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	

MLOps: From Machine Learning to Operations

Why this lecture

- Machine learning models are **no longer research prototypes**
- AI systems increasingly enter operational environments
- Reliability, reproducibility and traceability become critical

Traditional ML focuses on *training*.
Operations focus on *stability*.

MLOps connects both worlds.

Key question

- How do we turn ML code into an operational system?

This includes

- controlled software environments
- automated build and deployment
- clear separation of roles and responsibilities
- safe execution on HPC and production systems

This lecture focuses on *processes*, not algorithms.
But also lots of continuous integration (CI).

Why Machine Learning Needs Operations

Classical software assumptions

- Code fully defines behavior
- Same input → same output
- Changes are explicit and infrequent

Once deployed, behavior is largely predictable and stable.

Data Improvement

More & better training data

Bug Fixes

Fix errors in code/pipeline

Machine learning reality

- Behavior emerges from data
- Outputs are statistical, not deterministic
- Models degrade as data distributions change

Operational consequences

- Retraining becomes part of operations
- Monitoring must include model performance
- Reproducibility extends **beyond source code**

Feature Engineering

New features & transformations

Architecture

Improve model architecture

Hyperparameters

Optimize training parameters

Machine learning systems require operational control from the very beginning.

What Is MLOps?

Basic idea

- MLOps extends DevOps principles to machine learning
- Focus shifts from code alone to data and models
- The full model lifecycle becomes operational

MLOps treats ML systems as *long-lived services*, not one-off experiments.

What MLOps manages

- Data pipelines and preprocessing
- Training and validation workflows
- Model versions and metadata
- Deployment, monitoring and rollback

Why this is necessary

- ML behavior changes even when code does not
- Manual processes do not scale
- Missing control leads to silent failures

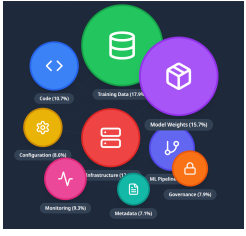
MLOps is not a tool — it is an engineering discipline.

DevOps vs MLOps: What Really Changes

DevOps perspective

- Software behavior defined by source code
- Releases are explicit and controlled
- Bugs are fixed by changing code

How do we monitor systems, and control changes, fixes, improvements?



MLOps extends DevOps — it does not replace it.

MLOps perspective

- Behavior emerges from data and training
- Models evolve through retraining
- Performance can change without code changes

Operational impact

- Data becomes a first-class artifact
- Validation must be continuous
- Rollback applies to models, not just code

The Classical MLOps Cycle

From experiment to operation

- Data ingestion and preprocessing
- Model training and validation
- Deployment into an operational environment

This cycle extends the classical DevOps loop by explicitly including data and models.

Modern MLOps has many elements of typical NWP development cycles. NWP also integrates data, e.g. orography, canopy layers, and real data through observations.

Operational feedback loop

- Monitoring of system and model performance
- Detection of drift and degradation
- Triggering retraining or rollback

Key property

- The cycle is continuous, not linear
- Operations actively influence development

MLOps is a closed loop, not a one-time deployment.

How Automation Is Achieved in MLOps

Automation principles

- Each step is explicitly defined and scripted
- Execution is trigger-based, not manual
- Results are logged and versioned automatically

Automation replaces informal procedures by repeatable execution rules.

Technical mechanisms

- CI/CD pipelines trigger builds and checks
- Version control tracks code, configs, and metadata
- Registries store models and runtime artifacts
- Containers freeze execution environments

Key effect

- Humans decide what should happen
- Systems enforce how it happens

Automation makes MLOps scalable, auditable, and safe.

CI and CD in MLOps

Continuous Integration (CI)

`CI = Every change is automatically built, checked, and validated`

- Triggered by commits to code or configuration
- Executes tests, linters, and consistency checks
- Detects problems early and reproducibly

Continuous Delivery / Deployment (CD)

`CD = Validated artifacts are automatically prepared for operation`

- Packages software, models, and environments
- Produces versioned, deployable artifacts
- Enables controlled rollout or rollback

CI/CD in MLOps

- CI validates logic and structure
- CD delivers operational artifacts

Important distinction

- CI/CD does not decide when models are used
- Operations retain execution control

Automation without loss of control.

