# Visualisation platform for AI Models

## Pre-rendered forecast charts on the European Weather Cloud

**Michiel Van Ginderachter** — Royal Meteorological Institute of Belgium

*European Weather Cloud User Workshop · 2026*

# The start: RMI's AI-LAM

## The model

- Data-driven **limited-area model** (Anemoi ecosystem)
- Grid **1069 × 1069**, ~**5.5 km** spacing
- **+0 … +144 h** in 6 h steps

## Where it runs

- Inference on ATOS GPU resources
- 00 and 12 UTC
- Under user account
- Output written as `zarr`

## The gap

Inference running robustly.

Results are on under local user-account

### Two questions

Where can I see the forecast?

Where do I find the data?

# Both answers live in the same EWC tenancy.

### "Where can I see it?"

A **URL**. Scrub through every field, every lead time of a recent run — no install, no notebook, no account on our machine.

### "Where do I find the data?"

A **bucket**. `s3://ai-lam-staging/…` on EWC object storage — the *same* zarr the viewer renders from, open to colleagues.

Three concrete jobs:

**(1)** daily sanity check on the operational-like runs ·

**(2)** look at **case studies** (e.g. the 2021 Vesder flood) ·

**(3)** something we can hand to a colleague as a URL, not as a notebook.

# Constraints we chose to accept

## Constraints

- **Runs close to the data** — inside the EWC, no egress of full forecast fields
- **One small VM**, no GPU, no Kubernetes, no database
- Must survive being ignored for weeks (systemd, not a tmux session)
- **Not public**: internal evaluation tool behind auth

## Deliberate non-goals

- Not a replacement for our operational visualisation chain
- No arbitrary user-defined domains, levels or colour scales
- No on-demand statistics
- No mobile-first design

> A small, boring, fully-owned tool.
> Total footprint: **~1 600 lines** of Python + JS.

# Pre-render

## What we already knew

`earthkit-plots`, `cartopy`, `matplotlib`.

Turning a field into a good-looking chart was, for us, a **solved problem**.

## What we did not know

Every single part of getting that chart in front of somebody else.

> So we put the work in the half we understood. Render the frames offline with the tools we already use — and the web side is reduced to **handing over files**, which is the simplest web problem there is.

