Berliner Hochschule für Technik

FB II : Mathematik | Physik | Chemie

Documentation for a MLOps process

Author: Radmir Gesler

Version: 7th February, 2026

Contents

1 Hardware, which has been used

Just to get some information about installed hardware on the current machine, the bash script given in the Listing 1.1 delivers such a value.

Listing 1.1: Bash script for extracting hardware information on a Linux machine

```bash
#!/bin/bash
echo "===== CPU ====="
lscpu | grep -E "Model name|Socket|Thread|CPU\\(s\\)|NUMA|MHz"

echo -e "\n===== MEMORY ====="
free -h

echo -e "\n===== DISKS ====="
lsblk -o NAME,SIZE,TYPE,MOUNTPOINT

echo -e "\n===== GPU ====="
if command -v nvidia-smi &> /dev/null
then
    nvidia-smi
else
    lspci | grep VGA
fi

echo -e "\n===== PCI DEVICES ====="
lspci -nn | grep -E "VGA|3D|Ethernet"

echo -e "\n===== TEMPERATURE SENSORS ====="
sensors
```

In my case it gives me:

```
===== CPU =====
CPU(s):                              36
On-line CPU(s) list:                 0-35
Model name:                          Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz
Thread(s) per core:                  2
Socket(s):                           1
CPU max MHz:                         4800,0000
CPU min MHz:                         1200,0000
NUMA node(s):                        1
NUMA node0 CPU(s):                   0-35

===== MEMORY =====
total        used        free      shared  buff/cache   available
Mem:           251Gi       4,1Gi       172Gi        19Mi        74Gi       245Gi
Swap:          2,0Gi          0B       2,0Gi

===== DISKS =====
NAME          SIZE TYPE MOUNTPOINT
sda           9,1T disk /data
nvme0n1     953,9G disk
|--nvme0n1p1   512M part /boot/efi
|--nvme0n1p2 953,4G part /

===== GPU =====
Wed Oct 15 15:45:53 2025
+-----------------------------------------------------------------------+
| NVIDIA-SMI 580.65.06  Driver V.: 580.65.06      CUDA Version: 13.0   |
+----------------------+------------------------+----------------------+
| GPU  Name            | Bus-Id          Disp.A |               Memory |
|======================+========================+======================+
|   0  Quadro RTX 8000 |   00000000:65:00.0 Off |            49152MiB |
+----------------------+------------------------+----------------------+
|   1  Quadro RTX 8000 |   00000000:B6:00.0 Off |            49152MiB |
+----------------------+------------------------+----------------------+

===== PCI DEVICES =====
01:00.0 Ethernet [0200]: Intel I210 Gigabit Network Connection [8086:1533] (rev 03)
02:00.0 Ethernet [0200]: Intel I210 Gigabit Network Connection [8086:1533] (rev 03)
65:00.0 VGA [0300]: NVIDIA TU102GL [Quadro RTX 6000/8000] [10de:1e30] (rev a1)
b6:00.0 VGA [0300]: NVIDIA TU102GL [Quadro RTX 6000/8000] [10de:1e30] (rev a1)
```

2 What is MLOps?

The goal of this summary is to understand the motivation behind **MLOps (Machine Learning Operations)**, its emergence as a discipline, and how it extends the principles of classical DevOps to the field of Machine Learning. We will distinguish between the research-oriented workflow typical for data scientists and the engineering-oriented workflow required to deploy and maintain machine learning systems at scale.

MLOps aims to close the gap between experimental model development and robust, automated, and maintainable production environments. It provides the methodological and technological foundation for *reproducible, automated, and continuously improved* ML systems.

Motivation

In traditional ML practice, models are trained and evaluated manually, often in Jupyter notebooks, on local data subsets. This works well for research and prototyping but fails when moving toward production due to:

- Lack of reproducibility (no fixed data versions, ad-hoc code changes).
- Manual retraining and deployment steps.
- Missing automation, monitoring, and feedback loops.
- Divergence between training and production environments.

MLOps introduces structure and automation similar to software engineering. It formalizes how models are trained, validated, deployed, monitored, and retrained, thus ensuring both scientific rigor and operational reliability.

2.1 Core Concepts

ML Research vs. ML Engineering

The difference between ML Research and ML Engineering can discussed as it shown in the table 2.1. But we can summarize it also as:

- **ML Research:** Focuses on designing algorithms, optimizing architectures, and proving performance improvements on benchmark datasets. The emphasis is on exploration, hypothesis testing, and innovation.
- **ML Engineering:** Concerned with building reliable, scalable, and maintainable ML systems. The emphasis is on reproducibility, continuous integration, monitoring, and lifecycle management.

While ML research is often experimental, ML engineering requires controlled processes and automation — comparable to the relationship between theoretical science and industrial engineering.

Aspect	**ML Research**	**ML Engineering**
Primary goal	Designing new algorithms and methods	Building reliable, scalable, and production-ready ML systems
Focus	Exploration, hypothesis testing, model innovation	System robustness, reproducibility, lifecycle management
Typical questions	"Can we improve accuracy?" "Does this architecture generalize better?"	"Can we deploy and maintain this reliably?" "How does it behave in production?"
Success criteria	Performance on benchmarks, novel contributions	Stability, scalability, maintainability, and monitoring effectiveness
Key activities	Experimentation, algorithm design, empirical evaluation	CI/CD integration, monitoring, retraining, versioning, deployment
Main emphasis	Innovation and theory	Infrastructure and operational excellence

Table 2.1: Comparison between Machine Learning Research and Machine Learning Engineering

As an example we will show why a Jupiter Notebook isn't a production system. Jupyter notebooks are excellent for interactive exploration but have inherent limitations:

- They lack version control for data and parameters.
- Execution order is not guaranteed, leading to hidden state issues.
- Manual execution cannot scale or be monitored.
- Integration with CI/CD, monitoring, and production APIs is minimal.

A notebook is a sandbox; production requires *pipelines*, *versioned artifacts*, and *infrastructure integration.*

2.2 The MLOps Pyramid

The conceptual structure of MLOps can be represented as a pyramid of five interdependent layers:

1. **Data Layer:** Reliable data ingestion, validation, and versioning. Tools such as DVC, LakeFS, or Delta Lake ensure reproducibility of datasets.
2. **Model Layer:** Training, tracking, and storing models in registries (e.g., MLflow, ModelDB). This layer provides reproducible artifacts.
3. **CI/CD Layer:** Continuous Integration and Continuous Deployment pipelines (e.g., GitLab CI, Jenkins, ArgoCD) automate testing and release cycles.
4. **Monitoring Layer:** Continuous observation of models in production (Prometheus, Grafana, Evidently AI). Detects performance degradation and drift.

5. **Governance Layer:** Enforces compliance, audit trails, explainability, and ethical considerations for model usage.

All the points above are summarized in the figure 2.1 as an image impression of the structural dependence of all components.

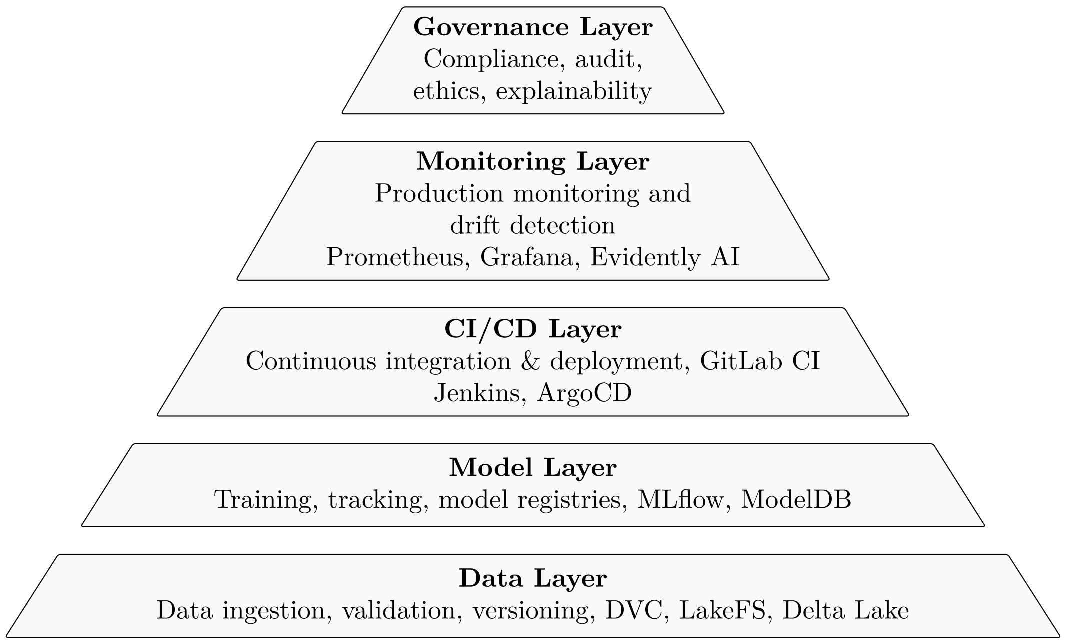

Governance Layer
Compliance, audit, ethics, explainability
Monitoring Layer
Production monitoring and drift detection
Prometheus, Grafana, Evidently AI
CI/CD Layer
Continuous integration & deployment, GitLab CI
Jenkins, ArgoCD
Model Layer
Training, tracking, model registries, MLflow, ModelDB
Data Layer
Data ingestion, validation, versioning, DVC, LakeFS, Delta Lake

Figure 2.1: Conceptual MLOps pyramid with interdependent layers from data management to governance.

Each layer builds upon the previous one, forming a hierarchy of reproducibility and automation — from raw data to responsible deployment. We will keep it in mind and dive deeper in the next part of current text to understand what the key principle of MLOps are.

