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	

Live Experiment Tracking — Minimal Example

Goal

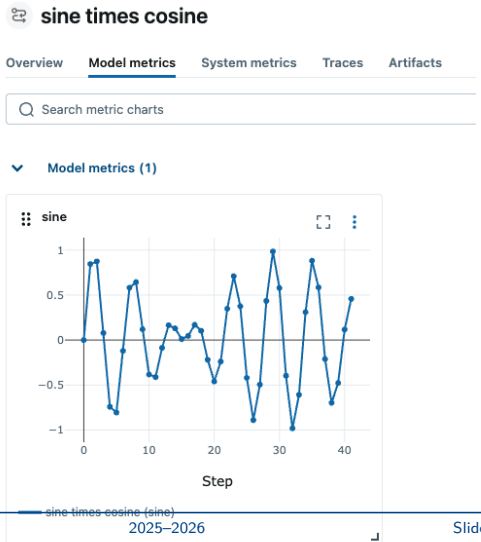
- Run a simple Python script
- Log values every second
- Observe results *live* in MLflow

Key idea

- No model
- No training
- Just time-dependent metrics

This example uses the notebook:

- `1_live_tracking.ipynb`



What Is Logged — And What Is Not

Logged quantities

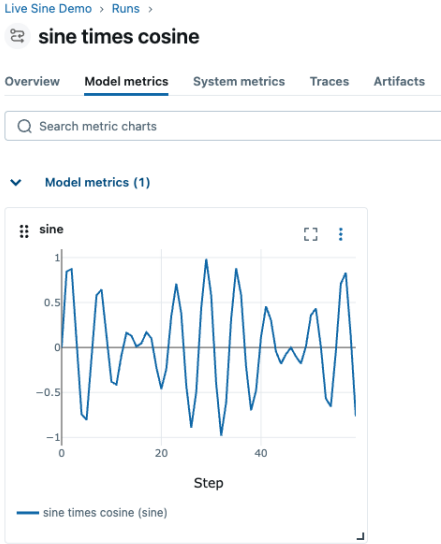
- Time step (logical step index)
- $\sin(t) \cdot \cos(t/10)$

Not logged

- No loss
- No gradients
- No model parameters

Interpretation

- MLflow tracks *numbers over time*
- The meaning is entirely user-defined



What This Example Demonstrates

MLflow is passive

- Python code runs independently
- MLflow only records what it is told

Live monitoring

- Metrics appear while code is running
- UI polls the backend periodically

Key separation

- Execution $\neq$ visualization

Mental model

Python code
$\rightarrow$ MLflow tracking backend
$\leftarrow$ MLflow Web UI

No direct coupling between:

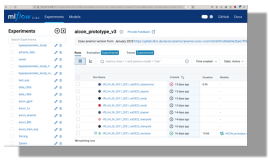
- execution
- user interface

Why Experiment Tracking?

- Repeated training runs
- Changing hyperparameters
- Evolving code versions
- Different datasets

Core requirement

- reproducibility
- transparency



Typical problems without tracking

- Best parameter settings get lost
- Results cannot be reproduced
- Missing or incomplete metadata
- Knowledge is locked in scripts

Common symptoms:

- many scripts with unclear differences
- results that cannot be explained later

Artifacts: Persisting Results Beyond Numbers

What are artifacts?

- Files produced during or after an experiment
- Stored together with parameters and metrics
- Provide context and interpretability

Typical artifacts in ML applications

- Trained model files (`.pt`, `.onnx`, `.pkl`)
- Diagnostic plots (loss curves, skill scores)
- Evaluation outputs (tables, reports)
- Configuration snapshots (YAML, JSON)
- Logs and summaries

Why artifacts matter

- Results are explainable
- Decisions become auditable
- Knowledge is not lost in scripts

How artifacts fit into the workflow

- Research and development
- Repeated experimentation
- Comparison and selection of models
- Preparing models for later use

Key idea

- Metrics answer *how well*
- Artifacts answer *why*

Core Objects: Experiments and Runs

Experiment

- Logical container for related executions
- Identified by name
- Groups comparable runs

Run

- One concrete execution of code
- Has a unique ID
- Records parameters, metrics, files

Key properties

- Named
- Timestamped
- Reproducible

Mental model

Experiment

$\supset$ $\text{Run}_1, \text{Run}_2, \ldots$

Runs differ by:

- parameters
- code state
- data

Comparing Parallel Runs

Setup

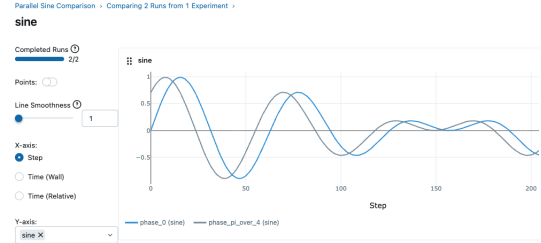
- One experiment
- Two independent runs
- Same metric, same step axis

