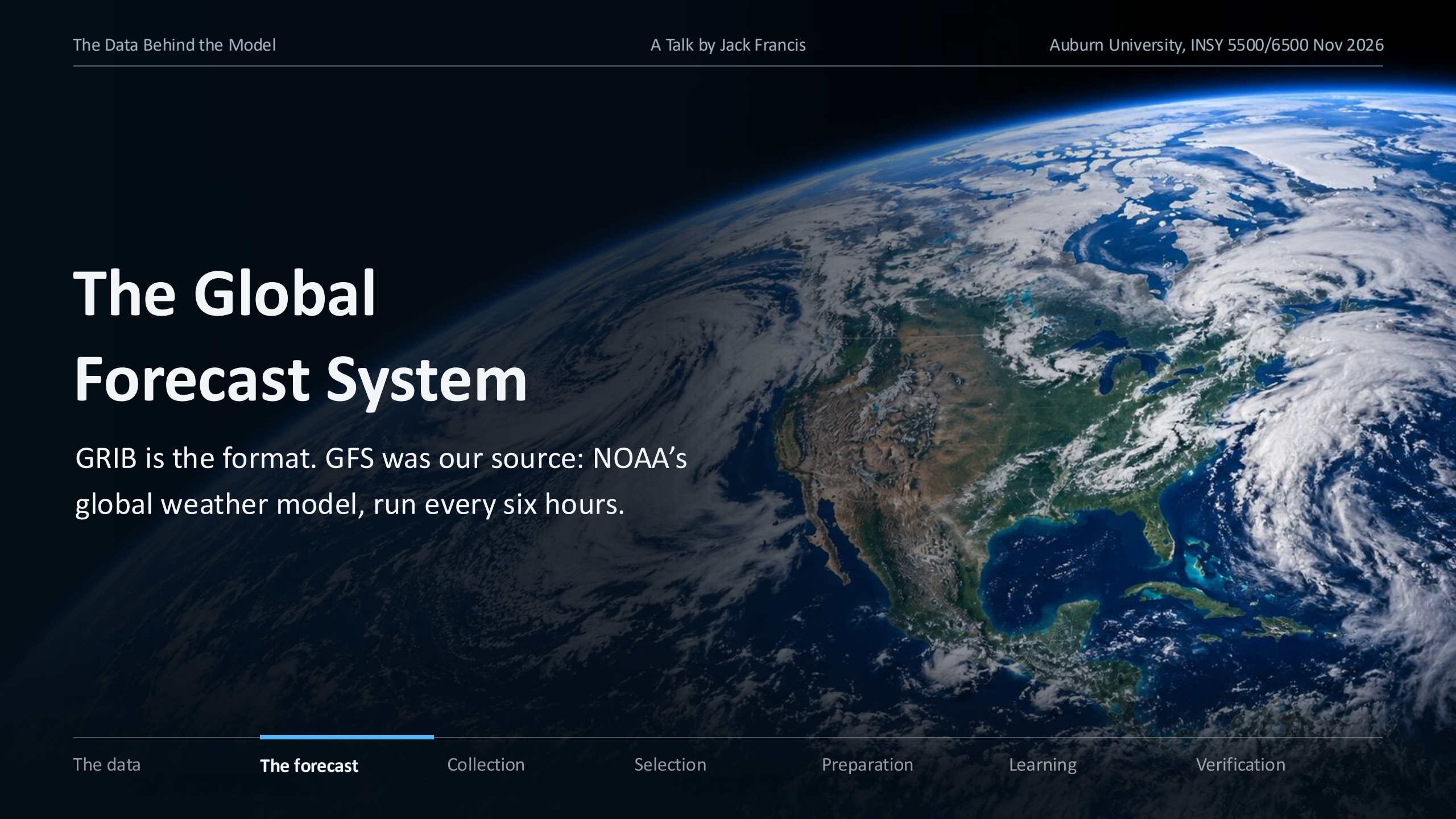
Real forecast values over Alabama: three parameters on three pressure surfaces, all from one file.



0.25° forecast, run 00 UTC 1 Oct 2025, +6 h. Vertical spacing is schematic.

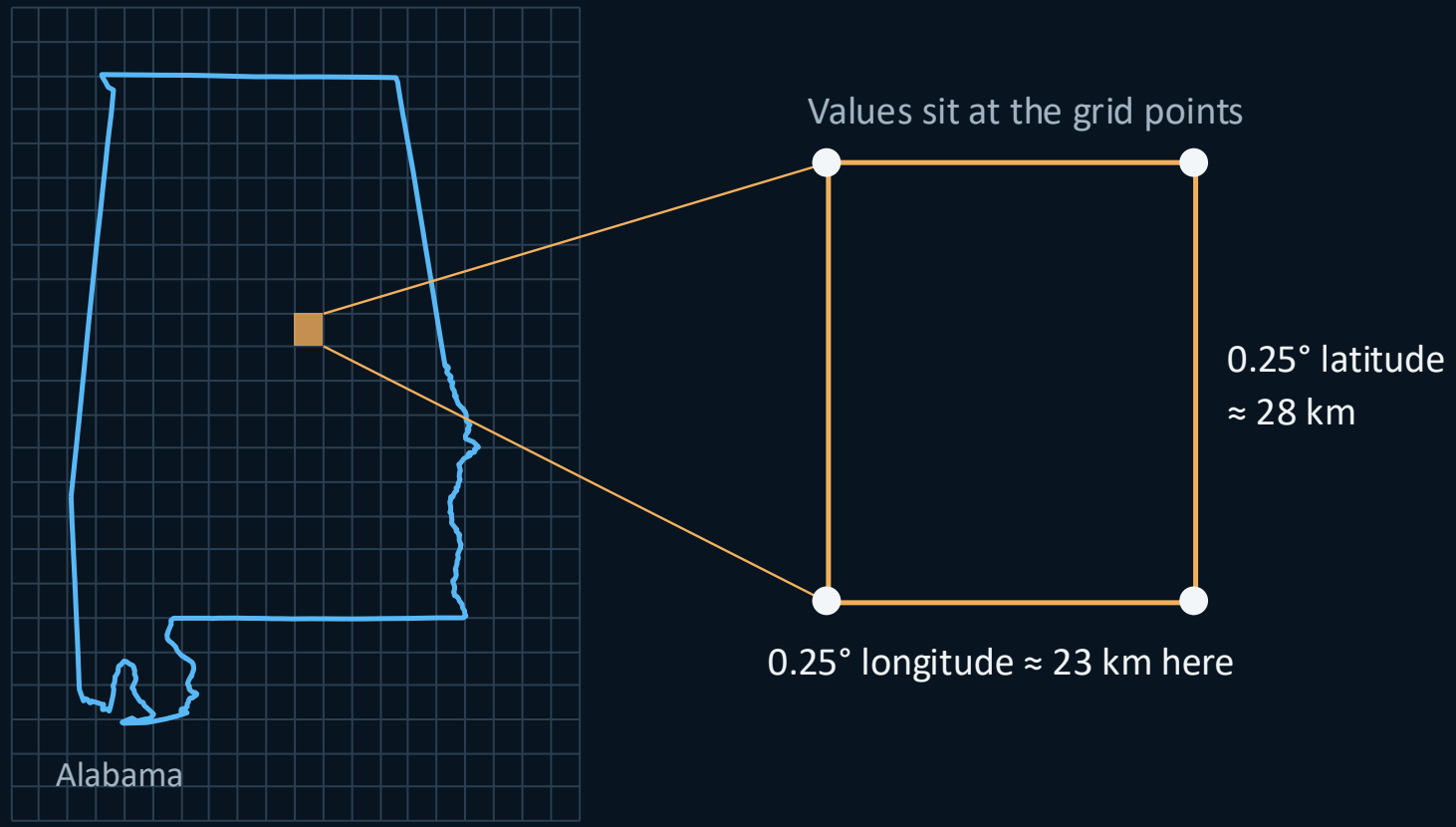
The Global Forecast System

GRIB is the format. GFS was our source: NOAA’s global weather model, run every six hours.



The 0.25° grid

The GFS output we used sits on a regular latitude–longitude grid with a point every quarter degree.



1,440 × 721

points in every global field

= 1,038,240 values

East–west spacing shrinks toward the poles because the grid is in degrees.

Common GFS parameters

The parameter, level and units together define a field: temperature alone appears at 57 different levels.