2.2.1 Key Principles of MLOps

MLOps is driven by several foundational principles:

Reproducibility.

Every experiment, dataset, model, and environment must be uniquely identifiable and reproducible at any time. This ensures scientific integrity and traceability.

Automation (Pipelines).

Training, validation, and deployment steps should be automated via pipelines (Airflow, Kubeflow, Tekton). Manual intervention is minimized.

Version Control.

Not only source code, but also data, model parameters, and configurations must be versioned. Git is extended by tools like DVC and MLflow registries.

Continuous Training (CT).

Analogous to Continuous Integration in software, models are retrained continuously as new data becomes available, keeping them current with evolving environments.

Model Drift and Retraining.

Deployed models degrade as data distributions shift ("drift"). MLOps integrates drift detection, monitoring, and automated retraining workflows.

While, I guess all points are intuitively understandable, but the *Model Drift* isn't. In the next part we will shortly introduce this phenomena.

2.2.2 Model Drift and Automated Retraining

Deployed machine learning models operate in non-stationary environments. While the training process assumes that the joint distribution

$$P_{\text{train}}(X, Y) \approx P_{\text{future}}(X, Y),$$

this assumption breaks down as physical conditions, data acquisition processes, or system dynamics evolve over time.

Such changes lead to ***Model Drift***, i.e., a degradation of predictive performance and physical consistency if no corrective mechanisms are introduced.

Types of Drift

Model degradation can be categorized into three fundamental drift classes:

- **Data Drift (Covariate Shift):** The marginal input distribution changes while the underlying mapping remains stable:

  $$P_{\text{train}}(X) \neq P_{\text{live}}(X), \quad P(Y \mid X) \approx \text{const.}$$

  This can be detected using statistical divergence measures such as the Kolmogorov–Smirnov test, Population Stability Index (PSI), or Wasserstein distance.
- **Concept Drift:** The conditional distribution changes:

  $$P_{\text{train}}(Y \mid X) \neq P_{\text{live}}(Y \mid X),$$

  which may occur in physical systems due to material aging, changing system dynamics, or updated operational conditions.
- **Model Drift:** Even if the environment is stable, the model may become miscalibrated or insufficient due to limited capacity, dataset bias, or evolving rare events.

This overview should be enough to get the idea behind Model Drift, for more deeply information see the chapter 7.

Summary

In the context of this work, MLOps bridges the gap between experimental machine learning research and production-grade machine Learning systems. It introduces a rigorous lifecycle based on versioned data, tracked experiments, automated pipelines, continuously monitored deployments, and governed model operation.

Beyond the conceptual illustration of MLOps, the current work is accompanied by a concrete example derived from my master's thesis. The objective of that thesis was to develop a machine learning procedure for solving an *inverse heat conduction problem*, specifically to estimate a *spatially dependent thermal diffusivity field*. In this work, this approach is extended into an MLOps context, with the goal of designing a production-capable system for predicting thermal diffusivity using automated and reproducible machine learning workflows.

In the context of machine learning for physical system modeling, several approaches exist. One particularly relevant method is the class of *Physics-Informed Neural Networks* (PINNs). During my thesis, I developed a PINN to solve the above-mentioned inverse heat problem. This method and its theoretical foundation will be briefly introduced in the following section.

2.3 Example: Scientific Machine Learning (PINNs)

To illustrate, consider the inverse heat conduction problem:

$$\partial_t u - \nabla \cdot (a(x,y)\nabla u) = f(x,y,t),$$

where the goal is to infer the spatially varying diffusivity $a(x,y)$ and possibly the source term $f(x,y,t)$ from observed temperature data $u(x,y,t)$.

The developed PINN system can be summarized as it is done in figure 2.2. If some is more interested in details of my work please see [1]. But for the terms of ML the Loss equations are important should be introduced here:

$$\mathcal{L}_{u_{\mathrm{D}}} = \frac{1}{m}\|\hat{u}_l - u_D\|_2^2 \tag{2.1}$$

$$\mathcal{L}_{f_{\mathrm{D}}} = \frac{1}{m}\|\hat{f}_l - f_D\|_2^2 \tag{2.2}$$

The above loss functions are used to train the models for the temperature u and/or heat source f. Using them the main loss function for PINN can be calculated:

$$\mathcal{L}_{PDE} = \frac{1}{n}\|(\tilde{u}_l)_t - \langle \nabla, \tilde{a}\nabla\tilde{u}\rangle - \tilde{f}\|_2^2 \tag{2.3}$$

That concludes this short introduction to the accompanying example. It is sufficient to assume that there exist some **code sources** that can be used as a basis and integrated into the MLOps system. The MLOps system will be designed to accept and analyze the measured data, to train the PINN, to use prediction to generate values and to monitor the reached results and PINN itself. The MLOps system will be derived step by step according to the chapters of this work. But firstly we will take a look how the PINN looks like in raw state as it now is and how it will be after the MLOps system is developed:

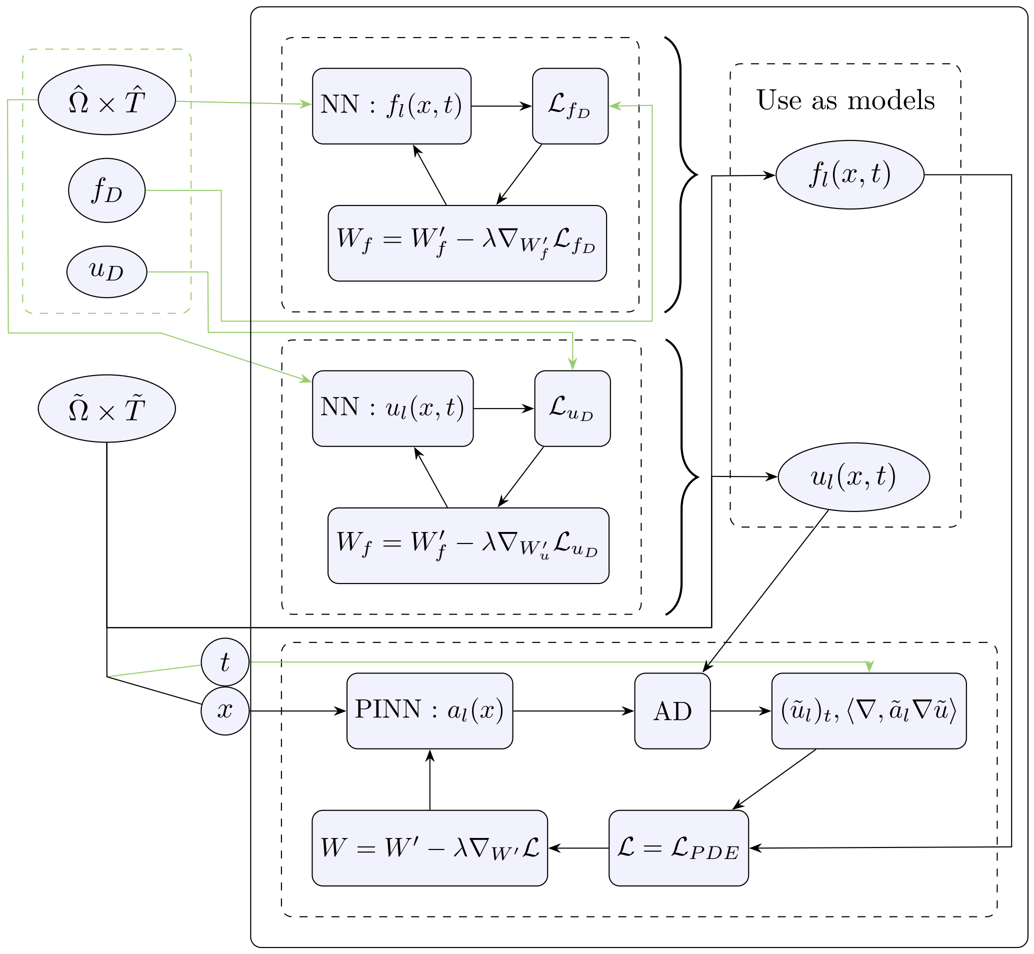
Without MLOps

- Data preprocessing, network training, and validation occur manually.
- Each retraining (e.g., when new measurements arrive) requires manual repetition.
- No model versioning or metadata tracking — results are difficult to reproduce.

With MLOps

- **Data Ingestion:** Experimental measurements are automatically collected and versioned (e.g., via DVC).
- **Training Pipeline:** Airflow triggers a PyTorch/DeepXDE training job on GPUs, logging all metrics to MLflow.
- **Model Registry:** The trained PINN is stored with version and metadata.
- **Deployment:** The model is deployed as an API service (e.g., KServe or FastAPI) for simulation queries.
- **Monitoring:** Drift detectors compare new observations with model predictions.
- **Retraining:** A CI/CD trigger starts retraining when deviation exceeds a threshold.

This workflow transforms a research prototype into a maintainable scientific software system — reproducible, observable, and scalable.



$\hat{\Omega} \times \hat{T}$
f_D
u_D
NN : $f_l(x,t)$
$\mathcal{L}_{f_D}$
$W_f = W'_f - \lambda \nabla_{W'_f} \mathcal{L}_{f_D}$
Use as models
$f_l(x,t)$
NN : $u_l(x,t)$
$\mathcal{L}_{u_D}$
$W_f = W'_f - \lambda \nabla_{W'_u} \mathcal{L}_{u_D}$
$u_l(x,t)$
$\tilde{\Omega} \times \tilde{T}$
t
x
PINN : $a_l(x)$
AD
$(\tilde{u}_l)_t, \langle \nabla, \tilde{a}_l \nabla \tilde{u} \rangle$
$W = W' - \lambda \nabla_{W'} \mathcal{L}$
$\mathcal{L} = \mathcal{L}_{PDE}$

Figure 2.2: Separated training of the Physics-Informed Neural Network (PINN) for the inverse heat conduction problem with respect to thermal diffusivity. The neural networks for u_l and f_l are trained in independent training loops using the datasets $\hat{\Omega} \times \hat{T}$ and the measurement data f_D and u_D, respectively. After this pretraining phase, the actual PINN training is carried out on the dataset $\tilde{\Omega} \times \tilde{T}$, where the pretrained models u_l and f_l are used to compute the PDE residual. The PINN itself receives only spatial inputs from $\tilde{\Omega}$. The green arrow originating from the node t represents the temporal consistency of the training data and supplies the corresponding time values required for evaluating the derivatives of u_l.

