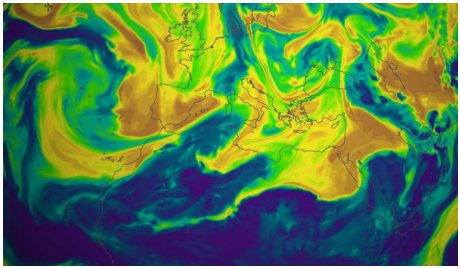
Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy playing with Python and AI/ML!

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	**Lab:** Python Startup: Basics	**Lab:** Feed-forward and Graph NNs	**Lab:** Graph Learning with PyTorch	**Lab:** Containers and Reproducibility	**Lab:** Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	**Lab:** Work environments, Python everywhere	**Lab:** Simple Transformer and LLM Use	**Lab:** Agent Frameworks	**Lab:** CI/CD Pipelines	**Lab:** Graph-based Assimilation
12:45–13:30			**Lunch Break**		
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	**Lab:** GRIB, NetCDF and Obs Visualisation	**Lab:** RAG Pipeline	**Lab:** DAWID Exploration	**Lab:** Anemoi Training Pipeline	**Lab:** Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	**The AI Transformation**	Learning from Observations Only
16:15–17:00	**Lab:** Torch Tensors and First Neural Net	**Lab:** Radar, SAT and Multimodal Data	**Lab:** MLflow Hands-on	**Lab: How work style could change**	**Lab:** ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

Why Emulators in Numerical Weather Prediction?



- High-resolution NWP is computationally expensive
- Many applications do not require full physical fidelity
- Fast forecasts enable new applications and scale

Emulator concept

- Learns the forecast step: state $\rightarrow$ state
- Operates within an existing NWP ecosystem
- Preserves grids, variables, and semantics

Goal of this lecture

- Understand emulator-based AI systems
- Compare AIFS and AICON
- Walk through AICON step by step

Emulators complement numerical models — they do not replace physical understanding.

Big-Tech AI Weather Models: Architectural Landscape

GraphCast (Google DeepMind)

- Graph Neural Network (GNN)
- Icosahedral grid, $\sim 0.25^\circ$
- Multi-mesh message passing
- Deterministic medium-range forecasts

Pangu-Weather (Huawei)

- 3D Transformer architecture
- Regular lat–lon grid, $\sim 0.25^\circ$
- Attention over space and vertical levels
- Image-like representation of atmosphere

GenCast (Google DeepMind)

- Diffusion model on graphs
- Icosahedral grid, $\sim 1^\circ$–0.25°
- Probabilistic forecasting
- Explicit uncertainty representation

FourCastNet (NVIDIA)

- Fourier Neural Operator (FNO)
- Regular lat–lon grid, $\sim 0.25^\circ$
- Spectral convolution in Fourier space
- Extremely fast inference

All models replace the forecast step — none are embedded in classical NWP systems.

Research and Operational AI Weather Systems

Aurora (Microsoft)

- Transformer-based foundation model
- Regular lat–lon grids
- Multi-task Earth-system scope
- Research-driven, not NWP-native

AIFS (ECMWF)

- Graph Neural Network (Anemoi)
- ECMWF Flagship ML Model
- Trained on ECMWF reanalysis ERA
- Operational global forecast system

BRIS (Met Norway)

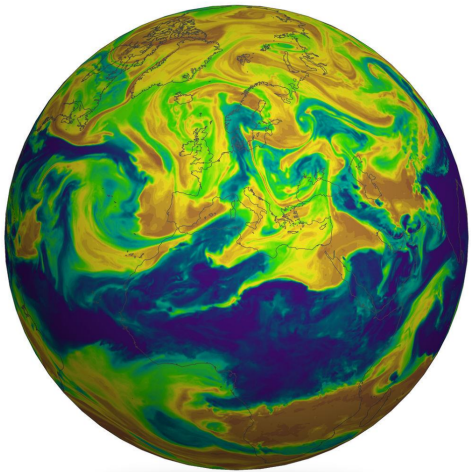
- Anemoi-based national system
- Nested Grid Approach, high resolution
- Independent training, AIFS-aligned
- Operational national usage