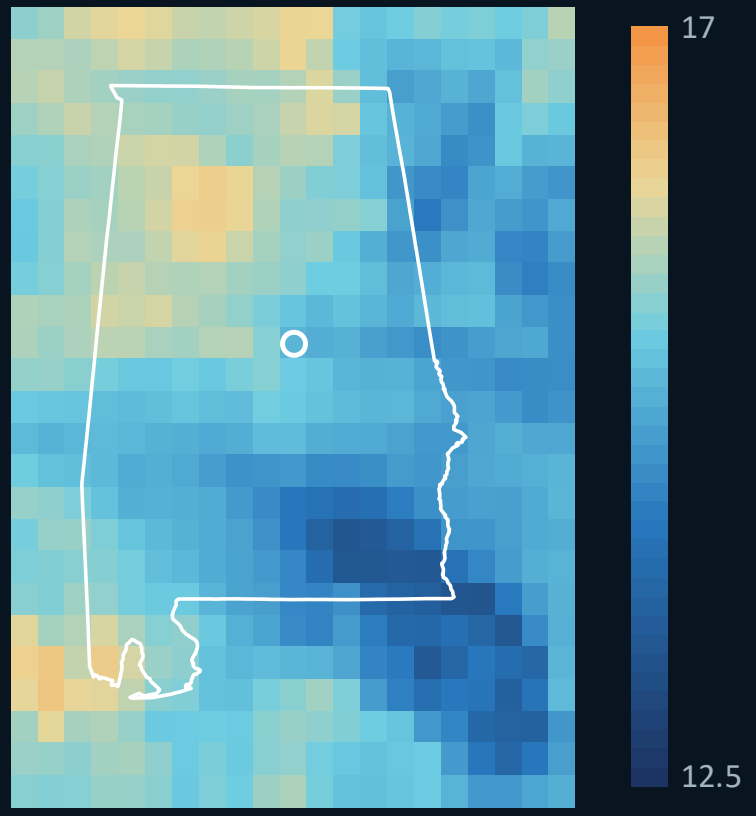
Parameter	GRIB name	Units	Example levels in the file
Temperature	TMP	K	Pressure levels, 2 m above ground
Relative humidity	RH	%	Pressure levels, 2 m above ground
Wind, east and north components	UGRD, VGRD	m/s	Pressure levels, 10 m above ground
Geopotential height	HGT	gpm	Pressure levels
Total cloud cover	TCDC	%	Entire atmosphere, cloud layers
Precipitation	APCP	kg/m²	Surface, accumulated over a period
Visibility	VIS	m	Surface
Convective available potential energy	CAPE	J/kg	Surface, layers above ground
Mean sea level pressure	PRMSL	Pa	Mean sea level

From bytes to an array

A GRIB reader decodes the message metadata and its packed values into a labeled grid of numbers.

```
import xarray as xr
ds = xr.open_dataset("tmp_850mb.grib2",
                     engine="cfgrib")
temp_c = ds["t"] - 273.15
temp_c.shape
(721, 1440)
temp_c.sel(latitude=33, longitude=273.5)
14.9
```

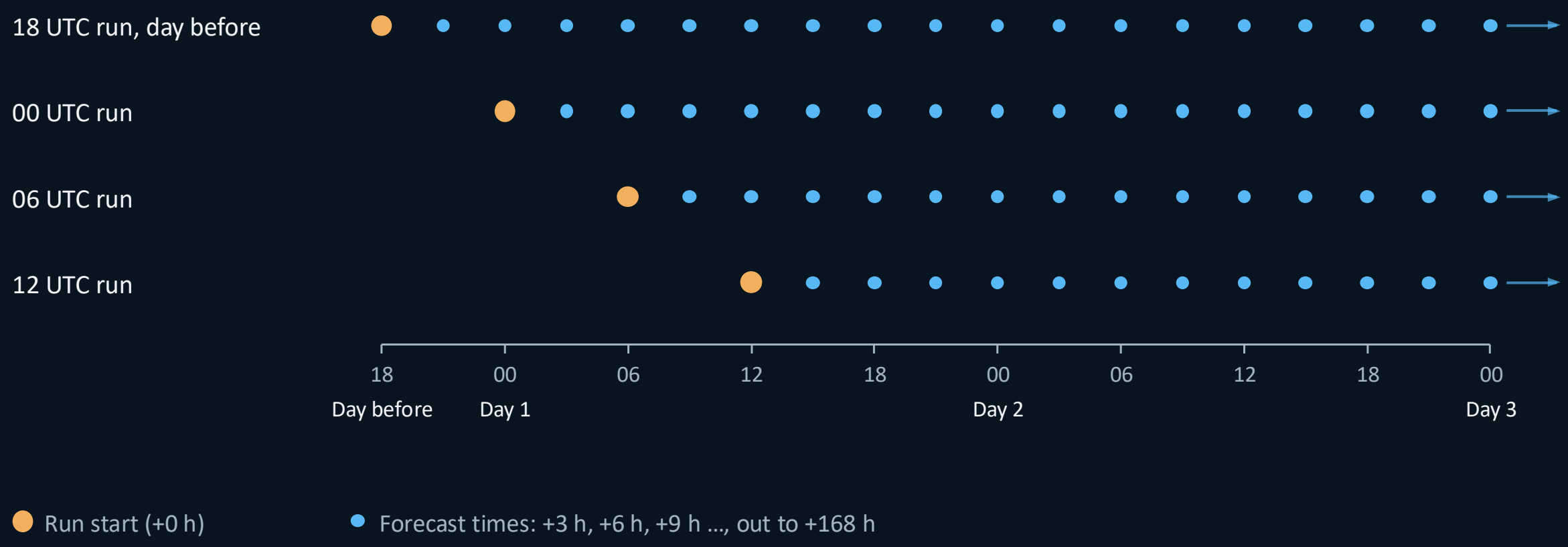
One common Python approach. GFS longitudes run 0–360°, so 86.5°W is 273.5°.



TMP at 850 hPa, °C

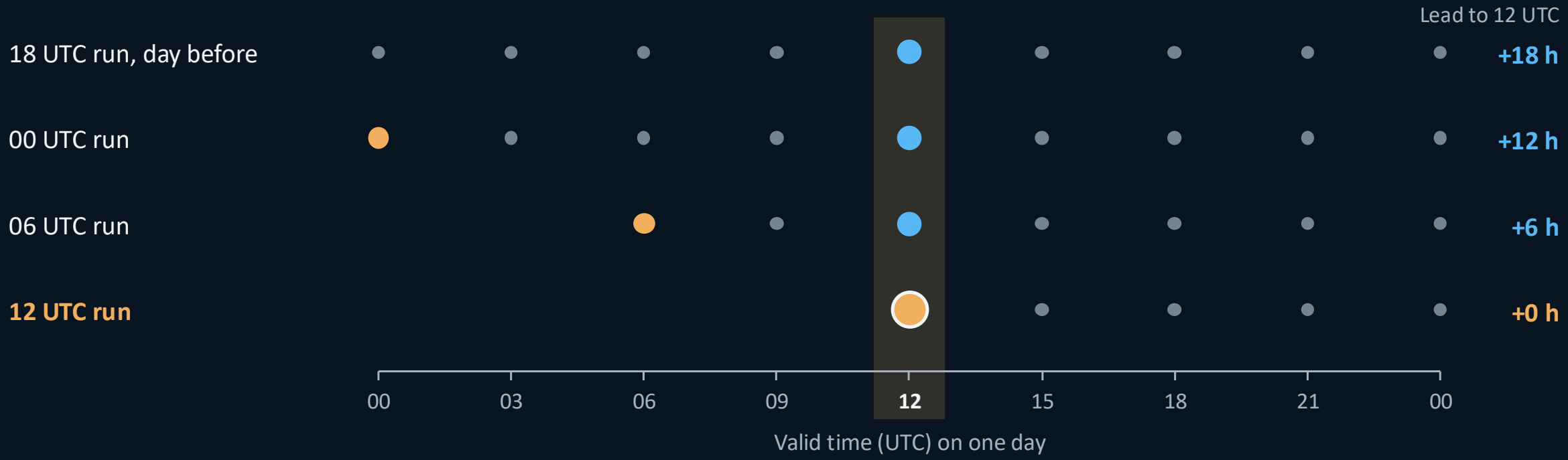
A new run every six hours

GFS starts a new model run at 00, 06, 12 and 18 UTC. Each run forecasts forward in 3-hour steps, out to 168 hours.



Many forecasts, one valid time

Earlier runs forecast 12 UTC at different lead times. The 12 UTC run's own first field is our reference for all of them.



+0 h = our "truth": the run starts from the latest satellite, balloon, station and aircraft observations.

Every earlier forecast is scored against the truth at its valid time.

Alignment is one join

Compute each forecast’s valid time, then join it to the +0 h field for that same time.

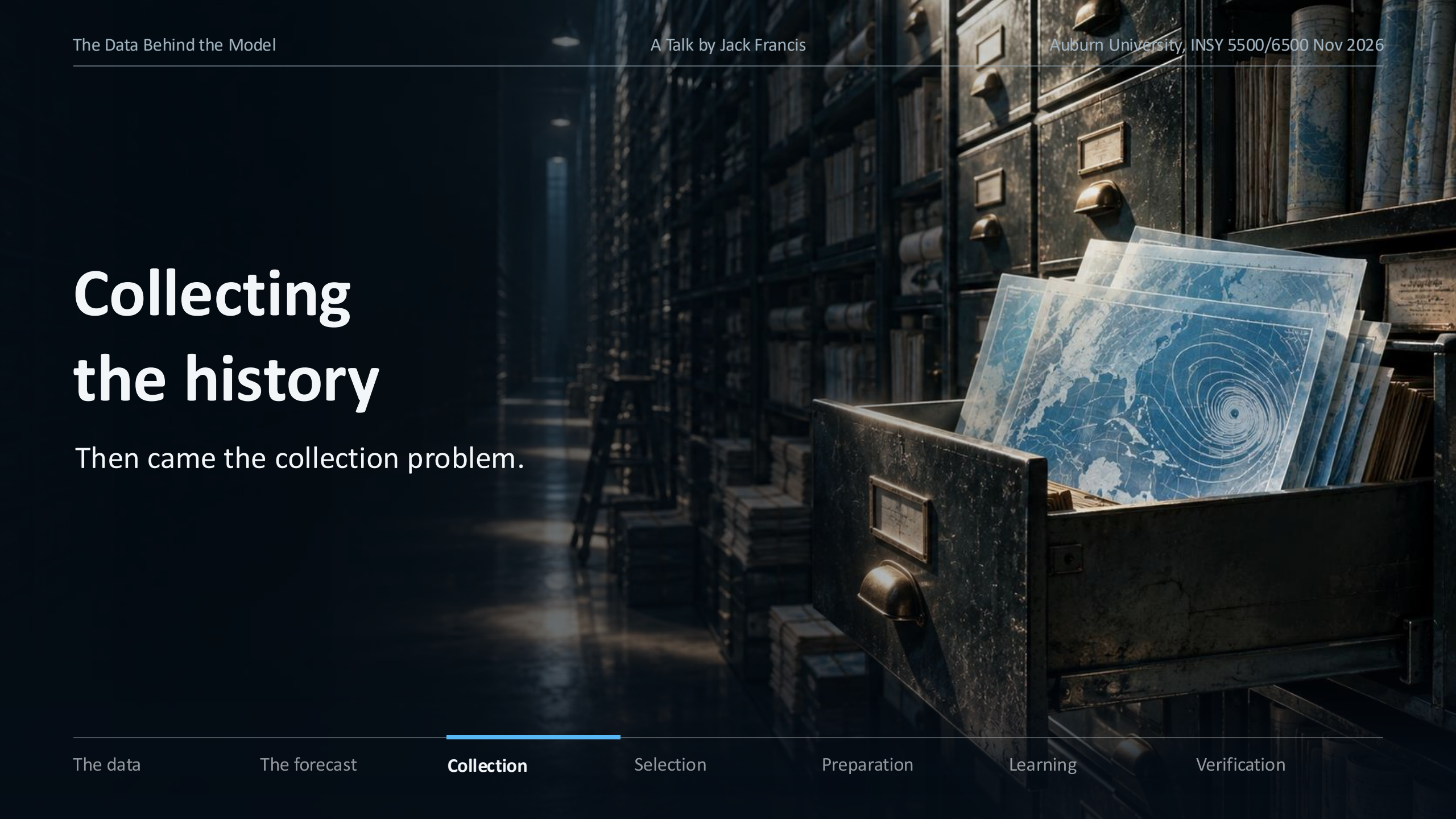
```
fc["valid_time"] = fc["run_time"] + pd.to_timedelta(fc["lead_h"], unit="h")
truth = fc[fc["lead_h"] == 0]
pairs = fc[fc["lead_h"] > 0].merge(truth[["valid_time", "t2m"]],
                                   on="valid_time", suffixes=("", "_truth"))
pairs["error"] = pairs["t2m"] - pairs["t2m_truth"]
```