3 Core MLOps Architecture

The objective of this module is to understand how the core components of an MLOps ecosystem fit together, and how containerization, orchestration, and automation form the foundation of scalable, reproducible machine learning systems. We will explore the relationship between **Docker**, **Kubernetes**, and **OpenShift** as infrastructural layers, and how these integrate with higher-level ML services such as **Airflow**, **MLflow**, and **Kubeflow**.

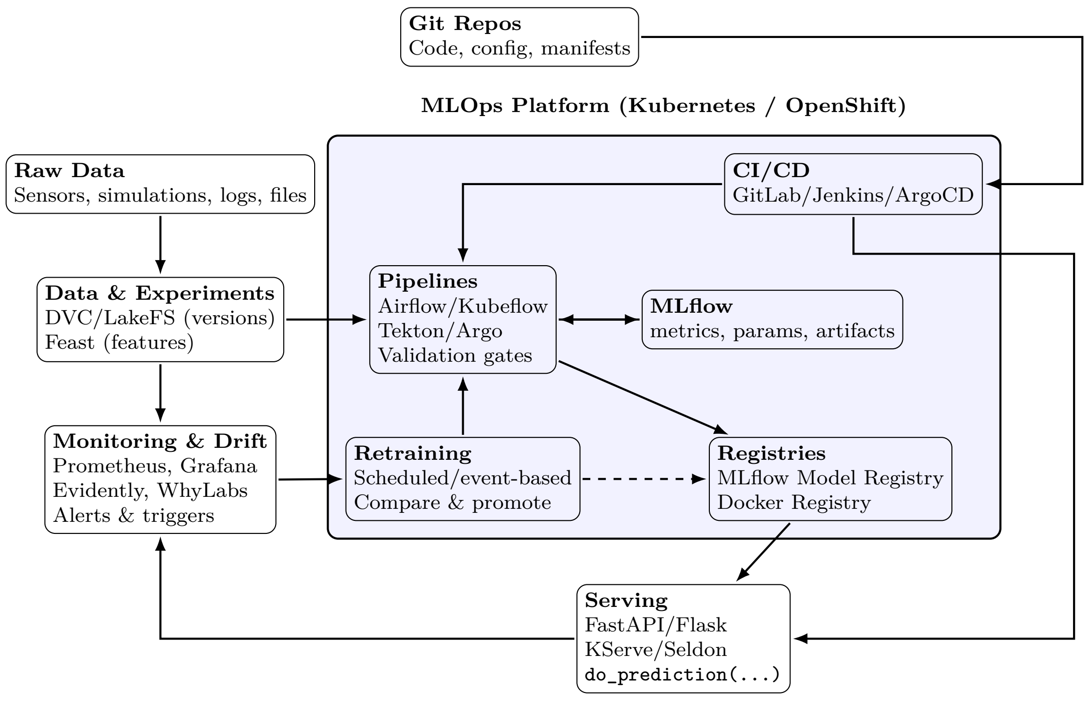


Figure 3.1: End-to-end MLOps architecture for an abstract problem `do_prediction(...)` which cares about: data, pipelines, deployment, monitoring, and retraining.

Stack Overview

At the heart of modern MLOps lies the idea of **infrastructure abstraction**: machine learning workloads are encapsulated inside containers and orchestrated across clusters in a reproducible and portable manner. A typical stack can be described as follows:

- **Docker + Kubernetes** provide environment isolation, scalability, and infrastructure abstraction.
- **Airflow**, **Kubeflow**, and **MLflow** integrate to deliver automation, tracking, and governance.
- **Declarative Pipelines** (YAML/JSON, Argo, Tekton) define reproducible workflows as code.

- **GitOps** principles ensure versioned, auditable, and automatically deployed infrastructure.

These layers work together to move from experimentation to production while maintaining scientific rigor, operational stability, and continuous improvement.

3.1 From Containers to Clusters: Docker, Kubernetes, and OpenShift

Docker: The Fundamental Building Block

Docker provides a lightweight mechanism to package applications, dependencies, and environments into isolated *containers*. A Docker container behaves like a minimal operating system containing exactly what the program needs to run, ensuring consistent behavior across development, testing, and production.

Conceptual analogy:

Each container is like a *single laboratory box*—self-contained, portable, and isolated. It has:

- Its own filesystem (based on the Docker image).
- Its own process space and networking.
- Shared access to the host's Linux kernel.

Why this matters for MLOps:

Reproducibility is central to scientific ML. A Docker container guarantees that a trained model behaves identically whether it runs on a local workstation, a GPU server, or a cloud node.

Example:

```
docker build -t pinn-trainer .
docker run --gpus all -v ./data:/data pinn-trainer
```

This command builds and launches a reproducible environment to train a Physics-Informed Neural Network (PINN) on GPU hardware.

Docker Compose: Managing Multiple Containers

As soon as an ML workflow involves multiple services—such as a training script, a database, and a monitoring dashboard—**Docker Compose** provides a unified configuration layer.

Example:

```
services:
  mlflow:
    image: ghcr.io/mlflow/mlflow:v2.16.0
    ports: ["5000:5000"]
  postgres:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: example
```

Compose simplifies orchestration on a single machine, but it remains limited to one host.

Kubernetes: Orchestrating Containers Across Clusters

When workloads must scale across multiple machines, **Kubernetes (K8s)** acts as an *orchestrator*. It automatically deploys, scales, and monitors containers across a distributed cluster of nodes.

Key Kubernetes concepts:

- **Pod:** Smallest deployable unit (one or more containers).
- **Node:** A physical or virtual machine where Pods run.
- **Deployment:** Blueprint for desired number of replicas and version management.
- **Service:** Provides a stable network endpoint for distributed Pods.
- **Ingress:** Manages external access (HTTP/HTTPS routing).

Kubernetes ensures high availability, self-healing, and dynamic scaling, forming the operational core of most MLOps infrastructures.

Analogy:

If each Docker container is a *laboratory box*, Kubernetes is the *facility manager*—it allocates space, restarts failed experiments, and keeps the environment balanced.

OpenShift: The Enterprise Platform Layer

OpenShift is Red Hat's enterprise-grade platform built on top of Kubernetes. It provides a more opinionated, secure, and user-friendly ecosystem by integrating:

- Developer tooling (builds, web console, source-to-image).
- CI/CD (Tekton pipelines, ArgoCD).
- Security and governance (OAuth, RBAC, quotas).
- Monitoring and logging (Prometheus, Grafana, EFK).

Analogy:

Kubernetes is the engine; OpenShift is the fully assembled car with integrated dashboards, autopilot, and safety systems.

Architectural Layering:

```
|--------------------------------|
| OpenShift (Platform Layer)     |
|   |-- Developer Tools, CI/CD   |
|   |-- Security, Monitoring     |
|---------------|----------------|
                |
|---------------|----------------|
| Kubernetes (Orchestration)     |
|   |-- Pods, Deployments, Nodes |
|   |-- Services, Ingress        |
|--------------|----------------|
               |
|--------------|----------------|
| Docker / CRI-O (Runtime)      |
|  |-- Images, Containers       |
|--------------|----------------|
               |
|--------------|----------------|
| Host OS (e.g., Ubuntu)        |
|-------------------------------|
```

This stack provides a full abstraction from hardware to application-level MLOps orchestration.

Architecture Layout

An MLOps system can be conceptualized as multiple layers, from infrastructure to model governance:

```
+------------------------------------------------------------+
|               Model Governance & Monitoring                |
|      (Evidently, Prometheus, Grafana, WhyLabs, etc.)      |
+------------------------------------------------------------+
|                   Continuous Deployment                    |
|    (ArgoCD, Jenkins, GitLab CI/CD, Tekton Pipelines)      |
+------------------------------------------------------------+
|             ML Workflow Orchestration Layer                |
| (Airflow, Kubeflow Pipelines, Prefect, Dagster, etc.)     |
+------------------------------------------------------------+
|            Experiment Tracking & Model Registry            |
|                   (MLflow, ModelDB)                        |
+------------------------------------------------------------+
|                  Containerized Training                    |
|            (Docker, Kubernetes, OpenShift)                 |
+------------------------------------------------------------+
|               Data Versioning and Ingestion                |
|           (DVC, LakeFS, Delta Lake, Feast)                 |
+------------------------------------------------------------+
|                      Hardware Layer                        |
|               (CPU, GPU, TPU clusters, Cloud)              |
+------------------------------------------------------------+
```

Each layer can evolve independently but interacts via versioned interfaces and declarative specifications.

Discussion

Why Kubernetes/OpenShift Forms the Backbone.

Kubernetes abstracts away infrastructure complexity. It treats compute nodes as a uniform resource pool, automatically managing container scheduling, scaling, and failover. Open-

Shift extends this with enterprise features—security, governance, multi-tenancy—making it suitable for research labs, production environments, and hybrid cloud deployments.

Containerization as a Reproducibility Mechanism.

Every experiment or training run can be encapsulated into a container image, ensuring identical results across hardware or environments. This property is critical in scientific ML, where reproducibility is as important as accuracy.