MLOps as a Role-Based Pipeline

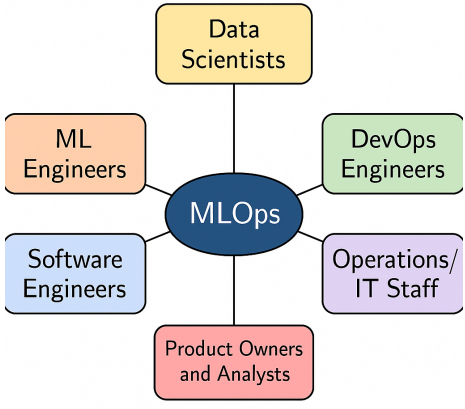
It is a **sequence of responsibilities** that together form a complete lifecycle:

- data preparation and feature design
- model development and validation
- packaging and deployment
- monitoring, feedback and retraining

Each of these steps:

- requires different skills,
- has different risks,
- and implies different ownership.

A useful way to understand MLOps is therefore as a role game, where different actors take responsibility for different parts of the pipeline.



Roles in MLOps Environments

Who Defines What "Good" Means?

In MLOps, quality is not a technical concept.

It is defined by:

- physical constraints,
- domain knowledge,
- operational requirements,
- user expectations.

These aspects cannot be learned from data alone.

Therefore:

- Domain experts define what "good" means,
- ML systems optimize *towards* this definition,
- Operations validate it against reality.

Without a domain expertise, metrics are meaningless.

Domain Expert

- Defines quality and success criteria
- Specifies relevant metrics and constraints
- Interprets model behaviour in context
- Does not implement models or pipelines

Roles in the MLOps Pipeline

Data Scientist

- Develops models to optimize domain criteria
- Explores data and hypotheses
- Translates questions into ML problems
- Does not decide operational relevance

ML Engineer

- Turns models into software artifacts
- Builds training and inference pipelines
- Ensures reproducibility and versioning
- Bridges research and operations

DevOps / Platform Engineer

- Provides CI/CD and build infrastructure
- Manages containers and registries
- Ensures security and scalability
- **Does not tune models**

Domain Expert / Operations / Users

- Run models in production
- Monitor behaviour and failures
- Validate outputs against reality
- Do not modify code or models

Roles should be clear, people may be the same.

Starting Point: A Baseline ML Application

We start with a minimal working ML example.

The goal at this stage is simple:

- load a pretrained model,
- apply it to input data,
- inspect the output.

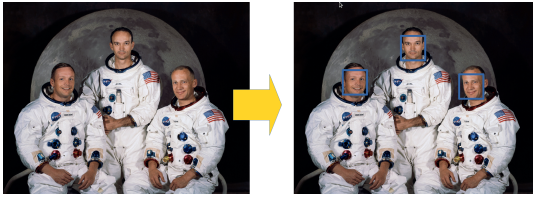
This corresponds to running the notebook:

- `00_face-detection-onnx.py`

At this point:

- the method works,
- results can be inspected visually.

Nothing here is operational yet.



The notebook instantiates a pretrained face-detection model and applies it directly to an input image.

```
def mark_faces(image_filename):
    """Mark all faces recognized in the image"""
    image = PIL.Image.open(image_filename)

    faces = detect_faces(image)

    render_data = detections_to_render_data(
        faces, bounds_color=Colors.GREEN, line_w
```

Input image

Raw input data, loaded via `PIL`. No preprocessing, no metadata, no assumptions beyond file availability.

Model output

Bounding boxes produced by a pretrained face-detection model and rendered directly onto the image.

This is pure inference: load → detect → render.

Docker: Installation and Input Files

If you have admin rights on your computer, installation of docker is easy.

```
brew install --cask docker
```

Step 1: Install Docker

- Install Docker Desktop (Mac)
- Start Docker service
- Verify installation:

Check Docker

```
1 docker --version
2 docker run hello-world
```

If this works, Docker is ready to use .

Step 2: Python code to run in Docker

02_code_to_execute_in_docker.py

```
1 from pathlib import Path
2
3 print("Hello from Docker")
4
5 out = Path("docker_output.txt")
6 out.write_text("Generated inside
      Docker\n")
7
8 print("Wrote:", out)
```

This is ordinary Python . No Docker logic inside the code.

Dockerfile, Image Build, and Execution

Step 3: Define the container

```
FROM python:3.11-slim
WORKDIR /app
COPY 02_code_to_execute_in_docker.py /app/
CMD ["python", "02_code_to_execute_in_docker.py"]
```

Docker freezes the runtime environment:

- operating system (Debian slim)
- Python version (3.11.x)
- system libraries and pip

A Docker image already acts as a virtual environment.
No venv activation is required; packages installed in the image are isolated by default.

Step 4: Build and run

Build image

```
1 docker build -t \
2   python-hello-docker \
3   -f 02_dockerfile.txt .
```

Run container

```
1 docker run --rm \
2   -v "$(pwd):/app" \
3   python-hello-docker
```

Volume mount exposes results. Without -v, files disappear.