AICON (DWD)

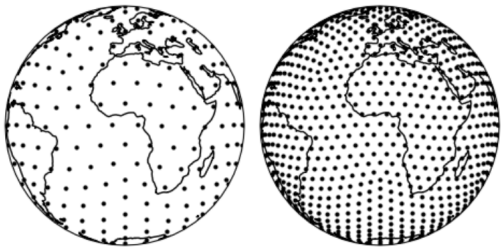
- Graph Neural Network (Anemoi)
- Native ICON triangular grid
- Trained on ICON-DREAM reanalysis
- Operational ICON emulator

European Meteorological Infrastructure is carrying out exciting development.

Anemoi based Emulators: Fields and Grids



Icosahedral Grid Representation



Triangular ICON mesh used as the computational graph for message passing and inference.

AICON learns a mapping on the ICON mesh, not on a Cartesian grid.

ICON-DREAM Reanalysis: High-quality training basis for AI emulators

Motivation

- reanalyses are key for climate services and many applications
- now also essential as training basis for AI-based NWP emulators

ICON reanalysis framework at DWD

- **ICON-DREAM:** global-to-regional ensemble reanalysis
- **ICON-FORCE:** 2.1 km ensemble reanalysis for Central Europe
- focus on Europe and on continuous updates (2010–present)

ICON-DREAM delivers consistent, high-resolution data

ICON-DREAM: key characteristics

- global ICON at **13 km**, 120 levels
- two-way nest over Europe at **6.5 km**
- DA: LETKF with **20-member ensemble** for B-covariances
- ensemble at 40 km (20 km over Europe)

ICON-FORCE (2 km):

- hourly LETKF (KENDA), incl. radar + SEVIRI
- 2-moment microphysics, snow / SST analyses

Outlook: backward extension to the 1980s using rescued obs + CDR products.

AICON Walkthrough: Environment Setup

Thanks to *Florian Prill* for a great Notebook!

Core ML stack

- Python $\geq$ 3.10
- PyTorch (CPU or CUDA)
- PyTorch Lightning
- PyTorch Geometric (PyG)

Anemoi framework

- `anemoi-core` (graphs, models, training)
- `anemoi-datasets` (Zarr + YAML I/O)
- `anemoi-inference` (generic inference)
- Supporting packages (e.g. transforms)

Data and configuration

- Zarr (chunked training data)
- Xarray (inspection and analysis)
- eccodes + earthkit (GRIB2 handling)
- Hydra / OmegaConf (YAML configuration)

Recommended setup

- Pinned `requirements.txt`
- Virtual environment or container
- Consistent library versions

Reproducibility depends on a stable, fully controlled software environment.

AICON Walkthrough: Manual Setup (Packages and ecCodes)

Install required Python packages

Pinned Python environment

```
# Install all required packages (version!)
%pip install --no-cache-dir -U -r ../requirements.txt

# Inspect installed versions
!pip freeze
```

Exact package versions are critical for PyTorch, PyG, Lightning, and Anemoi compatibility.

Configure ecCodes definitions

```
import os
from pathlib import Path

# Path to ICON/DWD ecCodes definitions
# (provided with the repo)
edzw_defs = Path("eccodes/definitions.edzw").resolve()

# Standard ecCodes definitions (system install)
std_defs = "/path/to/eccodes/definitions"

os.environ["ECCODES_DEFINITION_PATH"] = \
    f"{edzw_defs}:{std_defs}"
```

ICON GRIB decoding requires extended definition tables beyond standard ecCodes.

AICON Walkthrough: Notebook Initialization

Jupyter and runtime setup

Notebook basics

```
% Enable autoreload for
    iterative development
%load_ext autoreload
%autoreload 2

# Standard imports
import os
import numpy as np
import torch
```

Ensures code changes are picked up and the runtime environment is visible.