Run	Lead	Forecast (°C)	Truth (°C)	Error (°C)
30 Sep, 18 UTC	+18 h	18.83	19.13	**−0.30**
1 Oct, 00 UTC	+12 h	19.65	19.13	**+0.52**
1 Oct, 06 UTC	+6 h	19.58	19.13	**+0.45**

Real 2 m temperature at 33°N, 86.5°W, valid 12 UTC 1 Oct 2025.

Collecting the history

Then came the collection problem.



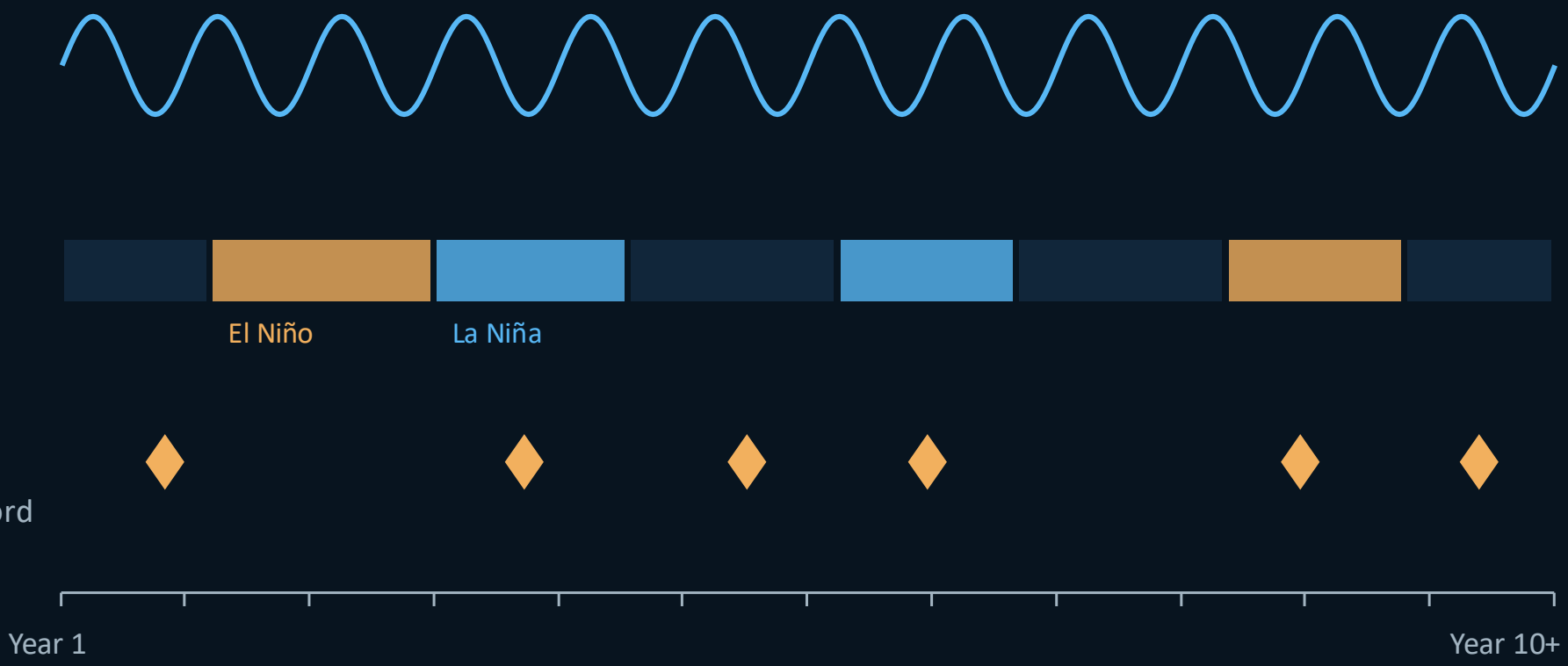
More than ten years of history

A useful training set needed many seasons, different large-scale weather patterns and enough rare, high-impact events.

Seasons
Every year

El Niño and La Niña
Irregular, every 2–7 years

High-impact events
Rare, so they need a long record



Schematic: phases and events are not actual dates.

How much storage?

Collecting whole files grows quickly.

Assumptions

0.5 GB per forecast file

57 files per run: +0 to +168 h, every 3 h

4 runs per day

114 GB per day

How much storage?

Collecting whole files grows quickly.

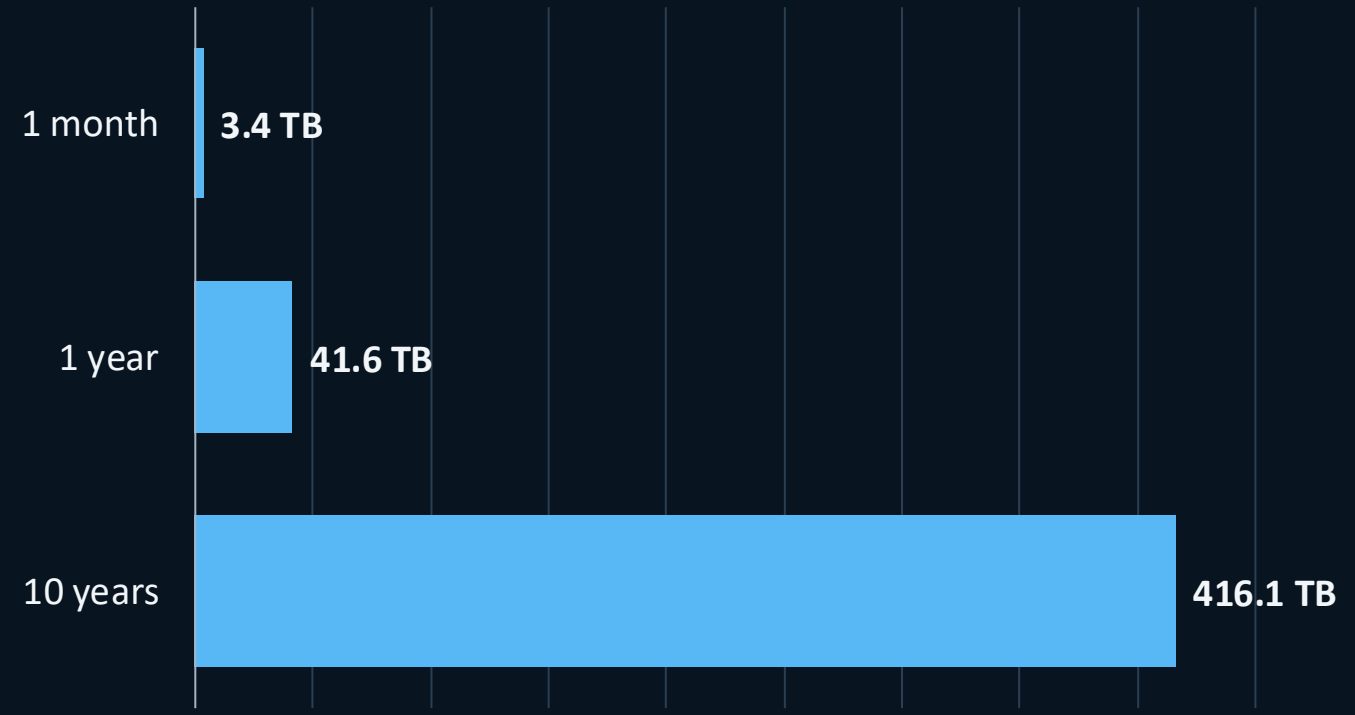
Assumptions

0.5 GB per forecast file

57 files per run: +0 to +168 h, every 3 h

4 runs per day

114 GB per day



Only the fields we needed

Each file has an inventory listing every message and the byte where it starts. Matching parameter and level lets you download just those bytes.