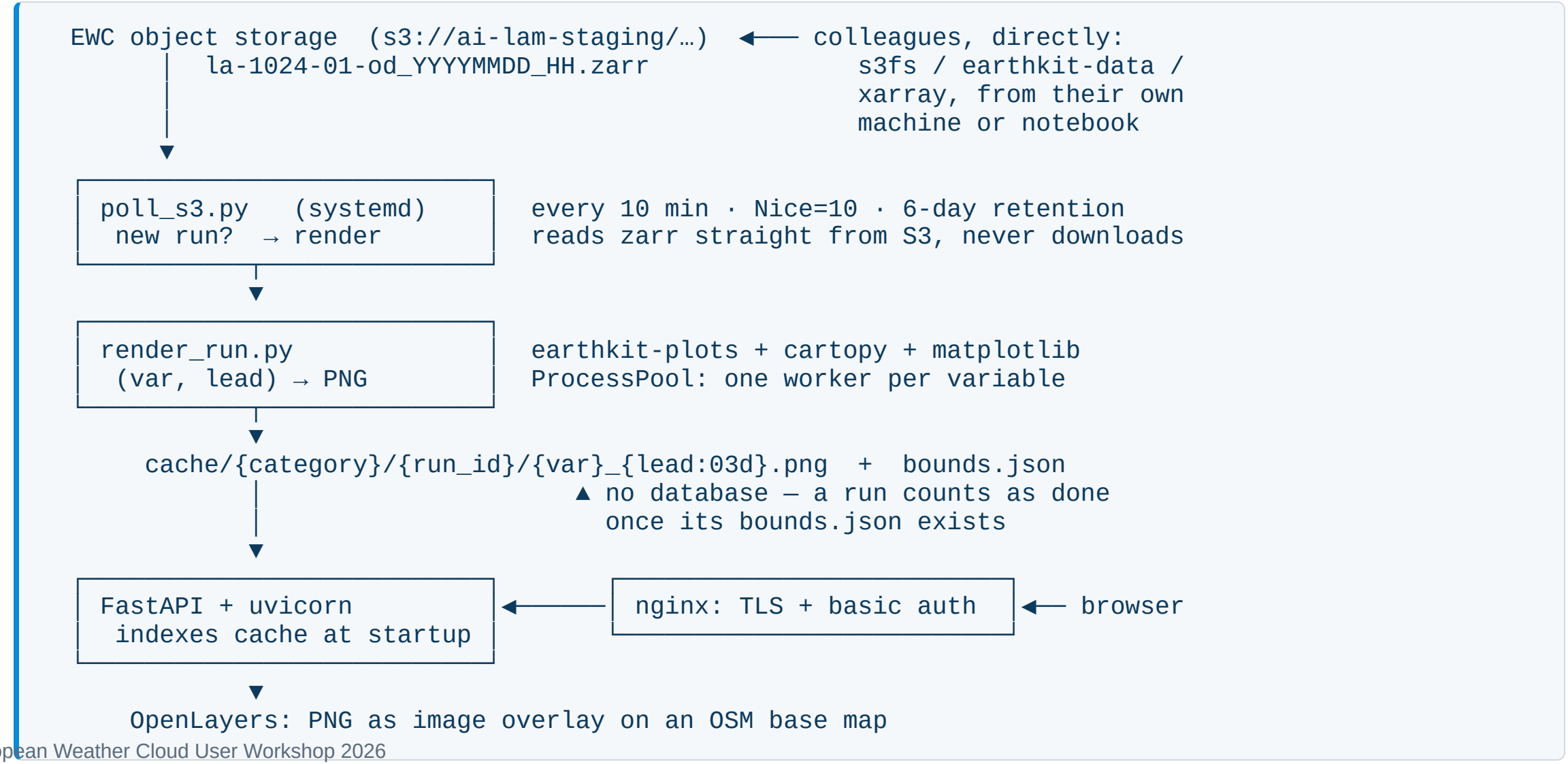
## What that bought us

Instant scrubbing · no CPU at request time · a backend small enough to read in one sitting

## What it costs

Disk · and a field set fixed in advance — no domain or colour scale we did not think of first

# Architecture



# What the numbers look like

**1 750**
PNG frames per run
(70 fields × 25 lead times)

**~7 min**
to render all of them,
8 workers on 8 cores

**2× daily**
00 and 12 UTC, rendered
unattended since April

~1.5 GB per run. A **6-day retention window** keeps the rolling cache bounded; curated case studies stay until we remove them.

# We had never done any of this before

## What we were comfortable with

- Python, `xarray`, `numpy`, `matplotlib`
- Submitting jobs to an HPC scheduler
- Reading NetCDF/GRIB off a shared filesystem

## What we had never done

- Provisioned a **virtual machine**
- Configured a **web server**
- Issued a **TLS certificate**
- Written a **systemd** unit
- Used **object storage** instead of a filesystem

# How we actually learned it

## Where the answers came from

- **EWC documentation** — the provisioning and networking pages answer most questions about standing up a VM
- **EWC webinars** — practical implementations of front-end and back-end machinery.
- **AI agents** — a fast way into unfamiliar territory: explaining an nginx config, drafting a systemd unit.

## What we did with it

`online_provision.md` — A literal **zero-to-HTTPS** runbook: every command, in order, with the traps annotated.

Rebuilding the machine is now under an hour — of which the VM itself is **5 minutes**; the rest is packages, certificates and auth.