CI/CD and Infrastructure-as-Code Integration.

MLOps inherits from DevOps the principles of automation and version control:

- **CI (Continuous Integration)** ensures that code changes are tested and validated automatically.
- **CD (Continuous Deployment)** enables fast, safe delivery of new models and services.
- **Infrastructure as Code (IaC)** expresses the entire system (nodes, pods, services, pipelines) as declarative YAML or JSON—enabling complete environment recreation.

Together, these mechanisms make ML systems scalable, observable, and maintainable—transforming one-time experiments into continuously evolving, production-grade scientific platforms.

3.2 Airflow, Kubeflow, and MLflow: Integration Patterns

Roles and Interfaces

- **Airflow** orchestrates the *end-to-end workflow*: data ingestion, validation, triggering training, post-processing, and deployment gates. It is task-centric and environment-agnostic.
- **Kubeflow (Pipelines + Katib + KServe)** executes *ML-specific jobs* on Kubernetes/OpenShift: GPU training, hyperparameter tuning, and model serving with autoscaling.
- **MLflow** provides *experiment tracking* and a *model registry*: parameters, metrics, artifacts, and model versions for governance and rollbacks.

3.2.1 Data Flow.

Airflow triggers a Kubeflow Pipeline (KFP) step that runs the GPU training container for the PINN. That step logs parameters/metrics/artifacts to MLflow. After evaluation, Airflow promotes the best run into the MLflow Model Registry and triggers a KServe deployment from the registered artifact.

End-to-end PINN Example (Research $\rightarrow$ Production)

Suppose we have:

- `pinn_model.py` defining $u_\theta(x, y, t)$, PDE residuals, and loss terms.
- `train.py` training the PINN on GPUs and logging to MLflow.
- `serve.py` (optional) packing a prediction endpoint for offline serving.

Airflow DAG (high-level).

```
@dag(schedule_interval="@daily", catchup=False)
def inverse_heat_mlops():
    prepare_data()     # ETL, validation, dataset versioning
    run_kfp_training() # trigger Kubeflow Pipeline job on GPU
    evaluate_runs()    # pick best MLflow run by metric (e.g., PDE residual)
    register_model()   # promote to MLflow Model Registry (stage="Staging")
    deploy_kserve()    # rolling deploy via KServe from the Registry artifact
```

Kubeflow Pipeline step (pseudo-spec).

```
component:
  name: Train PINN
  inputs: [dataset_uri, mlflow_uri, epochs, lr, a_reg, f_reg]
  container:
    image: registry/app/pinn-trainer:latest
    command:
      - python
      - /app/train.py
      - --data={{inputs.dataset_uri}}
      - --epochs={{inputs.epochs}}
      - --lr={{inputs.lr}}
      - --mlflow={{inputs.mlflow_uri}}
      - --a-reg={{inputs.a_reg}}
      - --f-reg={{inputs.f_reg}}
    resources:
      limits: {nvidia.com/gpu: "1"}
```

MLflow logging within `train.py` (excerpt).

```
import mlflow, mlflow.pytorch as mpt
mlflow.set_tracking_uri(os.environ.get("MLFLOW_TRACKING_URI"))
mlflow.set_experiment("inverse-heat-pinn")

with mlflow.start_run(run_name="pinn-gpu"):
    mlflow.log_param("epochs", epochs)
    mlflow.log_param("lr", lr)
    mlflow.log_param("a_reg", a_reg)
    mlflow.log_param("f_reg", f_reg)
    # ... training loop; log losses and PDE residuals
    mlflow.log_metric("train_residual", residual_value, step=epoch)
    # save model + artifacts (e.g., plots, checkpoints)
    mpt.log_model(model, artifact_path="model")
```

Promotion & deployment gate.

After training completes, a gate task in Airflow queries MLflow for the best run (e.g., minimal PDE residual or validation error). If thresholds are met, Airflow:

1. Promotes the run in the MLflow *Model Registry* (e.g., from “None” to “Staging” or “Production”).

2. Triggers a *KServe* deployment pulling the artifact from the Registry storage (PVC/S3).

Declarative Pipelines (YAML/JSON, Argo, Tekton)

MLOps pipelines should be *infrastructure-as-code*. Declarative specs ensure repeatability, reviews, and auditability.

Examples

Argo Workflow snippet (conceptual).

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata: { name: inverse-heat-pinn }
spec:
  entrypoint: main
  templates:
  - name: main
    steps:
    - - name: etl
        template: etl
      - name: train
        template: train
      - name: evaluate
        template: evaluate
      - name: deploy
        template: deploy
  - name: train
    container:
      image: registry/app/pinn-trainer:latest
      command: ["python","/app/train.py","--mlflow=http://mlflow:5000"]
```

Tekton Pipeline snippet (OpenShift Pipelines).

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata: { name: pinn-train-deploy }
spec:
  tasks:
  - name: train
    taskRef: { name: python-task }
    params:
    - name: image
      value: registry/app/pinn-trainer:latest
    - name: args
      value:
      - python
      - /app/train.py
      - --mlflow=$(params.mlflow_uri)
  - name: deploy
    runAfter: ["train"]
    taskRef: { name: kserve-deploy-task }
```

By storing these YAML specs in Git, we enable reviews, rollbacks, and reproducible automation across environments.

GitOps Principles

GitOps manages both *applications* and *infrastructure* through Git as the single source of truth. A controller (e.g., Argo CD) reconciles the live cluster state to match the Git state.

Core Practices

- Store Kubernetes/OpenShift manifests (Deployments, Services, Ingress/Routes, KServe, Tekton) in Git.
- Use branches/tags for *environments* (e.g., `dev`, `staging`, `prod`).
- Rely on *pull requests* for all changes (audit trail, review).
- Argo CD continuously compares live cluster state with Git and applies drifts back to the desired config.

Benefit for PINNs.

Changes in PDE loss weights, architecture, or serving resources (e.g., GPUs, memory) are versioned and reviewed. Rollbacks are trivial, and experiments map cleanly to infrastructure changes.

How OpenShift Integrates with Airflow, Kubeflow, and MLflow

OpenShift as the Enterprise Kubernetes Layer

OpenShift provides:

- **Security & Auth**: OAuth, RBAC, SCCs (Security Context Constraints).
- **Builds & CI/CD**: OpenShift Pipelines (Tekton), OpenShift GitOps (Argo CD).
- **Registry & Routes**: Integrated image registry and HTTP(S) routing via Routes.
- **Monitoring**: Prometheus/Grafana baked in; EFK for logs.

Running Airflow on OpenShift

- Package Airflow components as Deployments (webserver, scheduler, workers) with a `ClusterIP` Service and an OpenShift `Route`.
- Workers can request GPUs (`nvidia.com/gpu`) if PINN training is executed inside Airflow tasks.
- Airflow tasks trigger Kubeflow Pipelines via KFP SDK or Argo/Tekton via API.

Airflow Deployment (conceptual).

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: airflow-worker }
spec:
  template:
    spec:
      containers:
      - name: worker
        image: registry/airflow:2.10
        resources:
          limits: { nvidia.com/gpu: "1" }
```

Kubeflow / KServe on OpenShift

- Use **KServe** to serve models from MLflow artifacts (PVC, S3, or object storage).
- Auto-scaling via Knative; traffic management via OpenShift Routes/Ingress.

KServe `InferenceService` for PINN (conceptual).

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata: { name: pinn-predictor }
spec:
  predictor:
    pytorch:
      storageUri: s3://mlflow-artifacts/inverse-heat-pinn/Production
      resources:
        limits: { nvidia.com/gpu: "1", cpu: "2", memory: "8Gi" }
```

MLflow on OpenShift

- Run MLflow as a Deployment with a PersistentVolumeClaim for `backend-store` (e.g., SQLite/PostgreSQL) and `artifact-store` (PVC/S3).
- Expose MLflow UI via an OpenShift `Route`.

MLflow Deployment (conceptual).

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: mlflow }
spec:
  template:
    spec:
      containers:
      - name: mlflow
        image: ghcr.io/mlflow/mlflow:v2.16.0
        args: ["mlflow","server","--host","0.0.0.0","--port","5000",
               "--backend-store-uri","postgresql://.../mlflow",
               "--default-artifact-root","s3://mlflow-artifacts/"]
```

Putting It Together on OpenShift (GitOps)

1. **Git Repos**:
   - *App repo*: PINN code (`pinn_model.py`, `train.py`, Dockerfile).
   - *Ops repo*: OpenShift/K8s manifests (Airflow, MLflow, KServe, Tekton Pipelines).

2. **Build & Push**: OpenShift BuildConfigs or external CI build/push images to the internal registry.
3. **Deploy via Argo CD**: Sync manifests in *ops repo* to cluster namespaces (`dev` → `staging` → `prod`).
4. **Run Pipeline**: Tekton or Argo executes the declarative pipeline (ETL → Train → Evaluate → Deploy).
5. **Observe**: Prometheus/Grafana dashboards + MLflow metrics; alert on drift (Evidently) and trigger retraining.

Operational Notes for the PINN Use Case

- **Data versioning**: store measurement batches (HDF5/Parquet) with DVC/LakeFS; pass concrete dataset URIs into KFP runs.
- **Resource profiles**: declare GPU requests/limits per training task to ensure fair scheduling on shared clusters.
- **Artifacts**: save training curves, PDE residual maps, and mesh plots as MLflow artifacts to audit physical validity.
- **Promotion policy**: require thresholds on PDE residuals and boundary-condition violations before model promotion.
- **Rollback**: Argo CD and MLflow Registry both provide quick rollback to known-good model and manifests.

Summary (Integration at a Glance)

- **Airflow** = orchestration of the end-to-end job graph (triggers KFP/Tekton).
- **Kubeflow** = GPU training, tuning (*Katib*), and *KServe* for serving.
- **MLflow** = experiment tracking + registry for PINN model governance.
- **Declarative Pipelines** = Argo/Tekton YAML encode the workflow as code.
- **GitOps** = Argo CD keeps the cluster aligned with Git manifests.
- **OpenShift** = enterprise Kubernetes with security, routes, registry, pipelines, and GitOps—hosting all components consistently.

4 Data and Experiment Management

Goal

The goal of this module is to design a reproducible strategy for handling data, experiments, and models throughout the machine learning lifecycle. In scientific ML, this step ensures that datasets, scripts, and results can be precisely reproduced, verified, and compared across iterations. Reproducibility is not only an engineering necessity but also a scientific requirement.

Conceptual Overview

Every experiment can be expressed as a tuple:

$$E = (\mathcal{D}, \mathcal{M}, \Theta, \mathcal{R}),$$

where

- $\mathcal{D}$ — dataset or measurement configuration,
- $\mathcal{M}$ — model architecture and implementation,
- Θ — hyperparameters and physical constants,
- $\mathcal{R}$ — results (metrics, residuals, and artifacts).

Managing experiments therefore requires version control over all components of E. Tools such as DVC, LakeFS, Feast, and MLflow provide complementary functionality within this framework.

Data Versioning

Reproducible research demands that the dataset used for each training run can be reconstructed exactly — down to preprocessing and filtering steps. This is especially critical when dealing with measurement noise, temporal drift, or simulation snapshots in PDE data.

DVC (Data Version Control)

DVC extends Git semantics to large datasets. Instead of storing data directly in the repository, it creates lightweight `.dvc` metafiles containing hashes and remote storage references.

Typical workflow:

```
dvc init
dvc remote add origin s3://research-datasets
dvc add data/measurements.h5
git add data/measurements.h5.dvc .gitignore
git commit -m "Add initial measurement dataset"
dvc push
```

This guarantees that any model training job can reproduce the exact dataset version simply by:

```
dvc pull
```

Integration into the PINN pipeline:

Each Airflow/Kubeflow training step begins by running `dvc pull` to fetch the correct version of $u(x, y, t)$ measurements before training the inverse heat model. The DVC hash is logged into MLflow as a `data_version` parameter for full lineage tracking.

LakeFS

LakeFS provides Git-like version control for large-scale object storage (S3, GCS, Azure). It allows creation of isolated data branches for experiments, enabling parallel workflows:

- `main` branch for stable, validated datasets.
- `exp/#42` branch for experimental preprocessing or noise filtering.

When an experiment finishes, the branch can be merged into `main` or discarded.

Example:

The research team can branch the raw thermal measurement data into `exp/pinn-filtered` for preprocessing with spatial smoothing before training the PINN. This ensures that the original raw data remains untouched and fully recoverable.

Feature Stores

While DVC and LakeFS handle raw datasets, a **Feature Store** centralizes precomputed features for model consumption. This is crucial for production models that must ensure feature consistency between training and inference.

Feast

Feast (Feature Store) provides:

- Central registry of feature definitions (e.g., averaged heat flux, boundary gradients).
- Offline store (for training) and online store (for inference) synchronization.
- Consistent versioning and timestamps.

Example:

In the PINN inverse heat workflow, precomputed features such as

$$\phi_1(x, y) = \nabla_x u(x, y, t), \quad \phi_2(x, y) = \nabla_y u(x, y, t)$$

can be stored in Feast for reuse across experiments with different boundary conditions. The same features are available both to the training container and to real-time simulations served by KServe.

Experiment Tracking with MLflow

MLflow serves as the central hub for recording all metadata produced by training runs:

- **Parameters:** hyperparameters, regularization weights, learning rates.
- **Metrics:** losses, PDE residual norms, validation errors.
- **Artifacts:** model checkpoints, plots, configuration files.
- **Tags:** experiment identifiers, dataset versions, and domain parameters.

Example Logging Script.

```
with mlflow.start_run(run_name="inverse-heat-a=xy-var"):
    mlflow.log_param("dataset_hash", dvc_hash)
    mlflow.log_param("architecture", "PINN-5x128")
    mlflow.log_param("a_reg", 1e-3)
    mlflow.log_metric("pde_loss", loss_pde)
    mlflow.log_metric("bc_loss", loss_bc)
    mlflow.log_metric("val_residual", residual_norm)
    mlflow.log_artifact("plots/residual_map.png")
```

Model Registry.

Trained models can be registered with metadata and version numbers:

```
mlflow.register_model(
    "runs:/<run_id>/model",
    "inverse-heat-pinn"
)
```

Later stages of the MLOps pipeline (Airflow or Tekton) can promote models between stages (`Staging` → `Production`) based on performance metrics.

Structured Experimentation.

Tags allow automatic grouping:

```
mlflow.set_tags({
    "domain": "heat-inverse",
    "dataset_branch": "exp/pinn-filtered",
    "equation": "u_t - div(a grad u) = f",
    "optimizer": "Adam"
})
```

Experiments can then be queried or visualized via the MLflow UI or API, enabling comparisons across architectures and loss-weight configurations.

Model Lineage and Governance

Model lineage traces the complete genealogy of a trained model:

$$\text{Model} \to (\text{Code Version}, \text{Data Version}, \text{Parameters}, \text{Environment}).$$

Governance extends this with policies for reproducibility, approval, and compliance.

In practice:

- **Code lineage:** captured via Git commit SHA and Docker image tag.
- **Data lineage:** captured via DVC or LakeFS dataset hashes.
- **Experiment lineage:** maintained via MLflow runs and registered model versions.
- **Model approval:** controlled via MLflow Registry stages and CI/CD policies.

Governance Workflow in the PINN Project.

1. Each training run logs its dataset version (`dvc hash`), source commit, and container image.
2. Airflow automatically attaches these identifiers as MLflow tags.
3. A Tekton or ArgoCD gate promotes models that satisfy physics and validation constraints.
4. All promotions are versioned in GitOps repositories for auditability.

Example: Tracking PINN Experiments for $a(x, y)$

In the inverse heat problem

$$\partial_t u - \nabla \cdot (a(x, y)\nabla u) = f(x, y, t),$$

the goal is to infer both the thermal diffusivity $a(x, y)$ and possibly $f(x, y, t)$ from measured $u(x, y, t)$.

Experiment setup.

- **Dataset:** generated from simulated or experimental temperature fields, versioned with DVC.
- **Model:** PINN architecture ($u_\theta(x, y, t)$) defined in `pinn_model.py`.
- **Training Script:** `train.py`, parameterized by network depth, learning rate, and loss weights.
- **Tracking:** all runs logged to MLflow under the experiment `inverse-heat-pinn`.

Use Case.

Different hypotheses for $a(x, y)$ (e.g., polynomial vs. neural approximation) can be compared through their MLflow runs:

- Runs grouped by tag `architecture`.
- Metrics such as PDE residual or boundary violation used for comparison.
- Results visualized as contour plots (logged as artifacts).

Outcome.

By combining DVC (data versioning), LakeFS (branching datasets), Feast (feature management), and MLflow (experiment tracking), each PINN run is fully reproducible and scientifically auditable. This enables quantitative comparison of modeling choices, reliable peer verification, and compliant long-term archiving of simulation results.

5 Pipelines and Automation

Goal

The goal of this chapter is to design fully automated workflows that connect all components of the MLOps ecosystem — from data ingestion to model deployment. Automation ensures that every experiment, retraining, and deployment follows a repeatable, versioned process without manual intervention. In the context of scientific ML, such automation also supports reproducibility, traceability, and continuous scientific improvement.

Motivation

Machine learning pipelines are often complex, involving data validation, model training, evaluation, and deployment across heterogeneous systems. Manual orchestration is error-prone, difficult to reproduce, and does not scale. Automated pipelines encapsulate this complexity into declarative, executable workflows.
A well-designed MLOps pipeline transforms an idea into a measurable experiment:

$$\text{Data Ingestion} \rightarrow \text{Training} \rightarrow \text{Evaluation} \rightarrow \text{Deployment} \rightarrow \text{Monitoring}.$$

These steps form a continuous feedback loop for improving both model accuracy and operational robustness.

Tools and Orchestration Layers

Automation in MLOps typically involves multiple layers of orchestration:

Airflow DAGs (Directed Acyclic Graphs)

Apache Airflow is a general-purpose workflow orchestrator. It defines pipelines as DAGs where each node represents a task (e.g., data preprocessing, training, evaluation). Airflow provides:

- Python-based pipeline definition.
- Rich scheduling and dependency management.
- Integration with Kubernetes and external services.
- Fine-grained monitoring through the Airflow Web UI.

5.0.1 Conceptual Role.

Airflow is the high-level *conductor* coordinating the entire workflow. It can trigger Kubeflow Pipelines, Tekton pipelines, or even shell scripts for specialized tasks.

Kubeflow Pipelines

Kubeflow Pipelines (KFP) run machine learning tasks natively on Kubernetes/OpenShift. Each step in a pipeline is a containerized component, ensuring reproducibility and scalability.

5.0.2 Capabilities.

- Parameterized training and hyperparameter tuning.
- Distributed GPU/CPU scheduling.
- Integration with MLflow, S3, or MinIO artifact storage.
- Execution history with visualization of DAGs and metrics.

In the context of the inverse heat problem, a Kubeflow Pipeline might include:

1. Fetching and validating temperature datasets.
2. Launching the PINN training container on GPU nodes.
3. Logging metrics and residuals to MLflow.
4. Evaluating physics-based losses.
5. Registering the best-performing model.

Tekton / Argo Workflows

Tekton (OpenShift Pipelines) and **Argo Workflows** implement pipelines declaratively using YAML or JSON specifications.

5.0.3 Advantages.

- Pipelines are stored in Git — enabling versioned, auditable, and declarative automation (GitOps).
- Each task runs as a Kubernetes Pod, scaling automatically.
- Seamless integration with CI/CD systems (e.g., GitLab or Jenkins).

5.0.4 Example:

A Tekton pipeline in OpenShift could express the same PINN training workflow as a sequence of tasks:

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: inverse-heat-pinn
spec:
  tasks:
  - name: fetch-data
    taskRef: { name: dvc-fetch }
  - name: train
    runAfter: ["fetch-data"]
    taskRef: { name: pinn-train }
  - name: evaluate
    runAfter: ["train"]
    taskRef: { name: evaluate-metrics }
  - name: register
    runAfter: ["evaluate"]
    taskRef: { name: mlflow-register }
  - name: deploy
    runAfter: ["register"]
    taskRef: { name: kserve-deploy }
```

5.0.5 Execution Context.

Each task in this pipeline runs in an isolated container with its own resource request (e.g., GPU, CPU, memory). This separation ensures reproducibility and efficient utilization of compute clusters.