Inventory (.idx), real excerpt

```
461:335689980:...:HGT:850 mb:6 hour fcst:
462:336588078:...:TMP:850 mb:6 hour fcst:
463:337438321:...:RH:850 mb:6 hour fcst:
464:338319697:...:TCDC:850 mb:6 hour fcst:
465:338814767:...:SPFH:850 mb:6 hour fcst:
466:340075641:...:VVEL:850 mb:6 hour fcst:
467:341302315:...:DZDT:850 mb:6 hour fcst:
468:342475233:...:UGRD:850 mb:6 hour fcst:
469:343422093:...:VGRD:850 mb:6 hour fcst:
```

HTTP range requests

bytes=336588078-337438320

bytes=337438321-338319696

The next line's offset marks where each message ends.

One field, one request

Read the inventory like any other table, then ask the server for only the bytes you need.

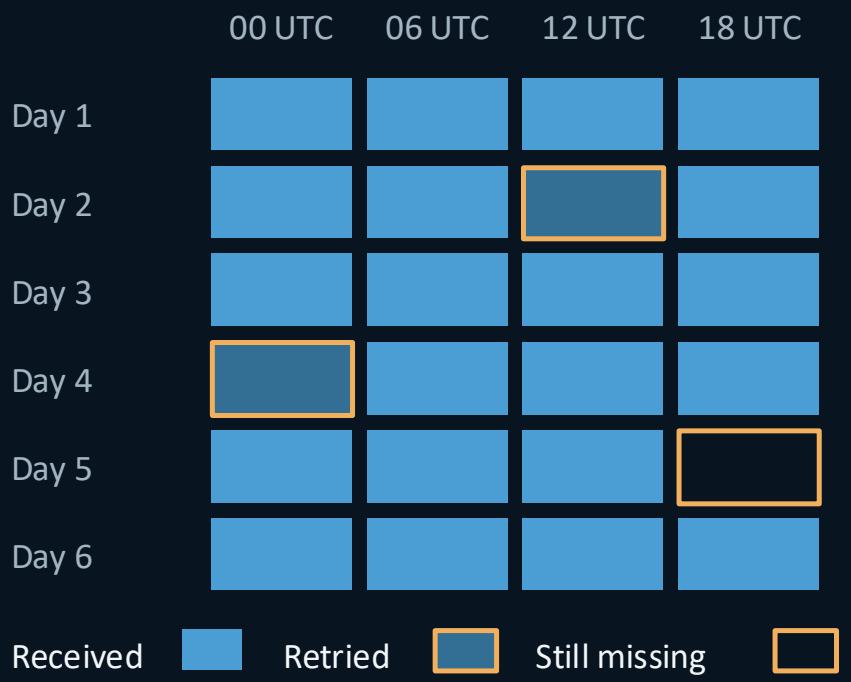
```
idx = pd.read_csv(url + ".idx", sep=":", header=None, usecols=[0, 1, 3, 4],
                  names=["msg", "start", "var", "level"])
idx["end"] = idx["start"].shift(-1) - 1
row = idx.query('var == "TMP" and level == "850 mb"').iloc[0]
wanted = {"Range": f"bytes={row.start}-{int(row.end)}"}
r = requests.get(url, headers=wanted)
len(r.content)
850243
```

0.85 MB

Temperature at 850 hPa, pulled from a 543 MB file with one request.
Real output from the same public file.

Missing files, retries and MinIO

Some downloads failed or arrived late. We compared expected files with received files, retried the gaps and stored the collected GRIB data in MinIO.



collected files

MinIO

Object storage for the collected GRIB files, shared by the rest of the pipeline

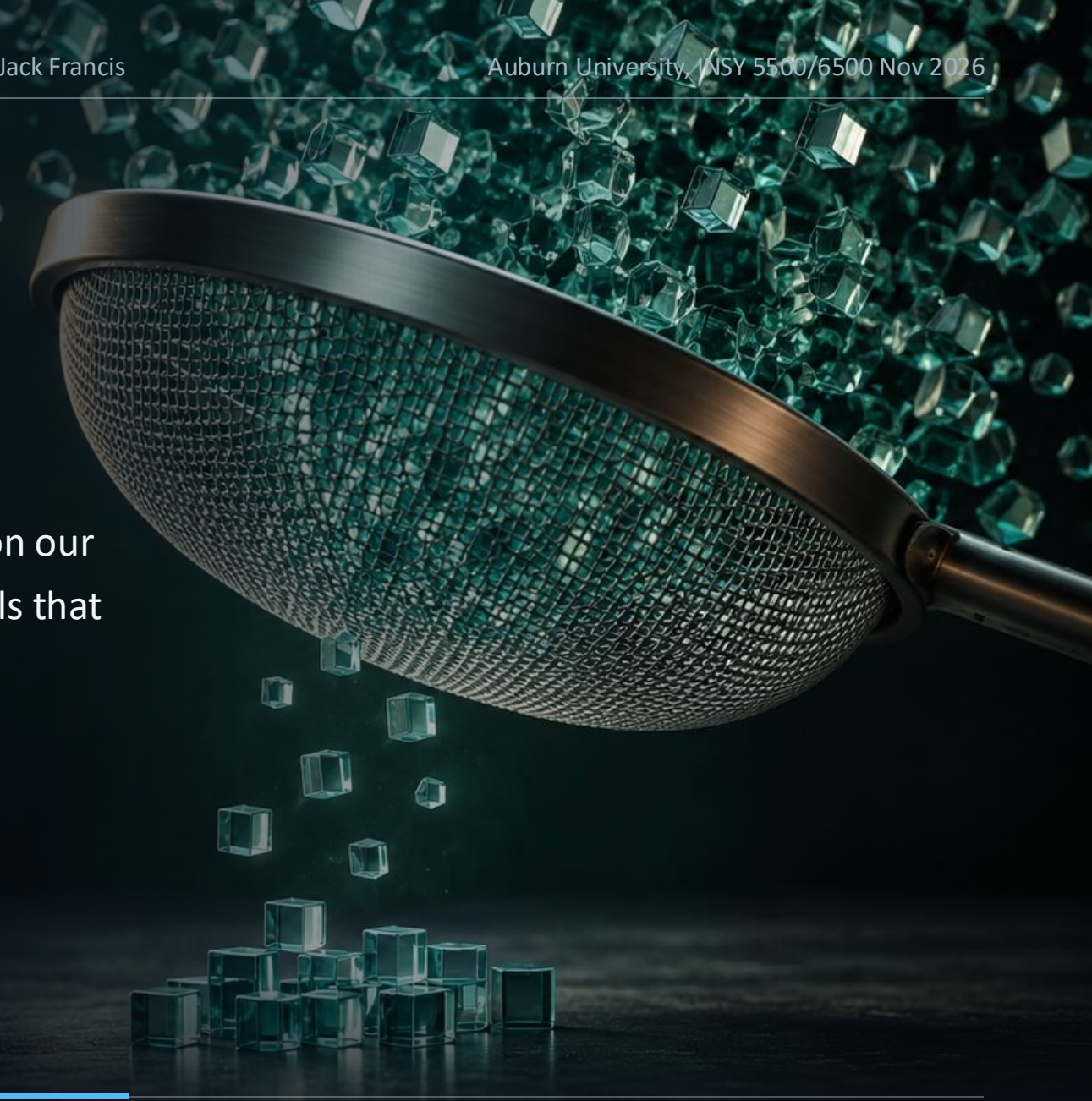
```
expected = pd.date_range(start, end, freq="6h")
missing = expected.difference(received)
len(missing) → 3, then 1 after retries
```

Choosing what mattered

Having the files was only the beginning.

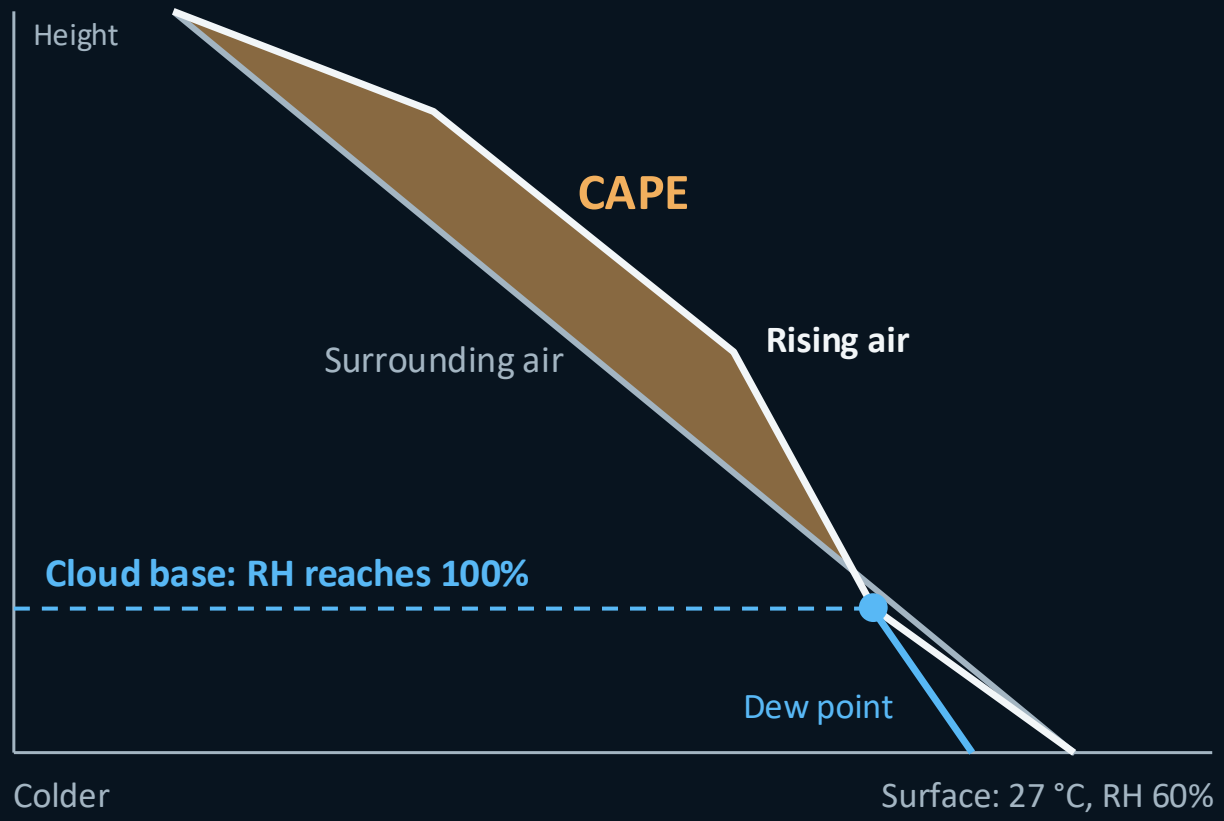
Each GFS file holds hundreds of fields. A meteorologist on our staff helped us narrow them to the parameters and levels that mattered for the sensor decision.