Difference between runs

- Single parameter: phase
- Everything else identical

Key observation

- MLflow overlays both curves automatically
- No plotting code required




```
def run_sine(phase, name):
  mlflow.set_tracking_uri("http://localhost:5000")
  mlflow.set_experiment("Parallel Sine Comparison")

  with mlflow.start_run(run_name=name):
    mlflow.log_param("phase", phase)
    for step in range(240):
      t = step * 0.1
      value = math.sin(t + phase)*math.cos((t+phase)/10)
      mlflow.log_metric("sine", value, step=step)
      time.sleep(1)
```

Tracking URI

Definition

- URI = `Uniform Resource Identifier`
- address of the MLflow tracking backend
- Decides where runs are written
- Destination of
  - experiments
  - runs
  - metrics
  - artifacts

Keep untouched:

- Training code
- Logging calls

How it is set in code

```
1 import mlflow
2
3 mlflow.set_tracking_uri(
4     "http://localhost:5000")
```

Typical values

- `file:./mlruns` for local storage only
- `http://localhost:5000` for using the mlflow server

The MLflow tracking behavior is fully determined by the tracking URI. If the URI is set to `sqlite:///mlflow.db`, the Python process writes experiment metadata directly into a local SQLite database, without using any HTTP communication or server process. In contrast, if the URI is set to `http://localhost:5000`, all tracking data is sent via HTTP to a running MLflow server, which then stores the results using its configured backend store (e.g. SQLite).

Minimal Logging Example

Essential steps

- Select an experiment
- Start a run
- Log parameters
- Log metrics

Key idea

- One run = one execution
- Everything else is optional

Upload Artefact

```
mlflow.log_artifact("a.png")
```

Minimal MLflow code

```
import mlflow

mlflow.set_experiment("Demo")

with mlflow.start_run():
    mlflow.log_param("lr", 1e-3)
    mlflow.log_metric("loss", 0.42)
```

What this code does

- Creates or selects the experiment `Demo`
- Opens a new run with a unique ID
- Stores one parameter and one metric
- Makes the run visible in the MLflow UI

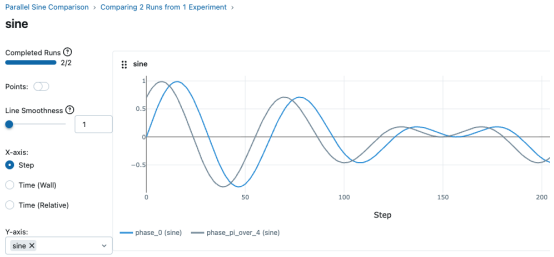
Inspecting Results in the MLflow UI

Grouping principle

- Runs are grouped by experiment
- Experiment name defines the comparison scope

Within one experiment

- Select one or more runs
- Compare metrics
- Inspect parameters
- Browse artifacts



Comparison of two runs within one experiment

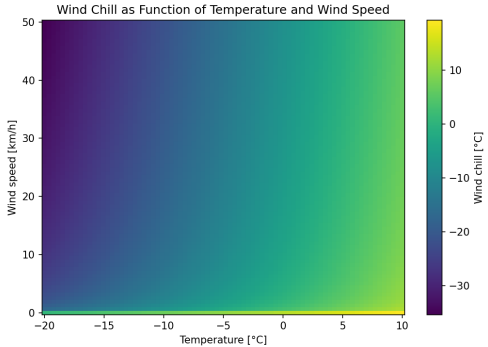
Example Problem: Wind Chill Regression

Problem setup

- Supervised regression task
- Two inputs: temperature, wind speed
- One target: wind chill

Why this example

- Simple but non-trivial
- Continuous target variable
- Suitable for long training runs



Wind Chill Surface

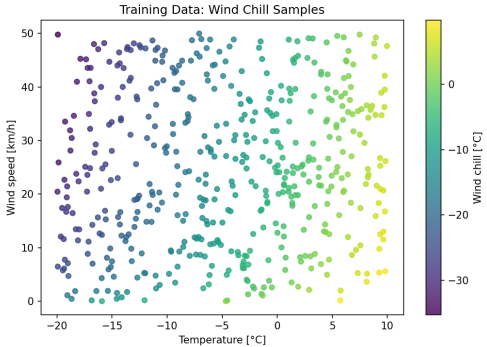
Data Generation and Preprocessing

Synthetic dataset

- Random temperature values
- Random wind speed values
- Wind chill computed from physical formula