Automated Retraining and Evaluation

Scientific ML systems often operate on time-dependent or evolving data. Automated retraining ensures that the model remains accurate and consistent as new observations are collected.

5.0.6 Triggering mechanisms.

- **Time-based retraining:** Airflow schedules a pipeline (e.g., daily or weekly).
- **Event-based retraining:** A new dataset commit (DVC push or LakeFS merge) triggers Tekton/Argo.
- **Drift-based retraining:** Monitoring systems (Evidently AI or Prometheus alerts) detect data or model drift and trigger retraining automatically.

5.0.7 Evaluation.

Every retraining run is evaluated using both statistical and physical metrics:

$$\mathcal{L}_{\text{total}} = w_{\text{PDE}}\mathcal{L}_{\text{PDE}} + w_{\text{BC}}\mathcal{L}_{\text{BC}} + w_{\text{IC}}\mathcal{L}_{\text{IC}}.$$

Metrics are logged in MLflow and visualized in dashboards. If a model's residuals or validation errors exceed thresholds, the deployment gate prevents promotion.

Validation and Deployment Gating

Before a trained model is moved into production, automated gates ensure it satisfies quantitative and physical constraints.

5.0.8 Example Gate Policy (Airflow Task).

```
def validate_model(**context):
    run_id = context["ti"].xcom_pull(task_ids="train_pinn")
    metrics = mlflow_client.get_run(run_id).data.metrics
    if metrics["pde_loss"] < 1e-3 and metrics["val_residual"] < 1e-2:
        promote_to_registry(run_id)
    else:
        raise AirflowSkipException("Model does not meet quality thresholds.")
```

This ensures that only physically consistent PINN models (those that respect the PDE constraints) are eligible for deployment.

5.0.9 Integration with GitOps.

After validation, the pipeline commits an updated model manifest (e.g., KServe `InferenceService`) into the GitOps repository. Argo CD detects this change and deploys the new model automatically on OpenShift.

Example Airflow DAG

The following DAG captures the complete inverse heat pipeline:

```
@dag(schedule_interval="@daily", catchup=False)
def heat_inverse_pipeline():
    load_data()         # DVC pull or LakeFS branch checkout
    preprocess()        # data normalization, feature extraction (Feast)
    train_pinn()        # launch GPU training container
    evaluate_model()    # compute physics-informed residuals
    register_model()    # push to MLflow registry
    validate_model()    # gating based on thresholds
    deploy_model()      # trigger KServe or Tekton deploy

pipeline = heat_inverse_pipeline()
```

This pipeline can run either as a local Airflow DAG or as a hybrid Airflow–Kubeflow integration, where Airflow orchestrates while Kubeflow executes GPU-heavy tasks.

Automated Deployment on OpenShift

- The training container image (`pinn-trainer`) is built and stored in OpenShift's internal registry.
- Tekton or ArgoCD pipelines deploy the latest approved model to a dedicated namespace (e.g., `ml-prod`).
- KServe reads the MLflow artifact and creates a scalable inference service with GPU access.
- Monitoring dashboards (Prometheus/Grafana) collect latency and drift metrics.

5.0.10 Example KServe Manifest.

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: inverse-heat-pinn
spec:
  predictor:
    pytorch:
      storageUri: s3://mlflow-artifacts/inverse-heat/Production
      resources:
        limits: { nvidia.com/gpu: "1", cpu: "2", memory: "8Gi" }
```

Continuous Integration and Delivery (CI/CD)

CI/CD ties the automation loop together:

- Code commits trigger automated builds of Docker images (CI).
- Successful builds push manifests and configurations into Git (CD).
- Argo CD or Tekton automatically synchronizes and deploys these changes into the OpenShift cluster.

5.0.11 Benefits for the PINN Project.

- Every change in PDE configuration or loss term becomes versioned and traceable.
- Model retraining and validation are fully automated.
- Deployment gates guarantee only physically consistent models reach production.

Summary

Automated pipelines form the *operational nervous system* of an MLOps platform. For the inverse heat PINN project:

- Airflow orchestrates the end-to-end experiment lifecycle.
- Kubeflow Pipelines execute containerized training and tuning.
- Tekton and Argo ensure declarative, auditable automation.
- Validation gates enforce physics-based constraints before promotion.
- OpenShift hosts the entire automation stack with integrated CI/CD, GPU scheduling, and monitoring.

This combination provides a reproducible, automated, and scientifically verifiable infrastructure for continuous PINN experimentation and deployment.

6 Deployment and Serving

Goal

The goal of this chapter is to deploy trained machine learning models in a reproducible, maintainable, and scalable manner. Deployment is the bridge between research and application: a model only becomes useful when it can be reliably consumed by other systems, users, or experiments. In scientific ML, the deployment stage must preserve reproducibility, numerical integrity, and version traceability.

Overview

Modern MLOps platforms treat model serving as an operational service that exposes models through standardized APIs. Every deployed model must satisfy:

- **Reproducibility:** the deployed model must match a specific MLflow model version, container hash, and data version.
- **Scalability:** the service must handle concurrent prediction requests or batch jobs.
- **Observability:** each prediction and its inputs are logged for auditing and performance analysis.

Model serving can take different forms — from lightweight REST APIs for experimentation to production-grade serving frameworks integrated with Kubernetes.

Techniques

REST APIs with FastAPI or Flask

For small-scale deployments or prototyping, a REST API is sufficient to expose model inference endpoints. **FastAPI** is particularly well-suited for ML because it is asynchronous, type-checked, and integrates easily with Docker and Kubernetes.

6.0.1 Example FastAPI Service.

```
from fastapi import FastAPI
import torch
from model import PINNModel

app = FastAPI(title="Inverse Heat PINN")

model = torch.jit.load("pinn_model.pt")
model.eval()

@app.get("/predict-temperature")
def predict_temperature(x: float, y: float, t: float):
    with torch.no_grad():
        inp = torch.tensor([[x, y, t]], dtype=torch.float32)
        u = model(inp).item()
        return {"u(x,y,t)": u}
```

6.0.2 Dockerfile for Serving.

```
FROM python:3.11
WORKDIR /app
COPY . .
RUN pip install fastapi uvicorn torch
EXPOSE 8080
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8080"]
```

The resulting container can be built and run as:

```
docker build -t pinn-api .
docker run -p 8080:8080 pinn-api
```

6.0.3 Usage.

A client can query the deployed model via:

```
curl "http://localhost:8080/predict-temperature?x=0.5&y=0.5&t=0.1"
```

This returns the predicted temperature $u(x, y, t)$.

Model Serving on Kubernetes: KServe and Seldon Core

When scaling to production or multi-user environments, manual API services become insufficient. **KServe** and **Seldon Core** provide standardized Kubernetes-native serving for ML models.

KServe (KFServing)

KServe allows deploying models directly from storage backends such as S3 or MLflow Registry without rebuilding containers.

6.0.4 Example KServe Deployment.

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: inverse-heat-pinn
spec:
  predictor:
    pytorch:
      storageUri: s3://mlflow-artifacts/inverse-heat-pinn/Production
      resources:
        limits:
          nvidia.com/gpu: "1"
          cpu: "2"
          memory: "8Gi"
```

6.0.5 Workflow.

1. Airflow or Tekton promotes the model in MLflow Registry to `Production`.
2. ArgoCD detects a manifest change in the GitOps repository.
3. OpenShift deploys the new `InferenceService`.
4. KServe spins up a dedicated inference Pod with the model mounted from object storage.

6.0.6 Endpoint.

Once deployed, KServe automatically exposes an HTTPS route such as:

```
https://inverse-heat-pinn.apps.openshift.cluster/predict
```

A POST request with JSON body:

```
{
  "instances": [[0.5, 0.5, 0.1]]
}
```

returns:

```
{"predictions": [0.8243]}
```

Seldon Core

Seldon Core is an alternative to KServe with richer model routing, A/B testing, and custom graph definitions.

6.0.7 Example Seldon Deployment.

```
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: pinn-deployment
spec:
  predictors:
  - name: default
    replicas: 2
    graph:
      name: model
      implementation: PYTORCH_SERVER
      modelUri: s3://mlflow-artifacts/inverse-heat-pinn/Production
    componentSpecs:
    - spec:
        containers:
        - name: model
          resources:
            limits:
              nvidia.com/gpu: "1"
```

Seldon integrates naturally with OpenShift's service mesh (Istio) for routing and traffic control between different model versions.

Continuous Deployment (CI/CD)

To maintain reliability, model deployment must be integrated into the CI/CD process.