Understanding the domain is part of the analysis.



Reading the atmosphere: CAPE and cloud base

A meteorologist knows which physical quantities carry the signal. Two examples:



CAPE

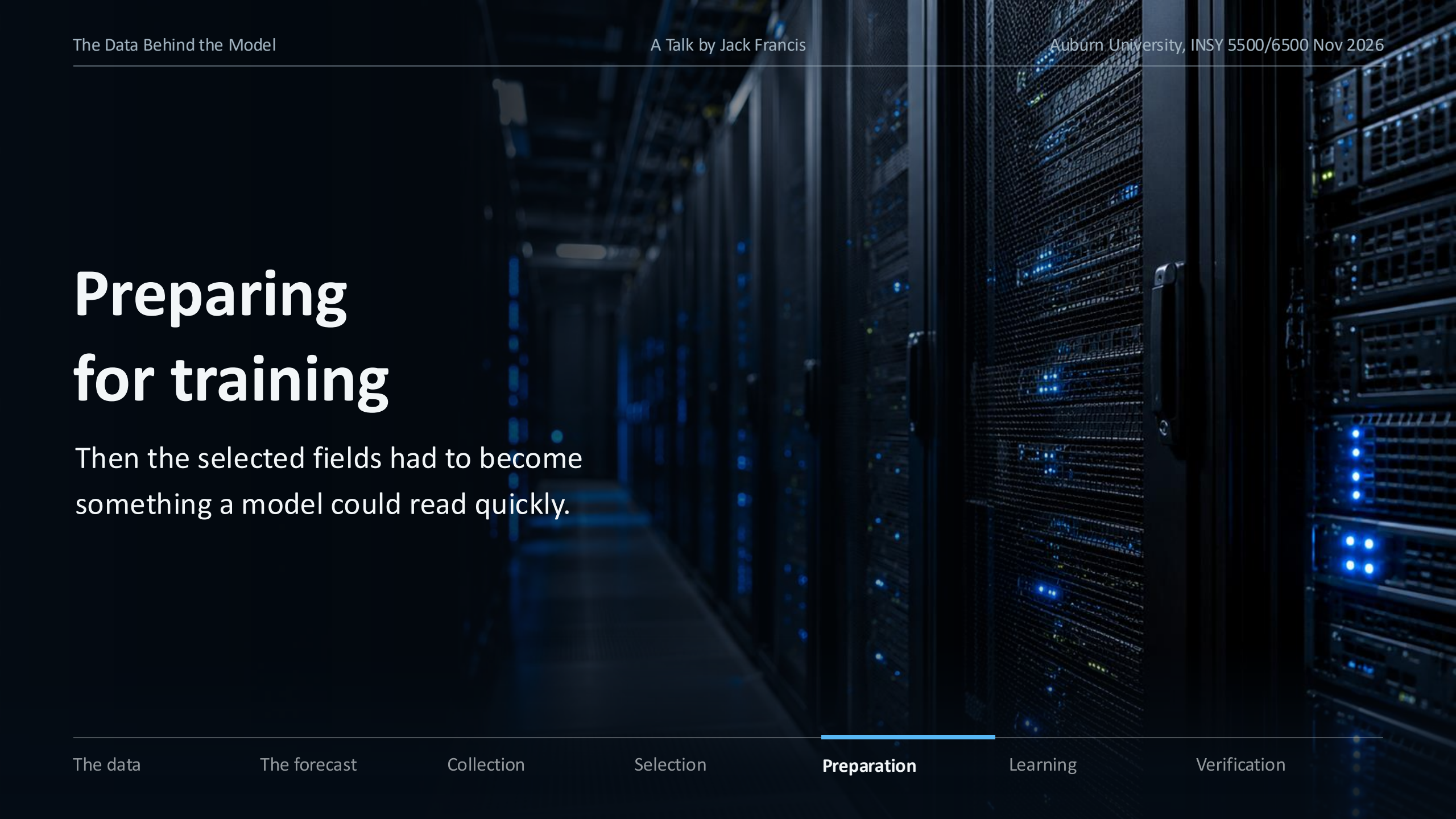
Convective available potential energy, in J/kg: the energy available to rising air. A key ingredient for thunderstorms, not a guarantee.

Cloud base

Rising air cools until relative humidity reaches 100%. Water condenses and cloud can form.

```
def cloud_base_m(t_c, rh):
    return (20 + t_c / 5) * (100 - rh)
cloud_base_m(27, 60)
1016.0
```

Lawrence (2005). Best above about 50% RH.



Preparing for training

Then the selected fields had to become something a model could read quickly.

GRIB to NPZ, MinIO to the DGX

We converted the selected GRIB data into NPZ files for training and moved those files to our DGX system.



MinIO
Collected GRIB files

Convert
Decode the selected fields

NPZ files
NumPy arrays, ready to load

DGX
GPU system for training

Why NPZ

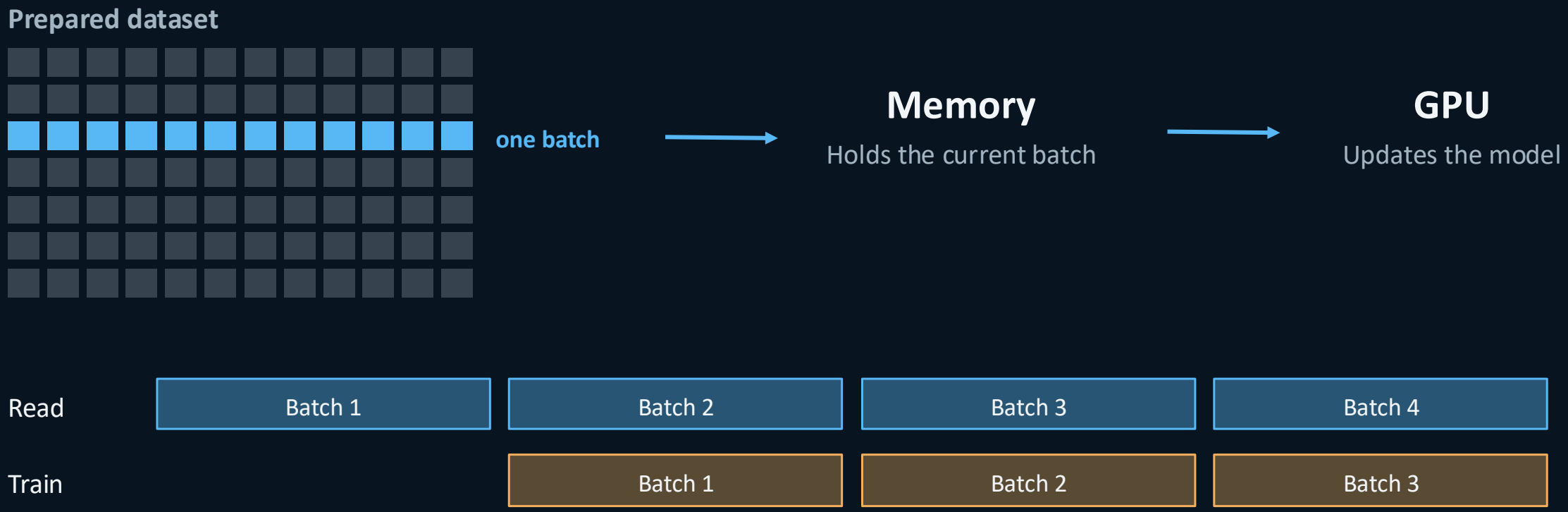
GRIB is built for distributing weather fields. NPZ stores NumPy arrays, so training code can load them without decoding GRIB every time.

What has to travel with the numbers

Units, levels, grid positions, issue time, lead time and valid time. Without them, an array is just numbers.

Training one batch at a time

The dataset was too large to load at once. Training read it in batches, and an efficient loader kept the GPUs supplied.



Reading the next batch while training on the current one keeps the GPUs busy.

The machine-learning connection

What the model learned, why we predicted quantiles, and how the loss shaped the bounds.



One training example

The model uses published forecast inputs, predicts two quantiles, and is scored against the later reference.

Inputs

Selected forecast fields

from GFS, plus the lead time. Available after the forecast is published.