Determinism and sanity checks

Reproducibility checks

```
# Fix random seeds
torch.manual_seed(42)
np.random.seed(42)

# Check PyTorch device
device = torch.device("cuda" if
      torch.cuda.is_available()
     else "cpu")
print("Using device:", device)
```

Deterministic behavior is essential for debugging and scientific comparison.

Initialization is not boilerplate — it defines the experimental contract.

AICON training setup (Anemoi): config-driven workflow

Main idea

- training is fully controlled by YAML configs
- Hydra composes config blocks:
  - data (Zarr), variables, normalization
  - model architecture (AICON / GNN)
  - trainer settings (epochs, batch, strategy)

Workflow

- choose config (start from integration test)
- instantiate trainer
- run training (checkpoints + logger)

Notebook pattern (core lines)

```
config_filename = "test_aicon_01.yaml"

# Hydra loads and resolves config
config = load_config(config_filename)

# Anemoi trainer wraps Lightning
trainer = AnemoiTrainer(config)

# run training
trainer.train()
```

Outputs

- checkpoints: `last.ckpt`, `inference-last.ckpt`
- logs: TensorBoard / MLflow-style metrics

Training diagnostics: loss curve and experiment logger

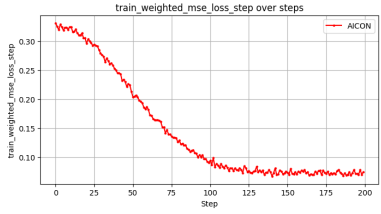
What we monitor

- training loss vs epoch/step
- validation loss (generalization)
- learning rate schedule
- walltime + throughput

Why it matters

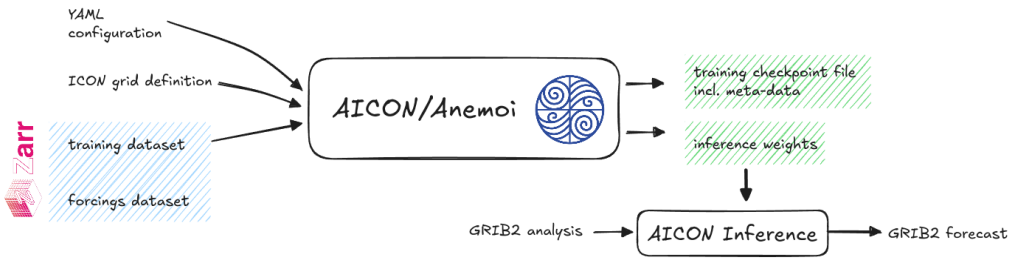
- early detection of instability
- reproducible experiments
- compare model variants

Take-away: training is observable and traceable.




```python
class GraphicalLiveLogger(Logger):
    def __init__(self, interval=10):
        super().__init__()
        self.metric_name = "train_weighted_mse_loss_step"
        self.metrics_history = []

    @property
    def name(self): return "LiveLogger"
    @property
    def version(self): return "0.1"

    def log_hyperparams(self, *args, **kwargs): pass

    def log_metrics(self, metrics, step):
        self.metrics_history.append((step, metrics))

    def finalize(self, status):
        steps, values = zip(*[(s,m[self.metric_name]) for s,m in self.metrics_history if self.metric_name in m])
        plt.figure(figsize=(8,4));
        plt.plot(steps, values, 'o-', ms=2, label="AICON", color="red")
        plt.title(f"{self.metric_name} over steps"); plt.xlabel("Step"); plt.ylabel(self.metric_name)
        plt.grid(True); plt.legend(); plt.show()

trainer.model.logger_enabled = True
trainer.loggers = GraphicalLiveLogger()
```

AICON training workflow (overview)



Pipeline: ICON data → graph dataset → training config → trainer → checkpoints + logs

Training configuration: input and outputs

Input data (from config)

```text
ICON mesh (NetCDF):
  <.../icon_mesh.nc>

Training interval:
  <YYYY-MM-DD HH> ... <YYYY-MM-DD HH>

Data files:
  <...>  (joined list)
```

Key point: a single config fixes both data geometry and time range.