Preparation

- Stack inputs into feature vectors
- Convert to tensors
- No normalization tricks required



Input–output relationship

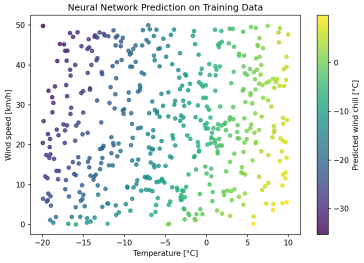
Model Definition

Neural network

- Feedforward architecture
- Two hidden layers
- Scalar regression output

Design choice

- Model kept intentionally simple
- Focus is on tracking, not architecture



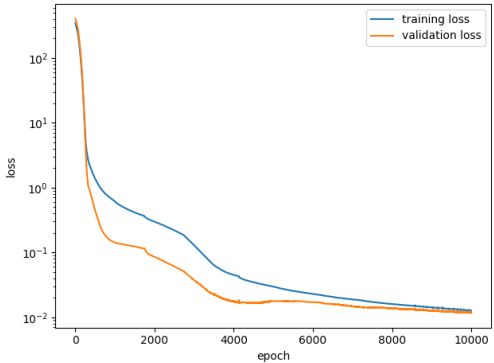
Training with MLflow Logging

What is logged

- Model hyperparameters
- Training loss per epoch
- Continuous progress information

Key benefit

- Training is observable while running
- Long runs become transparent



Live loss tracking in MLflow UI

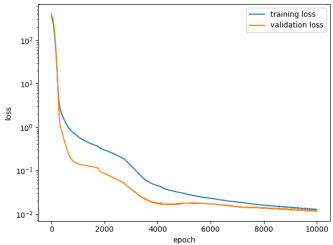
Persisting Results: Artifacts and Models

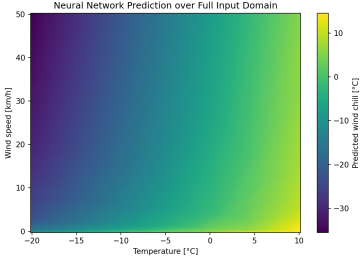
Artifacts

- Loss curves
- Diagnostic plots
- Configuration snapshots

Models

- Serialized network weights
- Input–output signature
- Reusable for later deployment





Local UI Mode

Purpose

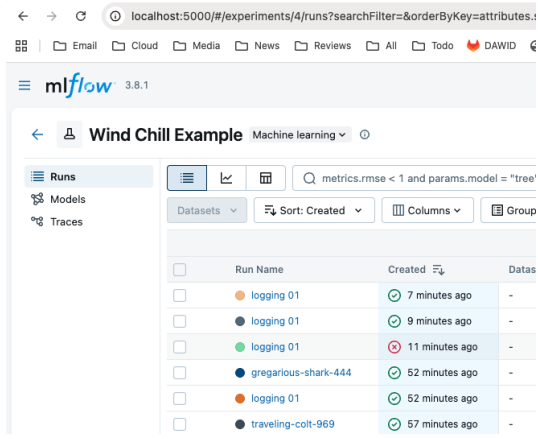
- Inspect completed or running experiments
- Visualize metrics and parameters
- Single-user exploration

Command

- `mlflow ui`

Key property

- Read-only visualization
- No training happens here



By default, the UI reads from the locally configured tracking backend (e.g. `mlruns/` or a local SQLite database).

Server Mode for Collaboration

Why server mode?

- Multiple users
- Multiple machines
- Shared experiment history

Command

- `mlflow server`

Important point

- Training code stays unchanged
- Only the tracking URI changes

Server startup

```
mlflow server \\
  --host 0.0.0.0 \\
  --port 5000
```

Clients connect via `http://<server>:5000` and log runs remotely.

Server Setup

An MLflow server can run on any host reachable by domain name or IP address; clients only need the tracking URI to connect.

Background and Persistent Execution

Why this matters

- Long-running experiments
- Remote machines
- Logout-safe operation

Servers must survive terminals.

These mechanisms are generic system tools for running long-lived services; MLflow behaves like any other server process.

Common patterns

```
mlflow server ... &

nohup mlflow server ... &

screen -S mlflow
```

Use `screen` or `tmux` for interactive re-attachment and monitoring.

```
tmux new -s mlflow          # start a persister
mlflow server --host 0.0.0.0 --port 5000
Ctrl+B  D                   # detach from sessio
tmux attach -t mlflow       # reconnect later
```

Authentication and Credentials

Why authentication?