Outputs

q05 and q95

for each predicted parameter



Target

Reference (the +0 h “truth”)

at the valid time, known only later

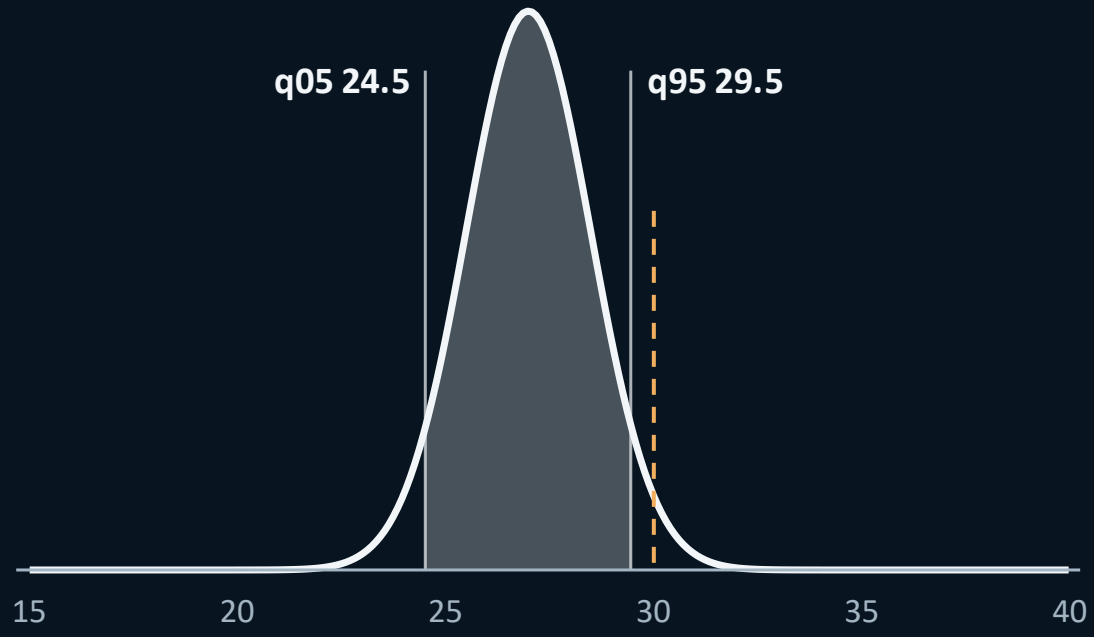
Loss

scores q05 and q95 against the reference

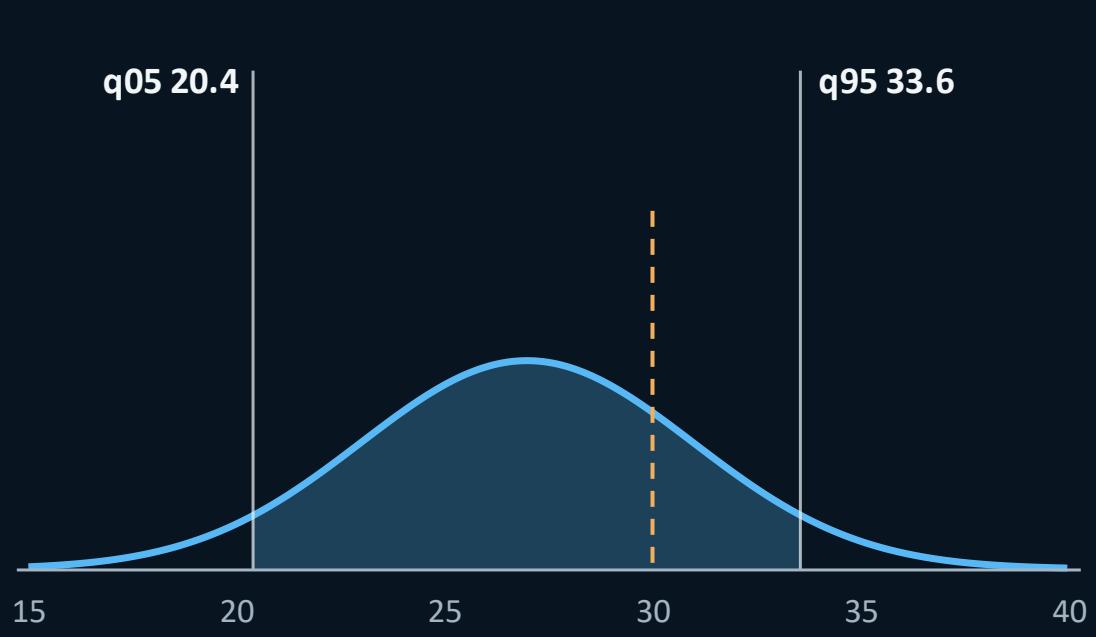
Why predict quantiles

A single prediction can’t say how far reality might stray. Learning two quantiles lets the range widen or narrow forecast by forecast.

Lower uncertainty: range 24.5 to 29.5 °C



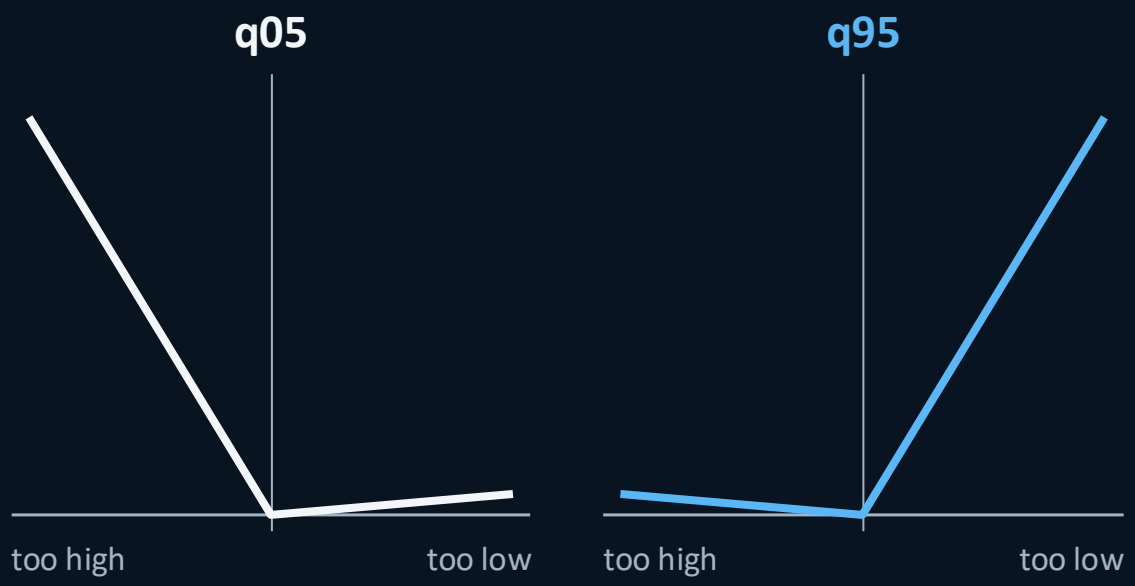
Higher uncertainty: range 20.4 to 33.6 °C



q05: about 5% of outcomes fall below. q95: about 5% fall above. Dashed line: Sensor A limit.

The pinball loss

Quantile loss charges misses on one side more than the other. For q95, predicting too low costs 19 times as much as the same miss on the high side.



Loss against error = reference − prediction

Worked example for q95, reference = 28 °C

q95 prediction	Error	Loss
26 °C, too low	+2	**0.95 × 2 = 1.90**
30 °C, too high	−2	0.05 × 2 = 0.10

loss = τ × error when error ≥ 0

loss = (τ − 1) × error when error < 0

τ is the quantile level: 0.05 or 0.95.

Minimizing the average loss pushes q95 up until only about 5% of references fall above it.

Delivering the model

Training happened on the DGX, MLflow tracked the models, and our GitLab pipeline packaged the selected model with each software release.