Output directories

```text
Checkpoints:
  <.../checkpoints/...>

Logs (incl. MLFlow):
  <.../logs/...>
```

Take-away: config ⇒ reproducible paths and artifacts.

Training dataset (Zarr) inspected as Xarray

Key idea

- dataset is a 4D tensor : (time, variable, ensemble, cell)
- efficient I/O via chunking (Dask)
- includes metadata:
  - latitudes / longitudes
  - variable-wise statistics (mean, std, min/max)

Take-away: this is ML-ready data with rich metadata.

▸ Dimensions: (variable: 84, time: 724, ensemble: 1, cell: 11520)

▸ Coordinates: (0)

▾ Data variables:

count	(variable)	float64	dask.array<chunksize=(84,), me...
data	(time, variable, ensemble, cell)	float32	dask.array<chunksize=(1, 84, 1,...
dates	(time)	datetime64[s]	dask.array<chunksize=(724,), m...
has_nans	(variable)	object	dask.array<chunksize=(84,), me...
latitudes	(cell)	float64	dask.array<chunksize=(11520,),...
longitudes	(cell)	float64	dask.array<chunksize=(11520,),...
maximum	(variable)	float64	dask.array<chunksize=(84,), me...
mean	(variable)	float64	dask.array<chunksize=(84,), me...
minimum	(variable)	float64	dask.array<chunksize=(84,), me...
squares	(variable)	float64	dask.array<chunksize=(84,), me...
stdev	(variable)	float64	dask.array<chunksize=(84,), me...
sums	(variable)	float64	dask.array<chunksize=(84,), me...

Vertical levels: selected ICON layers in the dataset

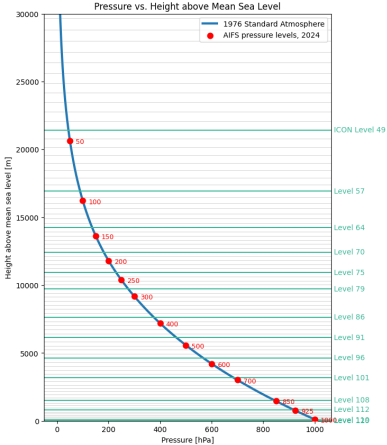
Variables on model levels

shortName	Name	Levels
P	Pressure	49,57,64,70,75,79,86,...,12
QV	Specific humidity	49,57,64,70,75,79,86,...,12
T	Temperature	49,57,64,70,75,79,86,...,12
U	U-component of wind	49,57,64,70,75,79,86,...,12
V	V-component of wind	49,57,64,70,75,79,86,...,12
W	Vertical velocity (geom.)	49,57,64,70,75,79,86,...,12

Forcings / static features (examples):

EMIS_RAD, FR_LAND, FR_LAKE, HSURF, Z0, insolation, SSO_*, sin/cos(latitude, longitude, local_time, julian_day)

Idea: multi-level state + forcings $\Rightarrow$ complete ML input for AICON.



ICON level structure: full model vertical grid (selection highlighted).

ICON multimesh: the hidden graph behind AICON

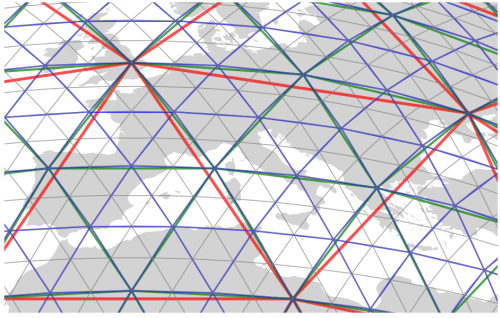
Key idea

- AICON does not run on a regular lat–lon grid
- it uses ICON's hierarchical triangular mesh
- the GNN operates on a hidden multi-mesh

Multi-mesh principle

- union of coarse-to-fine subgraphs
  RnB0 ∪ RnB1 ∪ ... ∪ RnBk
- mixes short-range + long-range edges
- similar spirit as GraphCast multi-mesh

Take-away: the multi-mesh gives the GNN both local physics and global context.