- Shared infrastructure
- Access control
- Separation of users

MLflow support

- Basic authentication
- Server-side user management

In auth_config.ini:

Credential handling

- Server: auth configuration
- Client: credential file

In auth_config.ini:

```
[mlflow]
auth_enabled = true
database_uri = sqlite:///mlflow_auth.db
admin_username = admin
admin_password = ChangeThisPassword888!
```

```
export MLFLOW_AUTH_CONFIG_PATH=auth_config.ini
export MLFLOW_FLASK_SERVER_SECRET_KEY='8346918649864986498'
mlflow server --host 0.0.0.0 --port 5000 --app-name basic-auth
```

Using a Trained Model from MLflow

Key idea

- Trained models are stored as MLflow artifacts
- A model can be loaded in any notebook or script
- No access to training code is required

Workflow

- Identify the model via its run ID
- **Load the model using an MLflow URI**
- Apply forward inference on new data
- Optionally log new artifacts back to the same run

```
[3]: import mlflow.pytorch
     import torch

     # Point to the model via run_id
     run_id = "50848f8f09f1471bb4070ea8de9076e3"

     model = mlflow.pytorch.load_model(
         f"runs:/{run_id}/model"
     )

     model.eval()

     print(model)

Downloading artifacts:   0%|          | 0/1 [00:00<?, ?it/s]
Downloading artifacts: 100%|██████████| 8/8 [00:00<00:00, 920.79it/s]
wind_chill_model(
  (fc1): Linear(in_features=2, out_features=20, bias=True)
  (fc2): Linear(in_features=20, out_features=20, bias=True)
  (fc3): Linear(in_features=20, out_features=1, bias=True)
  (relu): ReLU()
)
```

Loading a model directly from MLflow

Reproducibility

- Architecture, weights, environment stored
- Input/output signature enforced
- Full traceability to the training run

Model Registry: What Is Stored

Core concept

- A registered model version references exactly one run
- Model versions are immutable
- Full traceability to data and code

Stored per model version

- Trained parameters (weights)
- Model architecture
- MLFlow flavor metadata
- Environment specification
- Optional input/output signature

How the architecture is stored

- Serialized via the MLFlow flavor
- For PyTorch:
  - Python class structure
  - State dictionary (parameter tensors)
  - Loader reference (`mlflow.pytorch`)
- Architecture is reconstructed at load time

When a model is logged from PyTorch, MLFlow stores all information needed to reconstruct the complete neural network at the PyTorch level, including the network structure, trained parameters, and the code required to load the model.

Model Versions and Parallel Development

Versioning principle

- Each model version corresponds to one run
- Versions are ordered but independent
- Older versions remain accessible

What changes between versions

- Training data
- Hyperparameters
- Network architecture
- Optimization settings

Why this matters

- Multiple ideas explored in parallel
- No overwriting of previous results
- Safe comparison and rollback

Model evolution is additive, not destructive.

What “safe rollback” means

- Older model versions remain unchanged
- Switching back requires no retraining
- Deployment can point to any previous version

Model Lineage and Training Continuation

What MLFlow tracks automatically

- Each run has a unique identifier
- Each model version points to exactly one run
- Parameters, metrics, artifacts are immutable

What MLFlow does not infer

- No automatic parent-child model tree
- No implicit notion of fine-tuning or continuation

How lineage is expressed explicitly

- **Log the parent run ID as a parameter or tag**
- Log the source model URI when continuing training
- Use naming and tagging conventions

Example intent

- "run B continues training from run A"
- "model v3 fine-tuned from model v1"

Lineage is recorded by metadata, not guessed.

```
import mlflow
with mlflow.start_run():
    mlflow.set_tag("parent_run_id", "50848f8f09f1471bb4070ea8de9076e3")
    mlflow.set_tag("source_model", "runs:/50848f8f09f1471bb4070ea8de9076e3/model
    mlflow.set_tag("training_type", "fine-tuning")
```

What MLFlow Stores (and What It Does Not)

Stored explicitly

- Parameters and metrics
- Artifacts (plots, files, models)
- Model binaries (framework-specific)
- Environment YAML files
  (only when a model is logged)

YAML handling

- Generated during `log_model`
- Stored as part of the model artifact
- Enables reproducible loading

Not stored automatically

- Full runtime environment
- OS-level dependencies
- GPUs, drivers, system libraries

Important boundary

- MLFlow does not infer environments
- Reproducibility requires explicit logging

MLFlow records what you declare, not what it guesses.

From Tracking to Operations

What MLFlow enables

- Versioned model artifacts
- Stable model identifiers
- Promotion via registry stages
- Rollback to known-good models

Operational contract

- Training produces runs
- Registry exposes deployable models
- Inference consumes model URIs

What MLFlow is not

- Not a deployment platform
- Not a monitoring system
- Not a CI/CD engine

Typical integration

- MLFlow for model lifecycle
- External systems for serving
- External monitoring and alerts

MLFlow provides the training state of a model, not its operational execution.