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 CI/CD Is Not Optional for AI/ML

What continuously changes in AI/ML

- Model parameters through retraining
- Training and validation data
- Feature engineering and preprocessing
- Hyperparameters and runtime configuration

As a consequence:

- System behavior is not fixed
- Outputs depend on code, data, and environment
- Small changes can have large effects

Why manual workflows break down

- Experiments cannot be reproduced reliably
- Results depend on undocumented environments or training data
- Errors surface late or not at all
- Deployment decisions become guesswork

Without automation:

- Models cannot be trusted operationally
- Debugging becomes forensic work

CI/CD is the mechanism that makes AI/ML systems controllable and trustworthy.

AI/ML Is Not Special — But Updating Becomes Complex

From a software engineering perspective

- AI/ML systems are still software systems
- They use standard languages, libraries and toolchains
- The same DevOps principles apply

There is no special engineering magic in AI/ML:

- version control
- testing
- packaging
- deployment

CI/CD is needed to manage updates, not to handle "AI magic".

Where the real complexity comes from

- Frequent retraining with new data
- Multiple preprocessing and feature pipelines
- Changing model architectures
- Different loss functions and objectives
- Many runtime and training configurations

As a result:

- Updates are continuous
- Reproducibility becomes non-trivial
- Manual tracking no longer works

AI/ML CI/CD Pipeline

Key components for building robust MLOps workflows

Training Data
- Data Versioning
- Quality Checks
- Labeling Pipeline
- Data Drift Detection

Model Architecture
- Model Registry
- Architecture Versioning
- Hyperparameters
- Code Repository

MLflow
- Experiment Tracking
- Model Registry
- Artifact Storage
- Metrics Logging

CI/CD Pipeline
- Automated Testing
- Model Training
- Build & Package
- Deployment Automation

Verification
- Model Validation
- Performance Tests
- A/B Testing
- Shadow Deployment

Applications
- API Endpoints
- Batch Inference
- Real-time Serving
- Model Integration

Monitoring
- Performance Metrics
- Model Drift
- System Health
- Alert Management

Starting Locally: A Concrete Git Hook Example

Black is an automatic code formatter for Python.
It rewrites Python source code into a single, consistent style, without asking questions.

Minimal pre-commit hook

.git/hooks/pre-commit

```
1 #!/bin/sh
2 pytest || exit 1
3 black .
```

This hook is executed automatically when running:

- `git commit`

If tests fail, the commit is blocked.

Git hooks enforce local discipline — not global policy.

What this enforces locally

- Code must be syntactically correct
- Tests must pass
- Code formatting is consistent

Key properties:

- Runs before code leaves the laptop
- No CI server involved
- Immediate feedback to the developer

When Are Git Hooks Executed?

Git commands trigger hooks
Git automatically executes hooks at well-defined points in its workflow.

Common examples:

- `pre-commit` — before a commit is created
- `commit-msg` — to validate commit messages
- `pre-push` — before pushing to a remote

Hooks run before Git completes the command.

Effect on the workflow

- Hook succeeds → Git continues
- Hook fails → Git aborts the command

This means:

- Invalid code never enters the repository
- Errors are caught immediately
- No manual checks are required

Hooks are deterministic: same input → same outcome.

Hooks turn Git commands into enforced quality gates.

A Git Hook in Action: What Actually Happens

Before committing

- Python test file is poorly formatted
- Code is syntactically valid
- Tests pass when run manually

After running `black`:

- Formatting is rewritten automatically
- No code logic is changed

During `git commit`

- `pytest` is executed automatically
- `black` is executed automatically
- Both run via the pre-commit hook

The hook turns a manual checklist into an automatic guarantee.

Outcome:

- Tests pass
- Formatting is consistent
- Commit is accepted

```
def add( a ,b ):
  return a+b
def test_answer( ):
  print("Testing add(1, 3) == 4")
  assert add(1,3)==4
```

```
def add(a, b):
    return a + b
def test_answer():
    print("Testing add(1, 3) == 4")
    assert add(1, 3) == 4
```

Strengths and Limits of Git Hooks

Strengths

- Extremely fast feedback. Helps you!!
- No external infrastructure required
- Works offline
- Integrated directly into Git commands

Git hooks are ideal for:

- formatting checks
- unit tests
- catching trivial mistakes early

Limitations

- Run only on the developer machine
- Not enforced across a team
- Can be bypassed or disabled
- No neutral execution environment

As a consequence:

- Hooks improve discipline
- But they do not guarantee quality

Git hooks accelerate development — CI platforms enforce standards.