Building Docker Containers on GitHub

General principle

- Docker containers do not need to be built locally
- GitHub provides CI runners with Docker installed
- A container can be built directly on the platform

What is required

- Source code
- A Dockerfile
- A CI workflow definition

CI builds containers as first-class artifacts.

The build is moved from the laptop to the platform.

Concrete example (this lecture)

- Python script generates a plot
- Dockerfile defines the runtime
- GitHub Actions builds the image
- Image is pushed to a container registry

Result

- Reproducible container image
- Independent of local setup
- Ready for download and execution

Code lives in Git.
Images live in the registry.

Downloading and Running a Container Image

Pulling a container image

- Images are downloaded via Docker CLI
- Private images require authentication
- Access is granted via registry tokens

Download image

```
docker pull ghcr.io/eumetnet-e-ai/docker-plot-03:latest
```

Important distinction

- Git access $\neq$ registry access
- Separate permissions for code and containers

Build once, run anywhere — with controlled environments.

Running the container

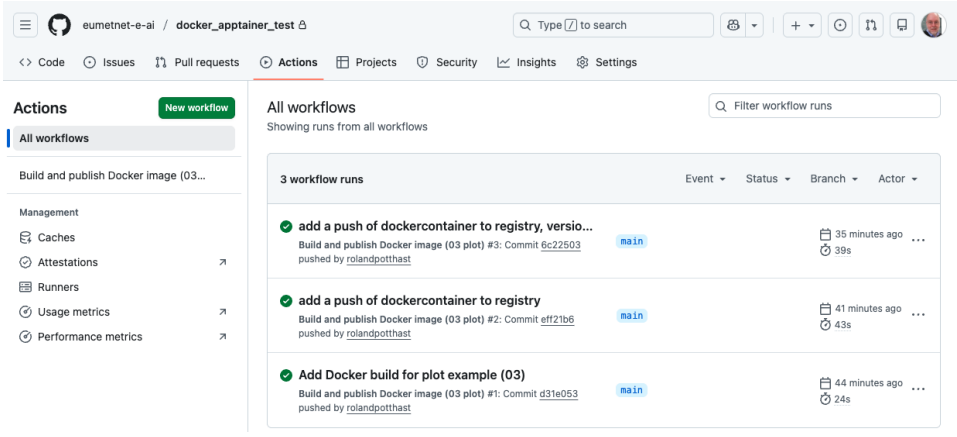
```
docker run --rm \
  -v "$(pwd)/03_docker_plot:/app" \
  ghcr.io/eumetnet-e-ai/docker-plot-03:latest
```

Why the volume mount matters

- Container files are ephemeral
- Volume mounts expose results
- Without `-v`, outputs disappear

Execution is decoupled from development.

Docker Containers Built on GitHub



Source code in Git → CI build on GitHub → Container registry → Pull and run anywhere

GitHub Container Registry (ghcr.io)

What is `ghcr.io`?

- GitHub's container registry service
- Stores Docker / OCI container images
- Integrated with GitHub repositories and CI

How it works

- CI builds an image from a Dockerfile
- Image is pushed to ghcr.io
- Image is identified by name, tag, and SHA

Containers are artifacts, not source code.

Access and usage

- Images are pulled via Docker or Apptainer
- Private images require authentication
- Access control is separate from Git

Browser access

- Containers can be inspected in the browser
- Tags, SHAs, and metadata are visible
- No direct download button

Click to inspect. Pull to execute.

ghcr.io stores images; runtimes execute them.

Apptainer on HPC: Build vs Run

What happens when pulling Docker images

- Apptainer converts Docker images into local containers
- This involves SquashFS compression
- Compression is memory- and thread-intensive

Typical failure on login nodes

- Limited memory and process counts
- SquashFS build cannot create threads

```
Out of memory (frag_thrd)
Failed to create thread
```

Important distinction

- Running containers is lightweight
- Building containers is a compute task

HPC policy (typical)

- Login nodes: editing, submitting jobs
- Compute nodes: builds, compression, heavy work

Apptainer pull = build step, not runtime.

Disk quota is irrelevant — memory and threads decide.

Correct Apptainer Workflow on HPC

Recommended solution

- Run container pulls as batch jobs
- Request sufficient memory
- Avoid compressed SIF builds on login nodes

Use sandbox format

- Uncompressed directory container
- Much lower memory pressure
- Fully usable with Apptainer

Batch job (concept)

```
#PBS -q gp_norm_all
#PBS -l memsz_job=128gb
#PBS -l cpunum_job=2

apptainer build --sandbox \
  my-container \
  docker://ghcr.io/ORG/IMAGE:TAG
```

Running the container

- No root privileges
- Current directory mounted by default
- Same performance as SIF

HPC rule: build elsewhere, run everywhere.

For production: build SIFs in CI, not on the cluster.