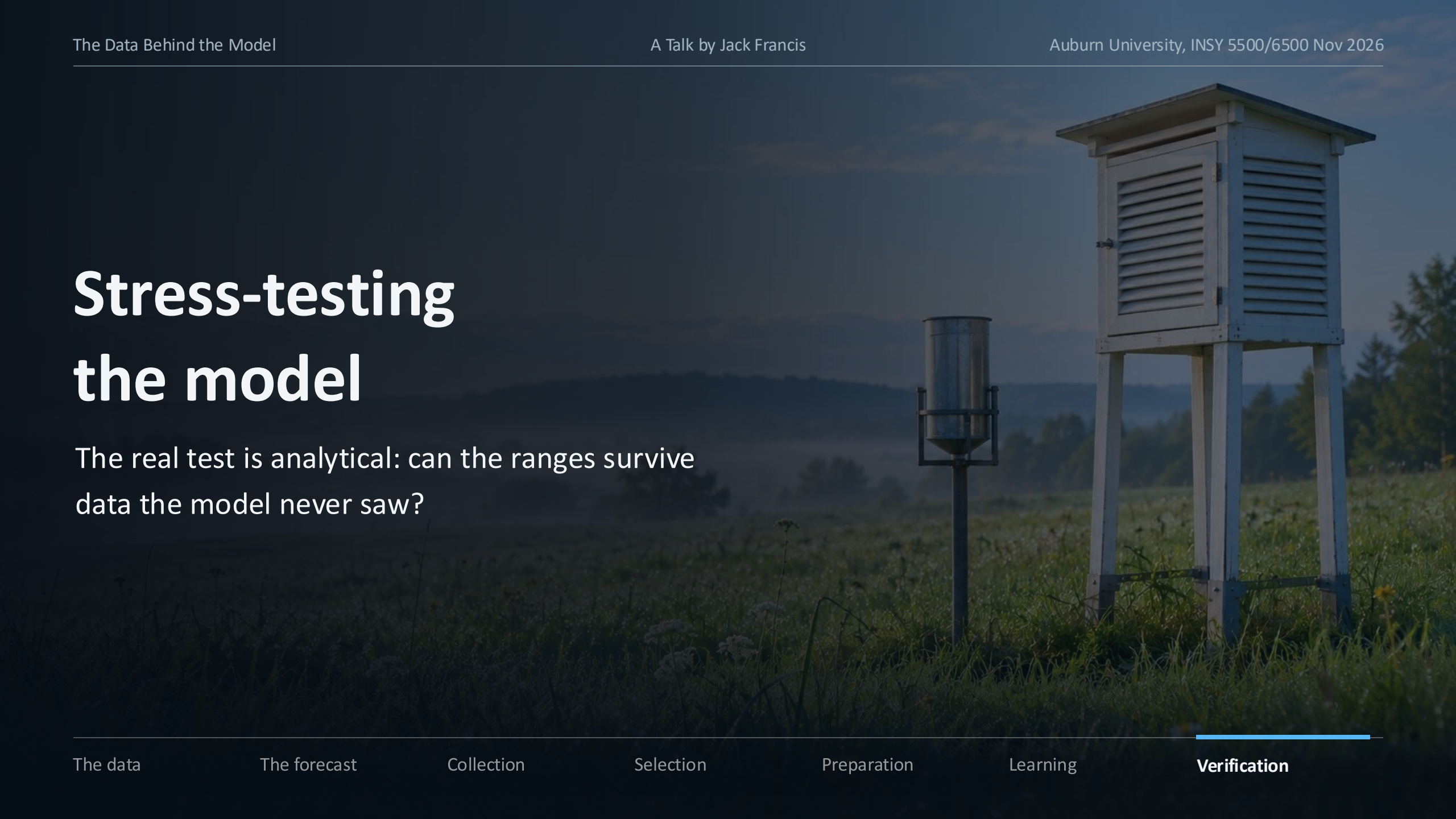
DGX
Train models

MLflow
Track runs and store models

GitLab pipeline
Retrieve the selected model

Release
Package the model with the software

Every release could be traced to a specific trained model.

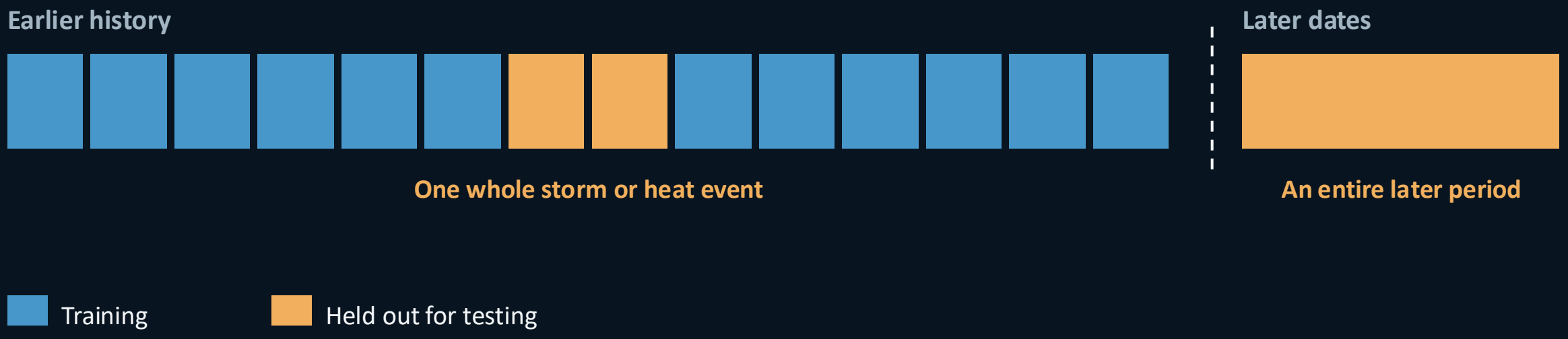


Stress-testing the model

The real test is analytical: can the ranges survive data the model never saw?

Testing on data the model never saw

We evaluated later historical periods and complete weather events held out from training.

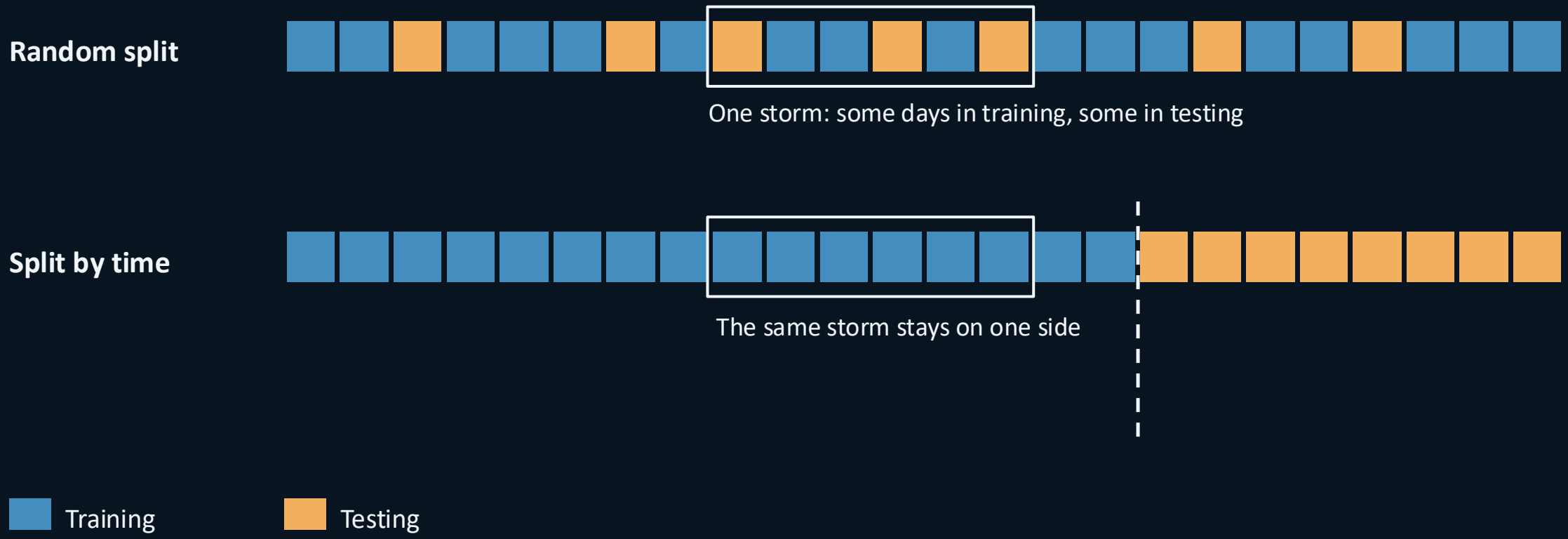


Whole events keep one storm from appearing in both training and testing. Later dates mimic using the model on the future.

Schematic: blocks are not actual dates or events.

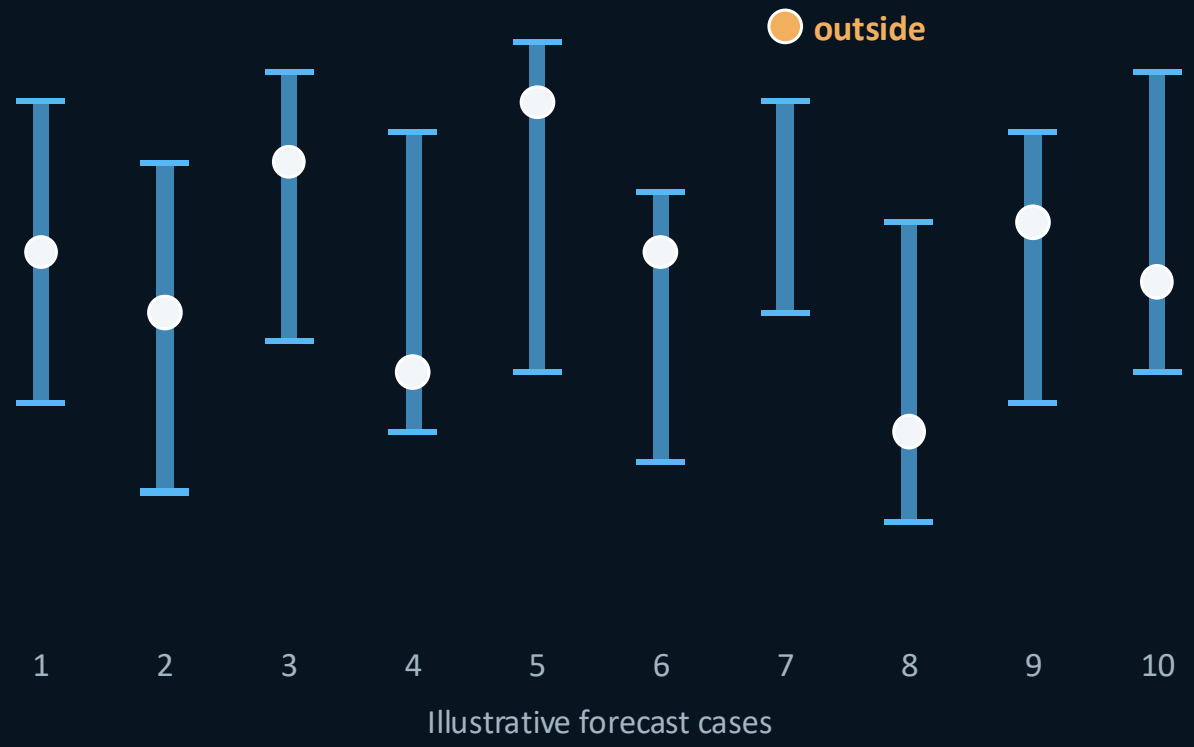
Temporal leakage

Weather persists for days, and each forecast spans time from issue to valid. A random split would let test information leak into training.



Measuring coverage

For each predicted parameter, we examined how often the reference fell between the q05 and q95 predictions.



Bar: q05 to q95. Dot: reference.

Coverage =

$$\frac{\text{cases with } q05 \le \text{reference} \le q95}{\text{all cases}}$$

Here: 9 ÷ 10 = 90%

Computed separately for each parameter and each lead time.