# The machine

## What we run

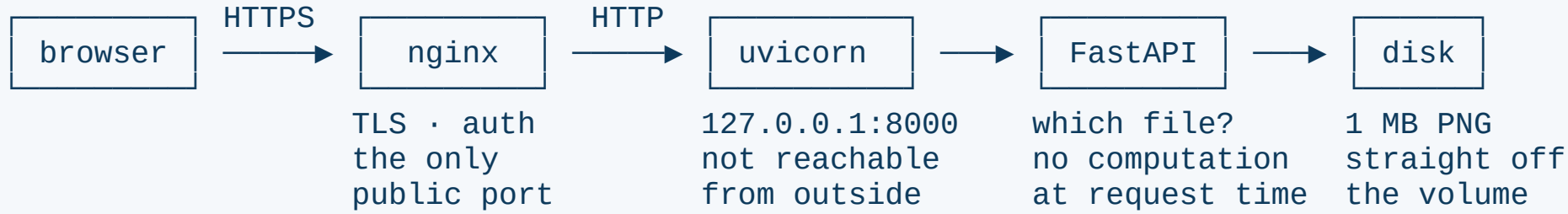
- `8cpu-32gbmem-80gbdisk` — 8 vCPU · 32 GB RAM · 80 GB volume
- **Ubuntu 24.04 LTS**
- **No GPU.**
- Volume is **48 % full** — the rolling 6-day cache is the only thing on it
- Provisioned in **5 minutes**. Idle most of the day: two renders, then nothing

## Decided at creation time — expensive to change later

- `NETWORKS: external-internet`
  → without it, the VM is simply not reachable
- `SECURITY GROUP: ssh-https`
  → only ports **22** and **443** are open

Build dependencies are not in the image — `cartopy` needs `build-essential`, `libgeos-dev`, `libproj-dev`, `libgdal-dev` before `uv sync` will succeed.

# What actually serves the pages




## FastAPI — the library

Turns an incoming **URL into a function call**.

The decorator declares which URL a function answers:

```python
@router.get("/api/tiles/{run}/{var}/{lead}")
def get_tile(run, var, lead):
    return FileResponse(png_path)
```

Essentially our whole backend — **~110 lines**.

## uvicorn — the process

The thing that actually **listens on a port** and runs that app.

> FastAPI decides *what* to answer.
> uvicorn does *the networking*.

Both pure Python, in the same environment as the renderer — one `uv sync`, no second language to learn.

# Getting it onto the internet

### 1 · Keep the app private

`uvicorn` binds to **`127.0.0.1:8000`** — not reachable from outside the machine.

### 2 · Put nginx in front

A reverse proxy — the **only** thing listening publicly.

### 3 · Get a certificate

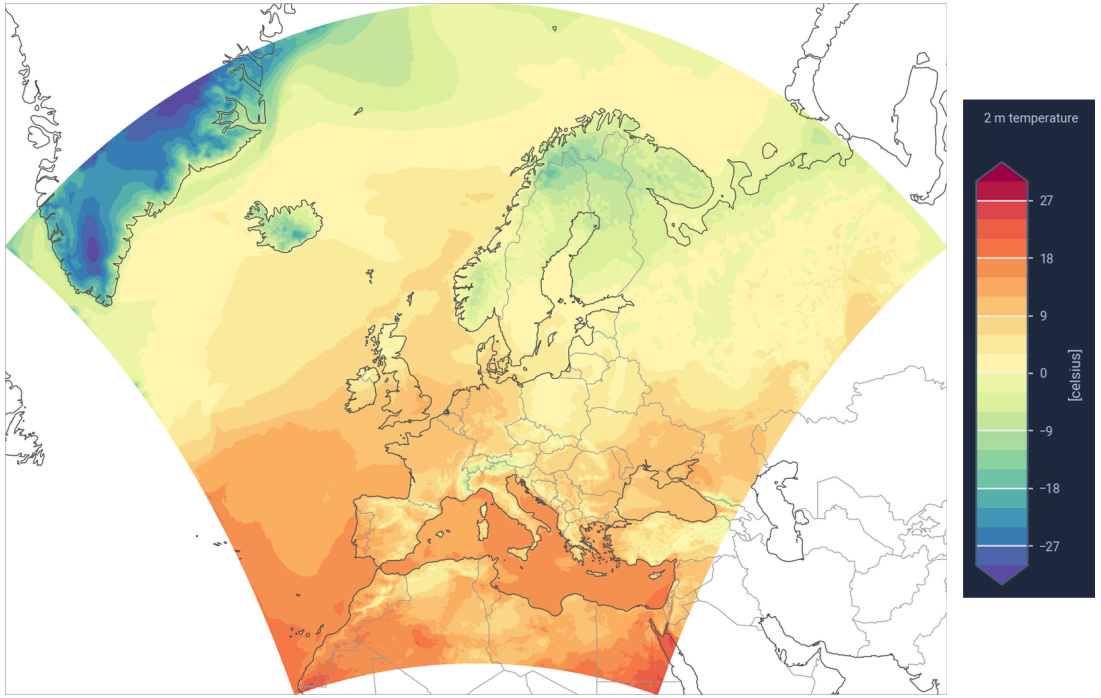
**`certbot --nginx`** → free Let's Encrypt cert **plus** an auto-renew timer, no downtime.