Building Apptainer Containers on GitHub

Motivation

- HPC systems often cannot build containers locally
- Docker images must be converted to Apptainer (SIF)
- Conversion is memory-intensive

Key idea

- Move the build step off the HPC
- Use GitHub Actions as build infrastructure
- Produce ready-to-run SIF artifacts

CI builds containers, HPC only runs them.

What GitHub provides

- Linux build nodes with Docker
- Sufficient memory for SquashFS
- Integrated access to ghcr.io

Required inputs

- Docker image in a registry
- Apptainer installed in CI
- Authentication token (`read:packages`)

Result: A portable single-file container.

CI Workflow: Docker Image to Apptainer SIF

Automated workflow

- Triggered after successful Docker build
- Runs on GitHub Linux runner
- Converts image to SIF format

Main steps within the Github environment

1. Install Apptainer
2. Authenticate to `ghcr.io`
3. Pull Docker image
4. Convert to SIF
5. Upload as artifact

This is CI/CD for runtime environments.

Core CI command on GitHub

```
apptainer pull \
  docker-plot-03.sif \
  docker://ghcr.io/ORG/IMAGE:TAG
```

Distribution

- SIF stored as CI artifact
- Downloadable via browser
- Transfer to HPC via scp

On the cluster

- No build step
- No Docker required
- Immediate execution

Triggering the SIF Build via GitHub Actions, YAML

What this workflow does

- Listens for a successful Docker build
- Runs only if the image exists
- Converts Docker → Apptainer

Why this design

- Avoids duplicate builds
- Enforces a clean dependency chain
- CI controls the runtime environment

Docker build is a prerequisite.

This makes the SIF build:

- deterministic, reproducible
- traceable to a commit

```
name: Export Apptainer SIF (Example 03)
on:
  workflow_run:
    workflows: ["Build and publish Docker image (0
    types: [completed]
jobs:
  build-sif:
    if: ${{ github.event.workflow_run.conclusion =
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: read
    steps:
      - name: Install Apptainer
        run: |
          sudo apt-get update
          sudo apt-get install -y \
```

Running the Apptainer Container on the HPC

On the HPC system

- No Docker available
- Apptainer pre-installed
- No build permissions required

Run the container

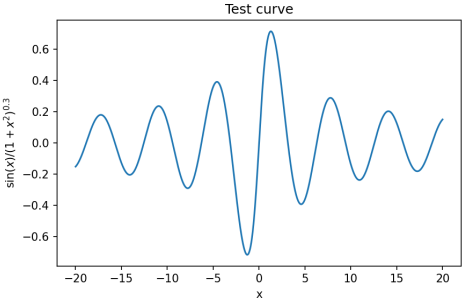
HPC execution

```
1 apptainer exec \
2 docker-plot-03.sif python \
3 -B 03_plot_curve.py
```

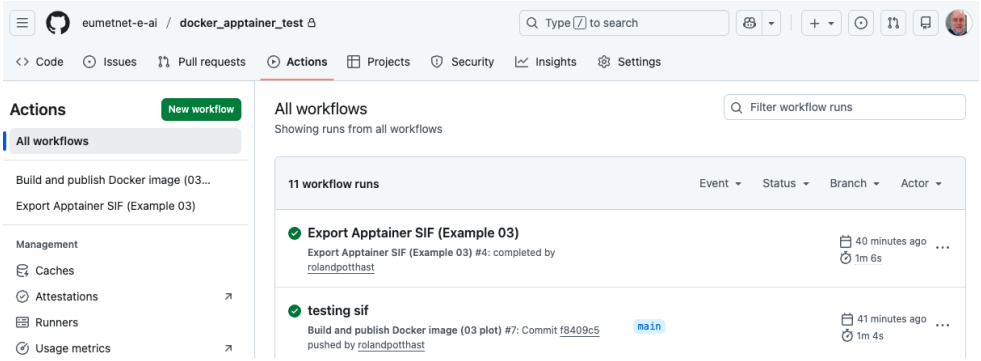
Same container, different system.

What happens

- Python code runs inside container
- Output file written to working directory
- No external dependencies needed



From Docker Image to Apptainer SIF on GitHub



Why Containers are Essential in ML Frameworks

The core problem in ML

- ML code depends on:
  - specific library versions
  - CUDA / CPU features
  - system-level dependencies
- These dependencies change over time
- Results become hard to reproduce

ML models are not standalone artifacts.

What containers provide

- Encapsulation of:
  - code
  - libraries
  - runtime environment
- Identical execution on:
  - laptops
  - CI systems
  - HPC clusters

Containers turn models into executable artifacts.

Reproducibility, portability, and controlled execution are non-negotiable in ML.