Hidden mesh edges shown level-wise: coarse levels enable long-range interaction.

AICON model: encoder – graph processor – decoder

Graph-to-graph forecasting

- input: state + forcings on ICON cells (nodes)
- output: next-step state (multi-var, multi-level)

Architecture blocks

- **Encoder:** maps raw variables to latent node/edge features
- **Processor:** message passing on the mesh graph (several layers)
- **Decoder:** projects latent features to physical output variables

Take-away: encoder/decoder makes the network variable-agnostic in latent space.

```
encoder:
  _target_: anemoi.models.layers.mapper.GraphTransformerForwardMapper
  _convert_: all
  trainable_size: ${model.trainable_parameters.data2hidden}
  sub_graph_edge_attributes: ${model.attributes.edges}
  num_chunks: 2
  cpu_offload: ${model.cpu_offload}
  mlp_hidden_ratio: 4
  num_heads: 16
  qk_norm: false
  layer_kernels:
    Linear:
      _target_: torch.nn.Linear
      _partial_: true
    Activation:
      _target_: torch.nn.GELU
decoder:
  _target_: anemoi.models.layers.mapper.GraphTransformerBackwardMapper
  _convert_: all
  trainable_size: ${model.trainable_parameters.hidden2data}
  sub_graph_edge_attributes: ${model.attributes.edges}
  num_chunks: 1
  cpu_offload: ${model.cpu_offload}
  mlp_hidden_ratio: 4
  num_heads: 16
  qk_norm: false
  initialise_data_extractor_zero: false
  layer_kernels:
    Linear:
      _target_: torch.nn.Linear
      _partial_: true
```

Encoder maps physics variables → latent graph; decoder maps latent → output fields.

Graph structure: local neighborhoods on the ICON mesh

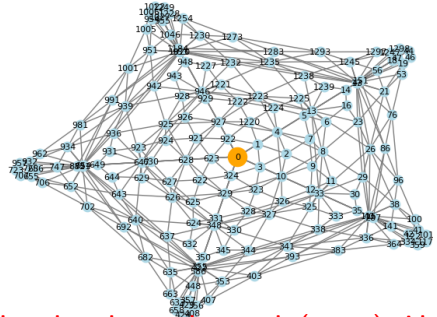
Why this matters

- AICON runs on the ICON grid as graph neural network (GNN)
- each ICON cell ⇒ one graph node
- edges connect mesh neighbors

Message passing view

- prediction at one node uses information from:
  - **1st neighbors** (directly connected)
  - **2nd neighbors** (neighbors-of-neighbors)
- this defines the local receptive field of the GNN

Take-away: GNNs learn transport + interaction patterns through local connectivity.



Example: subgraph around one node (orange) with 1st and 2nd neighbors.

Graph Transformer attention: which neighbors matter?

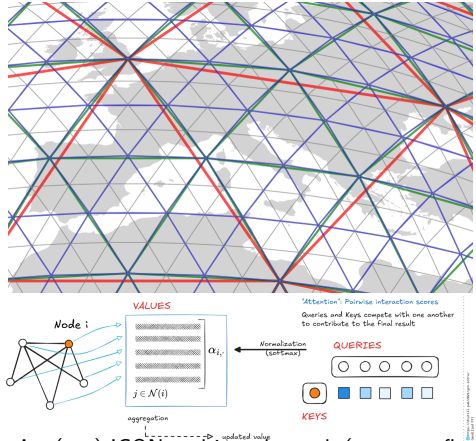
Graph Transformer vs. plain message passing

- classic GNN: neighbors contribute via fixed averaging / sum
- GraphTransformer: each edge gets a learned attention weight

Interpretation

- for one receiver node i, all incoming edges $(j \to i)$
- attention weights $\alpha_{i,j}$ sum to 1 (per head)
- thick edges $\Rightarrow$ high influence of neighbor j

Take-away: the model learns where to look on the graph depending on flow regime and structure.