Formatting vs Behavior: Two Different Checks

Code formatting (Black)

- Enforces a consistent style
- Removes whitespace and layout differences
- Does not change program logic

Example:

black reformats code

```
def add( a ,b ):
  return a+b
```

becomes:

```
def add(a, b):
    return a + b
```

Style and correctness are independent concerns.

Behavior testing (pytest)

- Checks whether code does the right thing
- Executes functions and validates results
- Fails if behavior changes unexpectedly

Formatting makes code readable. Testing makes code **correct**.

Code and Test: Defining and Enforcing Behavior

Formatted Code

test_example.py

```
1 def add(a, b):
2     return a + b
```

This code:

- is syntactically correct
- is well formatted
- may still be logically wrong

Formatting alone cannot guarantee correctness.

Correct behavior is defined by function tests, not by appearance.

Test that enforces behavior

test_example.py

```
1 def test_add():
2     assert add(2, 3) == 5
```

This test:

- executes the function
- checks the expected result
- fails automatically if behavior changes

The test can run:

- locally
- in Git hooks
- in CI pipelines

CI/CD Tools: Local Discipline and Correctness

Version control

- **Git**
  Tracks changes to source code and configuration files, enabling branching, merging, and full reconstruction of development history.
- **GitHub / GitLab**
  Hosting platforms for Git repositories, providing collaboration, access control, and CI integration.

Local enforcement

- **Git hooks**
  Local scripts executed automatically on Git events (e.g. `pre-commit`) to enforce rules before code is committed.
- **pre-commit framework**
  Version-controlled hook manager that ensures identical checks across developers and CI environments.

Code quality and correctness

- **Black**
  Automatic Python formatter that enforces a single, deterministic code style without configuration decisions.
- **pytest**
  Python testing framework that executes test functions and fails automatically when expected behavior is violated.

These tools:

- run fast
- provide immediate feedback
- prevent trivial errors from propagating

Local discipline reduces error rates before CI even starts.

CI/CD Tools: Automation and Execution

CI orchestration

- **GitHub Actions**
  Event-driven CI system integrated into GitHub, executing workflows defined in YAML on managed runners.
- **GitLab CI**
  Pipeline-based CI system configured via `.gitlab-ci.yml`, supporting self-hosted and specialized runners (e.g. HPC, GPU).
- **Jenkins**
  Standalone automation server using scripted pipelines, common in legacy and enterprise environments.

Execution environments

- **Python virtual environments**
  Isolate Python dependencies to ensure reproducible runtime behavior.
- **Containers (Docker, Apptainer)**
  Package applications and dependencies into portable, reproducible execution units across systems.

Artifacts

- **CI artifacts and registries**
  Store outputs such as logs, test reports, trained models, and container images produced during pipelines.

CI platforms enforce rules; environments make them reproducible.

Why Data Formats Matter in AI/ML

Characteristics of training data

- Very large (GB–TB scale)
- Multi-dimensional (time, space, channels)
- Often produced continuously

In contrast to classical workflows:

- Data rarely fits into memory
- Full sequential reads are uncommon

Typical ML access patterns

- Repeated sampling of small subsets
- Random or structured access
- Parallel reading by many workers

As a result, the data format directly affects:

- I/O performance
- Scalability of training
- Feasibility of distributed ML

Data formats are part of the ML infrastructure, not just storage.

Zarr: Why It Is Attractive for AI/ML

Design principles

- Chunked, array-based storage
- Designed for cloud and HPC
- Partial reads without full downloads

Why this helps ML training

- Efficient random sampling
- Scales to distributed workers
- Reduces I/O bottlenecks

Performance features

- Parallel reads of independent chunks
- Flexible chunk layout
- Compression per chunk

Typical use cases

- Images and video
- Satellite and geospatial data
- Scientific simulation output

Zarr optimizes data access patterns — not the ML logic itself.

Zarr: Limitations and When to Use Alternatives

Limitations

- Limited native integration with ML frameworks
- Metadata overhead for many small arrays
- Not well suited for tabular data
- Care needed for concurrent writes

Zarr adds complexity that is not always justified.

When to choose alternatives

- Tabular ML data: Parquet, Arrow
- TensorFlow pipelines: TFRecord
- PyTorch streaming: WebDataset
- Small datasets: NetCDF, HDF5, NPY

Guiding principle

- Choose formats based on data structure
- Optimize for access pattern, not fashion

Zarr is powerful when the problem matches the format.