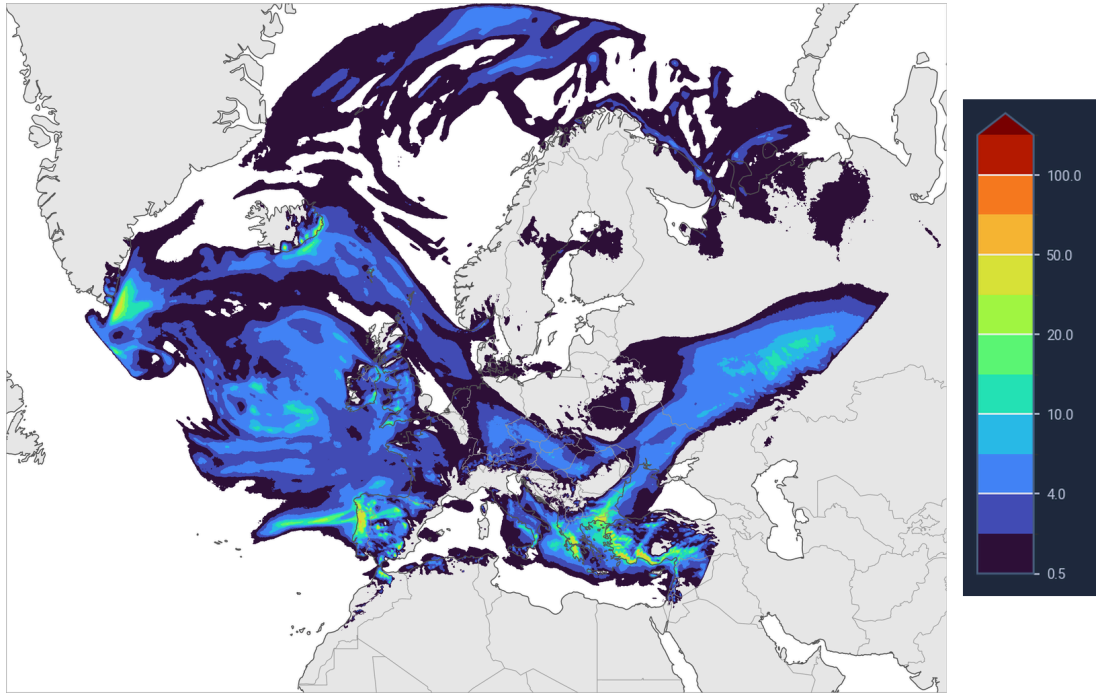
### 4 · Put a lock on it

**`htpasswd`** basic auth — an internal tool, not a public product.

**Then make it survive a reboot:** two `systemd` units — the web server and the S3 poller.

# What comes out the other end




**2 m temperature, +24 h** — one frame of 1 750 in a single run

**Precipitation, case study** — same pipeline, land masked, own colour scale

In the browser these sit at 75 % opacity over an OSM base map, with the colourbar rendered from the *same* style object — so the legend can never drift out of sync with the field.

# A short learning curve. A great deal of flexibility.

The European Weather Cloud asks for a modest investment in unfamiliar ground — provisioning, networking, certificates. What it returns is the freedom to build precisely the tool the work needs, and to run it alongside the data it depends on.