Coverage in a few lines

Two numbers answer two questions: are the ranges honest, and are they narrow enough to use?

```
df["inside"] = df["ref"].between(df["q05"], df["q95"])
df["inside"].mean()
0.9

(df["q95"] - df["q05"]).mean()
9.6

# one number per parameter and lead time
df.groupby(["param", "lead_h"])["inside"].mean()
```

90%

coverage: how often the reference landed inside

9.6 °C

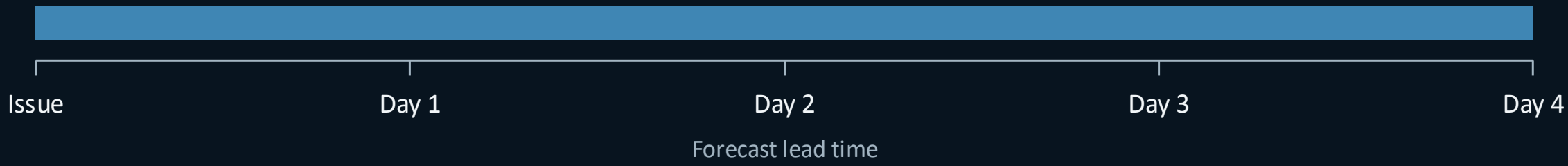
mean width of the range

The same ten illustrative cases as the previous slide.

What we found

>90%

In our held-out evaluation, the reference fell inside the q05–q95 range more than 90% of the time, through four days into the forecast.

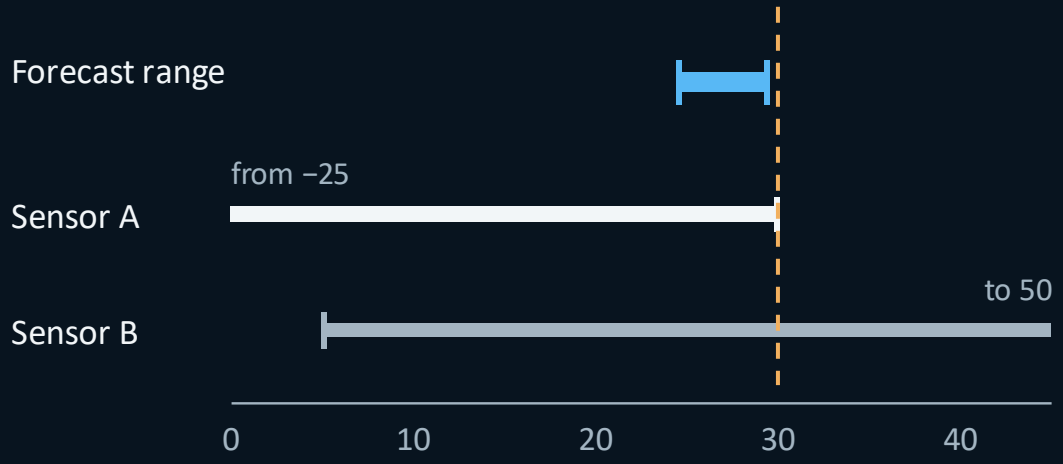


This measures coverage of the range. It does not measure point accuracy.

Narrow enough to help?

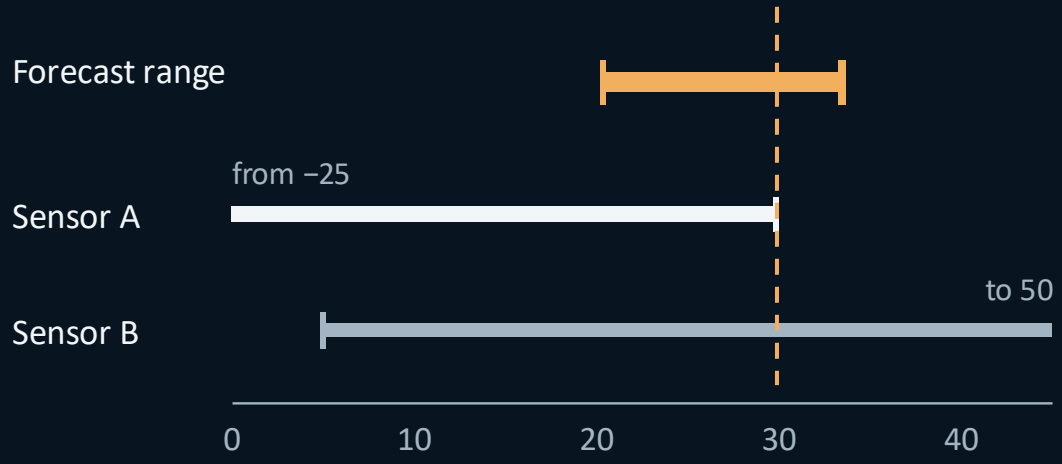
Coverage alone isn’t enough. A range that always contains the truth can still be too wide to choose a sensor.

Narrow range: 24.5 to 29.5 °C



Inside both operating ranges. Either sensor can work.

Wide range: 20.4 to 33.6 °C



Crosses Sensor A’s 30 °C limit. Sensor B is the safer choice.

Both ranges could have 90% coverage. A range describes plausible conditions. It cannot guarantee that a sensor stays within limits. Ranges in °C are illustrative.

Building the model was one part of the project



Model training was one step. Data analysis was the major part.

Understanding, collecting, selecting, aligning, moving and verifying the data made the result possible.

Three places data analysis carried the project



1

Understanding the data

Decoding the files and working with a meteorologist to choose what mattered

cfgrib, then .sel()



2

Building a trustworthy dataset

Aligning every forecast to its truth value and accounting for what was missing

merge(on="valid_time")



3

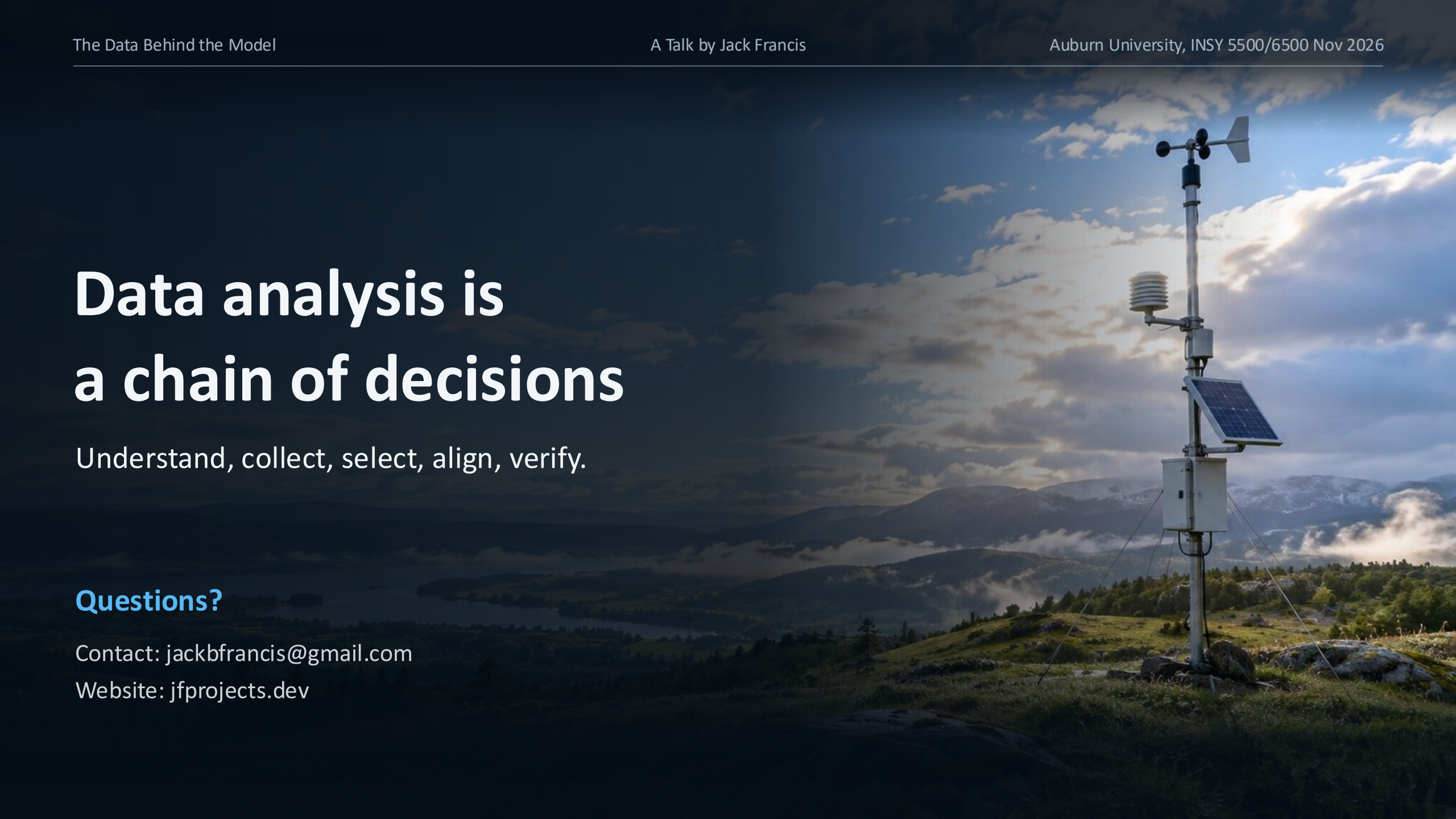
Verifying the result

Testing on held-out data, measuring coverage and asking whether the range helps

.between(q05, q95)

Data analysis is a chain of decisions

Understand, collect, select, align, verify.



Questions?

Contact: jackbfrancis@gmail.com
Website: jfprojects.dev