6.0.8 Pipeline Example (GitLab CI/CD).

```
stages:
  - build
  - deploy

build:
  stage: build
  script:
    - docker build -t registry/pinn-api:$CI_COMMIT_SHA .
    - docker push registry/pinn-api:$CI_COMMIT_SHA

deploy:
  stage: deploy
  script:
    - kubectl apply -f manifests/kserve-inverse-heat.yaml
  only:
    - main
```

6.0.9 Alternative Tools.

- **Jenkins:** suitable for on-prem clusters and legacy CI/CD pipelines.
- **Argo CD:** declarative GitOps tool continuously reconciling the live cluster state with Git manifests.

6.0.10 Integration with OpenShift.

On OpenShift, Tekton and ArgoCD provide native CI/CD and GitOps. Models promoted to the MLflow Registry automatically trigger new pipeline runs that update the deployed inference service.

Deployment Strategies

Blue-Green Deployment

Two identical environments (`blue` and `green`) exist concurrently:

- The `blue` environment serves current production traffic.
- The `green` environment deploys the new model.

Once validation passes, traffic is switched from blue to green. Rollback is instantaneous by redirecting traffic back to the old version.

6.0.11 Implementation in OpenShift.

Using OpenShift Routes:

```
oc set route-backends inverse-heat-pinn blue=0 green=100
```

can shift traffic between model versions atomically.

Canary Deployment

Instead of a full switch, traffic is gradually shifted to the new model version to monitor stability:

```
oc set route-backends inverse-heat-pinn blue=90 green=10
```

Monitoring tools (Prometheus, Grafana, Evidently) track error rates, latency, and drift metrics. If no degradation is detected, traffic is progressively increased until the new model fully replaces the old one.

Operational Integration for the PINN Project

6.0.12 Scenario.

After training and validation, a new PINN model for estimating $a(x, y)$ and $f(x, y, t)$ is promoted in MLflow and deployed automatically.

6.0.13 Pipeline Steps.

1. **Airflow or Tekton** detects new MLflow model promotion.
2. **ArgoCD** synchronizes the updated KServe manifest in Git.
3. **OpenShift** deploys the new inference service with GPU scheduling.
4. **Prometheus/Grafana** monitor inference latency and physical consistency.
5. **Evidently AI** analyzes prediction drift; significant drift triggers retraining.

6.0.14 API Specification for PINN Inference.

```
/predict-temperature(x, y, t)
  Method: GET
  Input:  float x, y, t
  Output: float u(x,y,t)
```

Example query:

```
curl "https://inverse-heat-pinn.apps.cluster/predict-temperature?x=0.5&y=0.5&t=0.1"
```

6.0.15 Response Example.

```
{
  "u(x,y,t)": 0.8243,
  "model_version": "1.3.2",
  "data_version": "dvc-8fa23b",
  "timestamp": "2025-10-20T15:22:17Z"
}
```

The response includes full lineage metadata, enabling complete traceability and reproducibility of scientific predictions.

Summary

Model deployment is the culmination of the MLOps lifecycle — transforming reproducible research artifacts into operational services. For the inverse heat PINN project:

- FastAPI enables quick experimentation and REST access.
- KServe and Seldon Core deliver scalable inference on Kubernetes/OpenShift.
- CI/CD systems (GitLab, Jenkins, ArgoCD) automate builds and rollouts.
- Blue-Green and Canary strategies minimize downtime and risk.
- Monitoring (Prometheus, Grafana, Evidently) ensures reliability and retraining triggers.

Through these mechanisms, scientific models evolve into robust, production-grade services that remain explainable, auditable, and continuously improvable.

7 Monitoring, Drift, and Retraining

Goal

Implement continuous monitoring and feedback loops for deployed models.

What to monitor (start simple)

1. **Service health:** latency, throughput, error rate.
2. **Prediction quality:** task metrics (e.g. MSE of temperature), confidence/uncertainty.
3. **Data properties:** input ranges and distribution summaries.
4. **Domain/PDE signals (PINN):** average PDE residual, boundary condition violations, physical constraint counters.

Types of drift

- **Data Drift** (covariate shift): $p_{\text{serving}}(x)$ deviates from $p_{\text{train}}(x)$.
- **Concept Drift**: $p(y \mid x)$ changes (same inputs, different temperature outcomes).
- **Model Drift**: model parameters or calibration degrade over time even if data seems similar.

Minimal signals and thresholds (practical defaults)

- **Latency (p95)** $< 200\,\text{ms}$; **Error rate** $< 0.1\%$.
- **Input sanity**: percent of (x, y, t) outside training bounds $< 1\%$.
- **Data drift**: PSI (Population Stability Index) for each feature < 0.2.
- **PINN physics**: mean PDE residual $\overline{R_{\text{PDE}}} < \tau$.

Monitoring tools

- **Prometheus:** scrapes numeric metrics.
- **Grafana:** dashboards/alerts on Prometheus.
- **Evidently AI:** computes drift/statistics and generates reports.
- **WhyLabs:** managed monitoring/logging for data & ML.

7.1 Model Drift

Deployed machine learning models operate in inherently non-stationary environments. Formally, most learning algorithms are based on the assumption that the training and deployment distributions coincide:

$$P_{\text{train}}(X, Y) \approx P_{\text{future}}(X, Y).$$

In practice, however, real-world systems evolve over time due to environmental, technological, or physical changes. This leads to a violation of the stationarity assumption and results in gradual or abrupt model performance degradation.

Let $\mathcal{D}_t$ denote the data distribution at time t. A learning system is trained at time t_0 using samples drawn from $\mathcal{D}_{t_0}$. Over time, the operational distribution evolves as:

$$\mathcal{D}_{t_0} \to \mathcal{D}_{t_1} \to \mathcal{D}_{t_2} \to \cdots$$

with potentially increasing discrepancy between $\mathcal{D}_{t_0}$ and $\mathcal{D}_{t_k}$.
This temporal evolution induces what is known as *model drift*, i.e., a systematic deviation between the model's learned representation and the true underlying data-generating process.
In physics-driven problems such as inverse heat conduction with Physics-Informed Neural Networks (PINNs), this drift additionally affects:

- the learned physical parameters $a(x, y)$,
- the inferred source term $f(x, y, t)$,
- and the model's PDE consistency via the residual $\mathcal{R}(x, y, t)$.

—

Types of Drift

Drift can be formally categorized according to which part of the joint distribution changes over time.
Let:

$$P_t(X, Y) = P_t(Y|X)P_t(X).$$

1. Data Drift (Covariate Shift)

Here, only the marginal input distribution changes:

$$P_{\text{train}}(X) \neq P_{\text{live}}(X), \quad P(Y|X) \approx \text{const}.$$

This situation arises frequently when sensor characteristics, data acquisition environments, or experimental setups change. In your PINN application, this might correspond to changes in spatial sampling density or a shift in measurement device precision without altering the underlying heat physics.
The severity of this change can be quantified using statistical divergence measures such as:

$$D_{\text{JS}}(P_{\text{train}}(X)\|P_{\text{live}}(X)), \quad W_1(P_{\text{train}}(X), P_{\text{live}}(X)),$$

where D_{JS} is the Jensen–Shannon divergence and W_1 is the Wasserstein distance.
—

2. Concept Drift

In this case, the conditional distribution changes:

$$P_{\text{train}}(Y|X) \neq P_{\text{live}}(Y|X).$$

This is particularly relevant in physical systems where materials, boundary conditions, or external forcing terms evolve.
For your PINN problem, this corresponds to changes in the physical mapping governed by:

$$u_t - \nabla \cdot (a(x, y)\nabla u) = f(x, y, t),$$

where the coefficients $a(x, y)$ and $f(x, y, t)$ themselves may drift over time.
Concept drift is more dangerous because it affects the true physical law representation, not just its observations.
—

3. Model Drift

Even under stationary $P(X, Y)$, a model may degrade due to:

- limited capacity,
- overfitting to outdated patterns,
- poor calibration to new regimes.

In PINNs, model drift manifests as:

$$\|\mathcal{R}_{\text{PDE}}(u_\theta)\|_{L^2(\Omega)} \uparrow, \quad \|\hat{a}(x, y) - a^*(x, y)\| \uparrow,$$

where $\mathcal{R}_{\text{PDE}}$ is the PDE residual functional.
—

Monitoring and Drift Detection

Let $\{X_t\}_{t \geq 0}$ denote incoming data streams. Drift detection can be formulated as a hypothesis testing problem:

$$H_0 : P_{\text{train}}(X) = P_{\text{live}}(X)$$
$$H_1 : P_{\text{train}}(X) \neq P_{\text{live}}(X)$$

Statistical tools for detecting drift include:

- Kolmogorov–Smirnov Tests,
- Population Stability Index (PSI),
- Maximum Mean Discrepancy (MMD),
- Wasserstein Distance.

In an MLOps context, monitoring operates on three layers:

1. **Input Level:** Distribution of features, spatial coordinates, and time signals.
2. **Output Level:** Distribution of predicted temperature fields or reconstructed $u(x, y, t)$.
3. **Physical Level:** Evolution of PDE residual norms:

$$\mathcal{E}_{\text{PDE}} = \int_\Omega |\mathcal{R}(x, y, t)|^2 d\Omega,$$

   serving as physics-consistency drift indicators.

—

7.1.1 Automated Retraining Loop

Let $\mathcal{M}_k$ denote the deployed model at cycle k. A retraining strategy is triggered when a monitoring functional exceeds a threshold:

$$\Phi(\mathcal{M}_k) > \epsilon.$$

This functional may be defined as:

$$\Phi(\mathcal{M}) = \alpha D_{\text{data}} + \beta D_{\text{concept}} + \gamma \mathcal{E}_{\text{PDE}},$$

where α, β, γ are weighting coefficients.
Triggered retraining pipelines perform:

1. Data re-collection: $\mathcal{D}_{k+1}$,

2. Versioning via DVC/LakeFS,
3. Model retraining: $\mathcal{M}_{k+1}$,
4. Evaluation via:
$$\text{RMSE}, \quad \mathcal{E}_{\text{PDE}}, \quad \|\hat{a} - \hat{a}^*\|,$$
5. Promotion using a controlled deployment scheme (Canary, Blue–Green).