Demo Forecast Dataset (Europe, 5 variables)

Dataset design

We generate a small synthetic forecast archive over Europe:

- grid: `lat` × `lon` (regular lat/lon)
- time axis: `valid_time` for lead times (e.g. 0–120h)
- variables:
  - `t2m` (2m temperature)
  - `u10`, `v10` (10m wind)
  - `mslp` (mean sea-level pressure)
  - `tp` (precipitation)

We mimic the access patterns of real NWP:

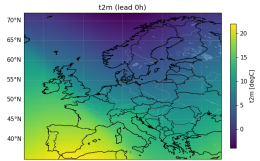
- map snapshot for one lead time
- town / station time series extraction
- local patches for ML training

Notebook outputs

- Zarr store: `data/demo_eu_forecast.zarr`
- first plots: Europe maps
- metadata: coordinates + attrs

Next

- generate fields with moving patterns
- write to Zarr with chunking



Lead Time: 0h

Synthetic Forecast Generation (moving patterns)

Idea

We generate physically plausible spatio-temporal structure:

- coherent "weather patterns" move eastward with time
- correlated variables:
  - pressure wave → wind via spatial gradients
  - temperature advected + noise
  - precipitation linked to fronts (thresholded patterns)

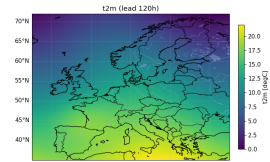
```
# grid (Europe)
lon2d, lat2d = np.meshgrid(lon, lat)
# moving phase (eastward shift with lead time)
phase = kx * lon2d + ky * lat2d - omega * lead_hours[t]

# temperature: advected pattern
t2m[t] = 10.0 + 6.0*np.cos(phase + 0.7) + 0.8*rng.normal(size=lon2d.shape)
```

Result

Fields are realistic enough to demonstrate:

- map snapshots
- point extraction
- patch-based ML sampling

Example snapshot



Lead Time: 120h

Writing the Zarr Archive (chunking for point extraction)

Zarr write: key decisions

We write the dataset as a chunked archive:

- **time chunk = 1:** fast town time series extraction
- **lat/lon chunks = 60:** moderate patch size for ML sampling

Chunk layout

- chunks stored as independent files
- town extraction reads few chunks only
- patches match ML mini-batches

We generate chunks that contain exactly one lead time and a 60×60 spatial patch.

Example:

`(valid_time, lat, lon) = (1,60,60)`

```
# ----------------------------------------------------------------
# Zarr write
# ----------------------------------------------------------------
OUTDIR = "data"
os.makedirs(OUTDIR, exist_ok=True)

ZARR_PATH = os.path.join(OUTDIR, "demo_eu_forecast.zarr")

# best practice: chunk so point extraction is fast
# - valid_time chunk = 1 (fast timeseries reading)
# - lat/lon chunks ~ moderate
chunked = ds.chunk({"valid_time": 1, "lat": 60, "lon": 60})

# overwrite if exists
if os.path.exists(ZARR_PATH):
    import shutil
    shutil.rmtree(ZARR_PATH)

chunked.to_zarr(ZARR_PATH, mode="w", consolidated=True)
print("Wrote:", ZARR_PATH)
```

Open Zarr Archive and Inspect Dataset

Open archive

We open the Zarr store as an `xarray.Dataset`:

- lazy loading via `dask.array`
- chunk layout is preserved: `(valid_time, lat, lon) = (1,60,60)`

Dataset structure

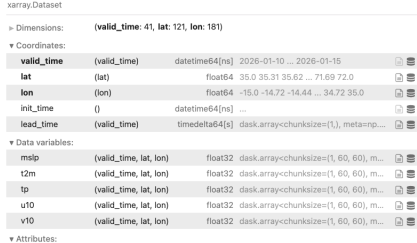
- dimensions: `valid_time=41`, `lat=121`, `lon=181`
- variables: `t2m`, `u10`, `v10`, `mslp`, `tp`
- coordinate: `lead_time` as timedelta

Key observation

Even though the archive is chunked, xarray provides a single logical dataset interface.

Example:

```
ZARR_PATH = "data/demo_eu_forecast.zarr"
ds = xr.open_zarr(ZARR_PATH, consolidated=True)
ds
```



Dataset preview: dimensions, coords, variables, and Dask chunks.