Example: (top) ICON multi-mesh graph (coarse + fine connections); (bottom) attention setup

Transfer learning: reuse weights on a finer hidden mesh

Motivation

- training on the finest multi-mesh is expensive
- idea: train on coarse mesh and transfer to finer mesh

What transfers well?

- most encoder/processor/decoder weights
- learned graph attention patterns are largely mesh-agnostic

What must be adapted?

- graph-dependent parameters:
  - hidden node coordinates / embeddings
  - extra trainable node/edge attributes
- these are re-initialized and fine-tuned on the new mesh

Example (AICON walkthrough)

- change hidden mesh resolution:
  - `max_level_multimesh`: $3 \rightarrow 4$
- load weights only + enable transfer learning

Config idea (YAML)

```
training:
  load_weights_only: true
  transfer_learning: true
  run_id: <pretrained_run_id>

graph:
  nodes:
    icon_mesh:
      node_builder:
```

Operational inference at DWD: why a dedicated tool?

Goal

- run AICON forecasts operationally and reproducibly
- integrate into existing NWP infrastructure (GRIB2, workflows, monitoring)

Why not only `anemoi-inference`?

- DWD-specific I/O and preprocessing (GRIB2 conventions)
- additional input engineering (e.g. soil moisture index etc.)
- packaging as stable runtime for ops (container)
- hooks for monitoring and verification pipelines

Key ingredients

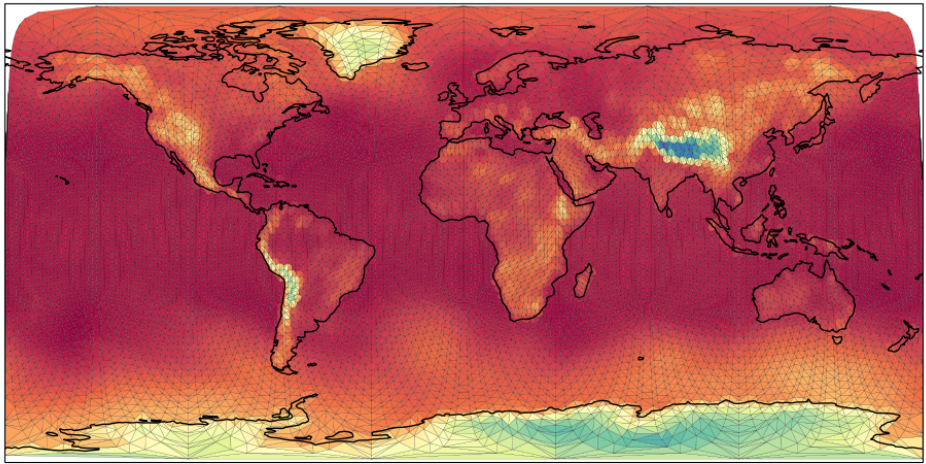
- **checkpoint:** `inference-last.ckpt`
- **input:** ICON / reanalysis data streams
- **output:** GRIB2 forecast products