# Where we go next

## The product

- Two-run **comparison**: AI-LAM vs NWP
- **Observations** overlaid on the field
- More interactivity — click a point for a meteogram
- **Client-side rendering**, for what we *cannot* enumerate in advance:
  custom levels, free zoom, on-the-fly differences

## The setup

- Adopt **EWC IAM** — the new tenancy-wide service for managing users and their credentials.
- Provision through **OpenStack** directly rather than the Morpheus UI
- An **Ansible** playbook, so the whole build is one command

Thinking of doing the same for your model? It is **one VM, one bucket and a couple of days** the first time round — and we wrote the runbook so you do not have to. Contact me.

# Thank you

**Michiel Van Ginderachter**

michiel.vanginderachter@meteo.be

Royal Meteorological Institute of Belgium

Stack: `earthkit-plots` · `cartopy` · `xarray` / `zarr` · `s3fs` · FastAPI · OpenLayers

Running on: **one EWC VM** + **one EWC bucket** — `nginx` + `uvicorn` + two `systemd` units

*Questions?*

# Backup slides

*Technical detail, for questions only*

# Backup — The frontend is deliberately thin

**OpenLayers**, ~200 lines, no build step, no framework:

- `ol.source.ImageStatic` — one PNG as a georeferenced overlay
- 75 % opacity over an **OSM** base layer, so geography stays readable
- `<select>` for category / run / field, `<input type=range>` for lead time
- `bounds.json` fetched **once per run**: extent, lead times, field list
- Changing the slider = swapping an image source. That's the whole interaction model.

## Why not map tiles?

Tiling 1 750 frames into a pyramid multiplies both render time and file count by an order of magnitude — to serve a domain that fits on one screen.

A single 3 MP PNG per frame is *the correct tile size* for a LAM.

## Colourbars as separate PNGs

Rendered from the same `Style` object as the map, so the legend can never drift out of sync with the field.

# Backup — Render in the projection you will display in

We first rendered in the model's native projection and reprojected afterwards. Two steps, one extra dependency, and a **visible north–south shift** against the OSM tiles.

Then we rendered directly in Mercator — and the shift was *still there*, smaller.

## The culprit

```
# Wrong: defaults to the WGS84 ellipsoid
ccrs.Mercator()

# Right: the sphere EPSG:3857 / OSM actually uses
ccrs.Mercator(globe=ccrs.Globe(
    ellipse=None,
    semimajor_axis=6378137,
    semiminor_axis=6378137))
```

Let **cartopy** do the projection change while it draws. No `rasterio` step, no resampling, no second interpolation of your model fields — and the PNG drops onto the OSM base map pixel-perfect.

# Backup — Pin the canvas, or your frames won't line up

Every frame must have **exactly** the same pixel extent — the frontend reuses one `imageExtent` for all 1 750 of them.

## What went wrong

- `bbox_inches="tight"` recomputes the crop **per frame** → a few pixels of variation → overlay jitter when scrubbing
- `ax.get_xlim()/get_ylim()` are not the data extent: cartopy adjusts them to enforce the aspect ratio

## What we do now

- Extent computed from the data's **lon/lat corners** via pyproj proj4 strings
- One throwaway **warm-up render** (the most complex style path) captures a `save_bbox`
- That same bbox is reused for **every** `savefig` in the run

Side effect worth the slide on its own: the warm-up also absorbs the one-off cost of matplotlib font-cache and cartopy feature-download, so the first *real* frame is as fast as the last.

# Backup — Parallelise along your storage layout

The zarr store is chunked **per variable**.

Our first parallel version split work **per frame** — so 25 workers each pulled the same variable chunk from S3. Object storage was the bottleneck, not the CPU.

```
# One task = one variable, all lead times
tasks.append((var, ds_plot, cfg,
              lead_items, ...))
```

Each worker fetches its chunk **once**, then renders 25 frames from memory.

**The parallel decomposition should mirror the chunking of the store you read from.** Obvious in hindsight; it was a >2× speed-up and it removed all the S3 retry noise from our logs.

Related: `--workers 0` = all cores for a manual backfill, but the systemd poller runs with `Nice=10` so an arriving forecast never starves the web server.