Town Forecast Extraction (nearest grid point)

Offline town database

We use a small built-in town table:

- no internet required
- town → `(lat,lon)`

Point extraction

From the full dataset we extract a single point forecast:

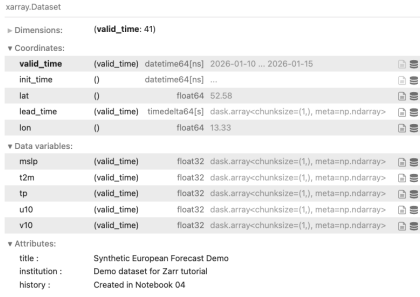
- use nearest neighbor selection: `ds.sel(..., method="nearest")`
- result: dataset with only dimension `valid_time`

Output

A compact dataset with time series for: `t2m`, `u10`, `v10`, `mslp`, `tp`

Example:

```
TOWNS = {
 "Berlin": (52.5200, 13.4050), ... }

pt = ds.sel(lat=lat0, lon=lon0, method="nearest")
```



Point forecast: only dimension `valid_time` remains.

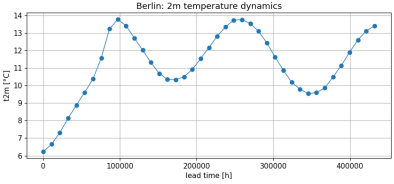
Town Forecast Dynamics (build time series table)

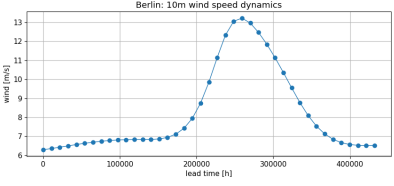
Prepare time series data

We convert the extracted point dataset into a clean table:

- convert times to `datetime`
- convert lead time to hours
- compute derived variables (e.g. wind speed)

```
# ------------------------------------------------------------
# Create a clean timeseries table
# ------------------------------------------------------------
lead_h = (pt["lead_time"].values / np.timedelta64(1, "h")).astype(int)

df = pd.DataFrame({
    "valid_time": pd.to_datetime(pt["valid_time"].values),
    "lead_h": lead_h,
    "t2m_C": pt["t2m"].values,
    "wind_ms": np.sqrt(pt["u10"].values**2 + pt["v10"].values**2),
    "mslp_hPa": pt["mslp"].values,
    "tp_mm": pt["tp"].values,
})
```





Local Map Plot (OSM basemap + 100 km temperature overlay)

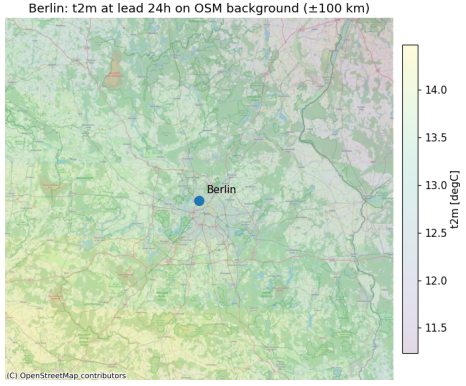
Visualize a local patch around the town:

- **OpenStreetMap (OSM)** background tiles
- overlay `t2m` for one lead time
- show only a 100 km box for clarity and speed

Key steps

- select lead time index (e.g. 24h)
- slice `t2m` in a local lon/lat window
- reproject to WebMercator (EPSG:3857)

```
t2m_local = ds["t2m"].isel(valid_time=i).sel(
    lat=slice(latc-dlat, latc+dlat),
    lon=slice(lonc-dlon, lonc+dlon)
)

# plot on OSM background (contextily)
ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik, zoom=10)
```



Example: Berlin, t2m at lead 24h (local 100 km window).

Training a CNN: field(t) -> field(t+1)

Learning task

We train a CNN to predict next-step temperature:

- input at time `t`: `t2m, u10, v10, mslp`
- target: `t2m(t+1)`

Patch training

Instead of full fields, we train on random patches:

- sample 64x64 patches from random positions
- efficient mini-batches on laptop/GPU
- same idea as Zarr chunks: local spatial tiles

Normalization

Channel-wise mean/std over time and space:

- stabilize training
- same scaling for training and evaluation

Example:

```
# inputs and target in RAM
t2m  = ds["t2m"].astype("float32").values
u10  = ds["u10"].astype("float32").values
v10  = ds["v10"].astype("float32").values
mslp = ds["mslp"].astype("float32").values

# normalization
Xin = np.stack([t2m, u10, v10, mslp], axis=1)
X_mean = Xin.mean(axis=(0,2,3), keepdims=True)
X_std  = Xin.std(axis=(0,2,3), keepdims=True) + 1e-6

y_mean = t2m.mean()
y_std  = t2m.std() + 1e-6


# patch sampler: X(t) -> y(t+1)
Xt, yt = sample_patch_batch(batch_size=8, patch=64)

# small CNN: 4ch -> 1ch
model = SmallCNN(in_ch=4, out_ch=1, hidden=32)
loss_fn = nn.MSELoss()
opt = optim.Adam(model.parameters(), lr=2e-3)
```