Operational packaging

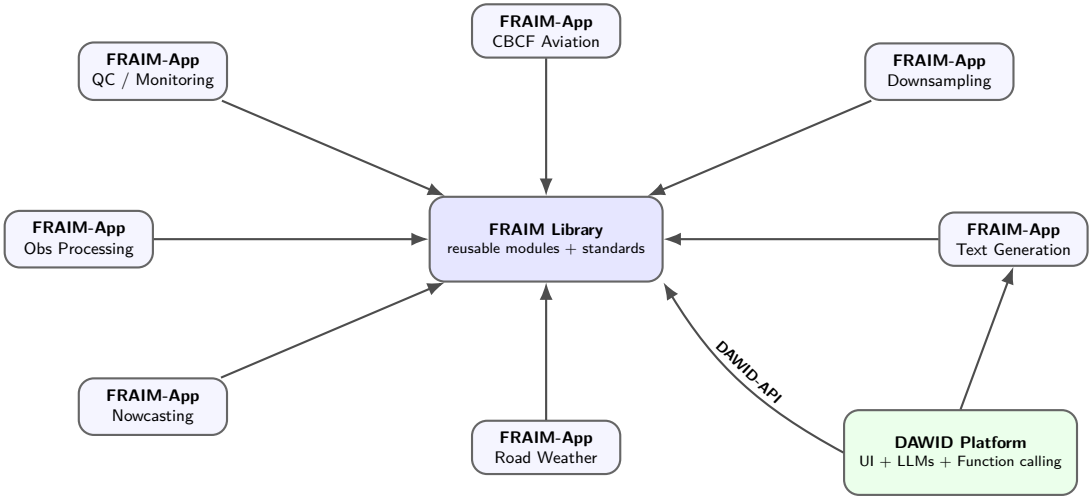
- Apptainer / Singularity container
- fixed paths to constants + weights
- versioning: container + config + checkpoint

Take-away: DWD inference tool = bridge from ML model to ops.

Demo: AICON forecast from operational inference



FRAIM Ecosystem: Central Library, FRAIM-Apps, and DAWID Platform



Anemoi vs. FRAIM — What are they about?

Anemoi (ECMWF & Partners)

- international collaboration to build ML-based forecast models
- **end-to-end framework** for training & inference
- well-defined pipeline: datasets → graphs/models → training → inference/deploy
- **Open Source**, pan-European community

Typical question:
How do we train and operate an ML forecasting model operationally?

FRAIM (E-AI / DWD / Partners)

- international collaboration focused on many AI applications
- **method toolbox / modular framework** across a broad portfolio: products, services, and smaller AI components
- platform/integration view: standards, building blocks, reuse
- **umbrella framework** (forecasting is only one use case)

Typical question:
How do we build a modular AI ecosystem for many meteorological products and services?

Pipeline view: where does each framework sit?

Anemoi: lifecycle of an ML forecast model

Data	→	**Training**	→	**Inference/Deploy**
(anemoi-datasets)		(anemoi-training)		(operational tooling)

FRAIM: system / toolbox view across multiple AI components

Use case	→	**Modules**	→	**Integration/Operations**
Forecast / Nowcast / QC / DA / Postproc QA, LLM, etc.		training, surrogates, platform + standards		

Take-away:
Anemoi is forecast-model-centric, FRAIM is architecture- & reuse-centric.

Concrete differences and a good integration story

Differences (short & concrete)

- **Scope:** Anemoi = end-to-end ML forecasting FRAIM = modular AI toolbox for many meteorological products
- **Core artifacts:**
  - **Anemoi:** datasets, graphs, model weights, training pipelines
  - **FRAIM:** central library + reusable modules + standards
- **Organization principle:**
  - Anemoi: one coherent forecasting stack
  - FRAIM: many use-case driven FRAIM-Apps (each in its own repo)

How does this combine well?

- FRAIM can integrate mfai or Anemoi as a forecasting or processing engine
- FRAIM then provides:
  - product-specific pipelines (QC, DA, verification, monitoring, etc.)
  - deployment patterns and operational standards

One-liner:
Anemoi = *forecast-model factory*;
FRAIM = *modular product ecosystem*.

DAWID — LLM Platform, Tools, and FRAIM Integration

DAWID capabilities

- **LLM user interface**
  - chat + document context (RAG)
  - role-based workflows (science / dev / ops)
- **Function calling**
  - controlled execution of domain functions
  - structured I/O and provenance
- **Agents (multi-step)**
  - plan → call tools → validate

Key idea: DAWID is the interactive AI cockpit.

DAWID-API integration

- **Unified API** to LLMs and tools
  - multi-LLM backends
  - tool/function endpoints
- **Link to FRAIM-Apps**
  - FRAIM-App → DAWID-API: reasoning + tooling
  - consistent interface across products

Take-away: DAWID connects LLMs + tools + FRAIM-Apps.