The updated model $\mathcal{M}_{k+1}$ replaces $\mathcal{M}_k$ only if:

$$\mathcal{J}(\mathcal{M}_{k+1}) < \mathcal{J}(\mathcal{M}_k),$$

where $\mathcal{J}$ is a model quality functional combining prediction accuracy and physical consistency.
—

Drift in the Context of Inverse Heat Problems

In inverse heat conduction using PINNs, drift manifests as:

- Evolution of estimated diffusivity fields $a(x, y, t)$,
- Changes in source term reconstruction $f(x, y, t)$,
- Increase in physics residual norms,
- Degradation in predicted temperature field fidelity.

This allows the development of *physics-aware drift metrics* beyond standard data-based metrics.
A possible drift monitor functional is:

$$D_{\text{phys}} = \int_\Omega |a_k(x, y) - a_{k-1}(x, y)|^2 \, dxdy + \int_\Omega |\mathcal{R}_k(x, y, t)|^2 dxdy,$$

enabling detection rooted in physical model deviations.
—

Conclusion

Model drift is not merely a statistical phenomenon but a dynamic system-level challenge involving evolving data distributions, changing physical laws, and model degradation. Modern MLOps frameworks address this by unifying:

- statistical drift detection,
- physics-informed monitoring,
- automated retraining pipelines,
- controlled deployment strategies.

This integration transforms machine learning systems from static estimators into self-correcting dynamical systems capable of long-term deployment in non-stationary environments.

7.1.2 Drift–Retraining Architecture

Figure 7.1 illustrates the automated drift detection and retraining loop implemented within the MLOps workflow.

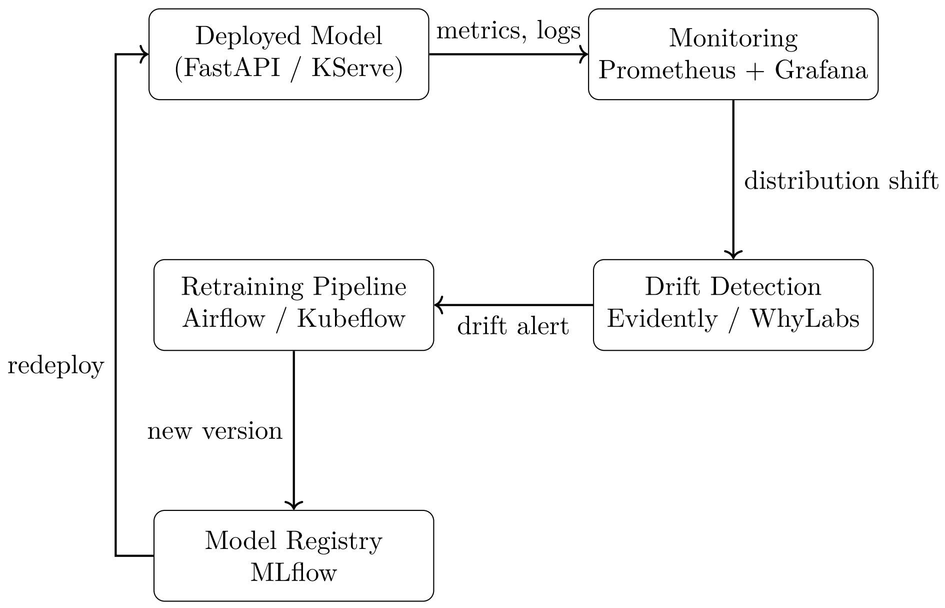


Figure 7.1: Automated model drift detection and retraining loop in an MLOps system.

Step 1 — Instrument the PINN service (Prometheus)

Listing 7.1: FastAPI metrics for the PINN service

```python
from fastapi import FastAPI
from prometheus_client import Counter, Gauge, Histogram, generate_latest
from prometheus_client import CONTENT_TYPE_LATEST
from starlette.responses import Response
import time

app = FastAPI()

# Service-level metrics
REQS = Counter("pinn_requests_total", "Total inference requests")
ERRS = Counter("pinn_errors_total", "Total inference errors")
LAT  = Histogram("pinn_latency_seconds", "Latency per request (s)")

# Data + physics metrics
OUT_OF_RANGE = Gauge("pinn_inputs_oob_ratio", "Ratio of inputs outside training bounds
    ")
PDE_RES_MEAN = Gauge("pinn_pde_residual_mean", "Mean PDE residual")
BC_VIOL = Gauge("pinn_bc_violation_ratio", "Boundary condition violation rate")

@app.get("/metrics")
def metrics():
    return Response(generate_latest(), media_type=CONTENT_TYPE_LATEST)

@app.post("/predict-temperature")
def predict(item: dict):
    start = time.time(); REQS.inc()
    try:
        # ... compute u(x,y,t) and PDE residual ...
        OUT_OF_RANGE.set(item["oob_ratio"])
        PDE_RES_MEAN.set(item["r_pde_mean"])
        BC_VIOL.set(item["bc_violation_ratio"])
        return {"u": 42.0}
    except Exception:
        ERRS.inc()
        raise
    finally:
        LAT.observe(time.time() - start)
```

Kubernetes annotation to enable scraping:

```
metadata:
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: /metrics
    prometheus.io/port: "8080"
```

Step 2 — Grafana: useful panels

- Latency p95:
  `histogram_quantile(0.95, sum(rate(pinn_latency_seconds_bucket[5m])) by (le))`
- Error rate:
  `rate(pinn_errors_total[5m]) / rate(pinn_requests_total[5m])`
- PDE residual trend:
  `pinn_pde_residual_mean`

Example alert rules:

```
pinn_inputs_oob_ratio > 0.05 for 15m   -> Data shift
pinn_pde_residual_mean > tau for 15m   -> Physics degradation
error_rate > 0.005 for 10m             -> Serving issue
```

Step 3 — Batch drift reports (Evidently)

Listing 7.2: Evidently data drift detection job

```python
import pandas as pd
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset

ref = pd.read_parquet("s3://ml/heat/reference.parquet")
cur = pd.read_parquet("s3://ml/heat/current.parquet")

report = Report(metrics=[DataDriftPreset()])
report.run(reference_data=ref, current_data=cur)
report.save_html("drift_report.html")

json = report.as_dict()
psi_ok = all(m["result"]["dataset_drift"] is False for m in json["metrics"]
             if "dataset_drift" in str(m["result"]))
if not psi_ok:
    open("/tmp/DRIFT_FLAG", "w").write("data_drift\n")
```

What to feed Evidently (PINN):

- Features: (x, y, t).

- Targets: measured u^* to compute residual metrics.
- Physics metrics: R_{PDE}, boundary condition violation rate.

Step 4 — Automatic retraining hooks

Listing 7.3: Airflow DAG for drift → retrain → deploy

```
@dag(schedule_interval="*/30 * * * *", catchup=False)
def monitor_and_retrain():
    drift = BashOperator(
        task_id="compute_drift",
        bash_command="python drift_report.py && test ! -f /tmp/DRIFT_FLAG"
    )
    retrain = BashOperator(
        task_id="train_pinn",
        bash_command="python train_pinn.py --exp mlops --out /mlruns"
    )
    eval_gate = BashOperator(
        task_id="eval_gate",
        bash_command="python eval_gate.py --min_pde <TAU> --max_mse <EPS>"
    )
    register = BashOperator(
        task_id="register_model",
        bash_command="python mlflow_register.py --stage Staging"
    )
    deploy = BashOperator(
        task_id="deploy",
        bash_command="python deploy_kserve.py --from-registry"
    )
    retrain.trigger_rule = "one_failed"
    drift >> [retrain, eval_gate]
    retrain >> eval_gate >> register >> deploy
```

Evaluation gates (PINN-aware)

- **Offline:** MSE, physics residual, BC violation rate.
- **Shadow:** route small traffic slice to candidate.
- **Promotion:** only if improvement > threshold.

Step 5 — Alerts to orchestration

- Prometheus Alertmanager → webhook triggers Airflow or Argo run.
- Grafana alerts → HTTP webhook for retraining DAG.
- On OpenShift: use a CronJob for drift check, store `drift_report.html`, expose via Route.

Step 6 — SLOs and Runbooks

- SLOs: latency $< 200\,\text{ms}$, error $< 0.1\%$, OOB $< 1\%$, $\overline{R_{\text{PDE}}} < \tau$.
- Alerts: thresholds, durations, owners.
- Runbook: rollback (ArgoCD, KServe), validation checklist.

Example (PINN inverse heat)

- Online: fast signals (PDE residual, input domain).
- Batch: drift reports on (x, y, t), u^*, and physics terms.
- MLflow: compare runs by tags (network depth, PDE weight).

Final loop summary

1. Instrument metrics (Prometheus).
2. Dashboard & alert (Grafana).
3. Run nightly drift check (Evidently).
4. Airflow DAG: drift $\rightarrow$ retrain $\rightarrow$ eval $\rightarrow$ register $\rightarrow$ deploy.
5. Promotion gate: no physics regression.
6. Full audit via MLflow & GitOps.

References

Literature

[1] Gesler Radmir. "Physics Informed Neural Networks (PINN) zur Lösung eines inversen Problems der Wärmeleitungsgleichung". In: (2025). URL: https://github.com/RadmirG/MA_Latex (cit. on p. 7).

List of Figures

List of Tables