Training Loop and Outcome (loss curve)

Training loop

We train on randomly sampled spatial patches:

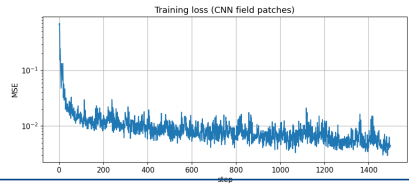
- each step draws a new batch: location + time
- objective: MSE between predicted and true `t2m(t+1)`
- optimizer: Adam

Expected behavior

- loss decreases quickly in the first iterations
- later it stabilizes (noise floor of the synthetic data)
- model learns local transport and correlations

Example:

```
rng = np.random.default_rng(123)
n_steps = 800
batch_size = 8

loss_hist = []

model.train()
for step in range(1, n_steps + 1):
    Xt, yt = sample_patch_batch(batch_size=batch_size, rng=rng)
    pred = model(Xt)
    loss = loss_fn(pred, yt)

    opt.zero_grad()
    loss.backward()
    opt.step()

    loss_hist.append(float(loss.item()))
```



Evaluation: rollout forecast from initial time

One-step skill

We test the CNN on random samples:

- predict `t2m(t+1)` from `X(t)`
- compare to truth with RMSE
- compare to persistence baseline

Rollout forecast

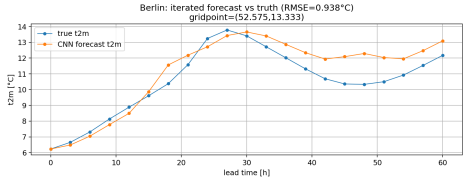
Starting from the initial forecast field:

- use CNN repeatedly: $\hat{x}_{t+1} = f_\theta(\hat{x}_t)$
- build a full forecast trajectory
- then extract town series for comparison

This gives a realistic "mini NWP" demonstration: Zarr archive -> ML training -> forecast rollout .

Example (rollout):

```
# rollout: start from true initial field at t0
t2m_hat = t2m[0].copy()

pred_series = [t2m_hat]
for k in range(0, nt-1):
    Xk = build_features(t2m_hat, u10[k], v10[k], mslp[k])
    t2m_hat = predict_next_field(model, Xk)
    pred_series.append(t2m_hat)

pred_series = np.stack(pred_series, axis=0)  # (time, lat, lon)
```



Outcome: Town Forecasts from ML Rollout

Town-level evaluation

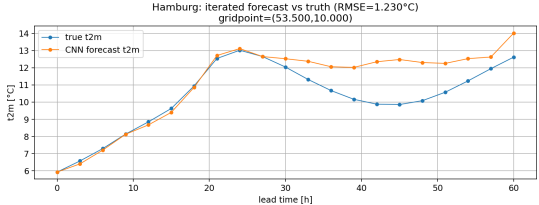
After rollout, we extract time series at town locations:

- nearest neighbor on the lat/lon grid
- compare truth vs ML forecast
- generate one plot per town

Saved outputs

- figures: `fc_<town>.png`
- fast way to compare skill across cities

```
for town, (lat0, lon0) in TOWNS.items():
    # nearest grid point
    pt_true = ds.sel(lat=lat0, lon=lon0, method="nearest")

    # predicted series at same grid point
    y_true = pt_true["t2m"].values[:n_forecast+1]
    y_pred = pred_series[:, iy, ix]   # from rollout
```



Zarr Chunk Decomposition (patch-only visualization)

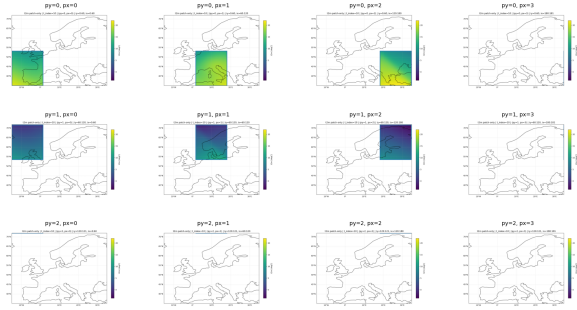
Goal

Visualize the Zarr chunk structure :

- each chunk corresponds to a spatial patch (60x60)
- plot patch values only (no full-field replot)
- use one global color scale (vmin/vmax of full field)

Procedure

- load full field once (for vmin/vmax)
- loop over patch indices (py,px)
- save one image per patch



Patch-only plots: py=0..2, px=0..3 (Europe extent, global color scale).