Cadi Ayyad University
Faculty of Sciences Semlalia

# Orchestrating Continuous NAS with Early Stopping

A Comprehensive Tutorial on Kubeflow Katib, PVC Storage, and FastAPI

**Prepared by:** ELAKIL Hakima

Academic Year: 2025–2026

# Contents

# 1 Project Context and Objectives

This laboratory explores the implementation of an automated Machine Learning pipeline. The transition from static model development to a dynamic MLOps lifecycle is achieved by integrating automated search, resource management, and containerized serving.

## 1.1 Core Objectives

- **Automated Optimization:** Using Katib to explore the search space of a Multi-Layer Perceptron (MLP) without manual intervention.
- **Resource Efficiency:** Minimizing compute waste via **Early Stopping** based on real-time validation metrics.
- **Data Persistence:** Implementing **Persistent Volume Claims (PVC)** to ensure model artifacts survive the ephemeral nature of Kubernetes pods.
- **Inference Readiness:** Deploying the optimized model through a **FastAPI** REST interface.

# 2 Theoretical Framework

## 2.1 Used Technologies and Their Roles

- **MicroK8s:** The lightweight Kubernetes orchestration engine that manages the cluster, networking, and storage.
- **Docker:** Containerization technology ensuring the training environment (Python, Scikit-Learn) is reproducible and portable.
- **Kubeflow Katib:** The AutoML component responsible for managing Experiments, generating Trials, and suggesting Hyperparameters.
- **Jobs and Trials:** Each training run is encapsulated in a Kubernetes Job launched by Katib as a Trial. Trials can run in parallel, and Katib automatically tracks metrics (e.g., accuracy) to determine the best performing model.
- **Persistent Volume Claim (PVC):** A storage abstraction that acts as a "Virtual Hard Drive," ensuring data persists even after training pods are deleted.
- **FastAPI:** A modern, high-performance web framework used to serve the trained model as a REST API.

## 2.2 The Essential Role of Experiments, Suggestions, and Trials

In the context of Neural Architecture Search (NAS), the concepts of **Experiments**, **Suggestions**, and **Trials** are fundamental:

- **The Experiment:** This is the high-level control unit. It defines the *Objective* (e.g., maximize accuracy), the *Search Space* (e.g., number of layers, learning rate), and the *Search Algorithm* (e.g., Random Search, Bayesian Optimization).
- **The Suggestion:** This represents the decision-making layer. Based on the selected search algorithm, Katib generates suggestions that propose new hyperparameter configurations to explore, using feedback from previous trials to guide the search process.
- **The Trial:** This is the actual execution unit. Each Trial corresponds to one specific configuration of hyperparameters proposed by a Suggestion. Trials run in parallel as Kubernetes pods, enabling efficient exploration of multiple architectures simultaneously.

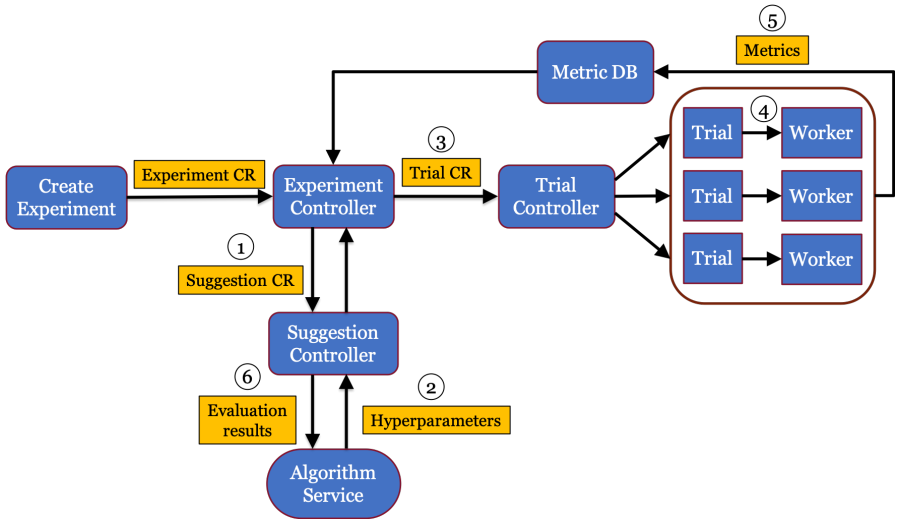


Figure 1: Kubeflow Katib: System Architecture

Katib follows the iterative workflow illustrated in the diagram. Once an **Experiment** is submitted, the Experiment Controller creates a **Suggestion** object. The Suggestion Controller then launches the AutoML algorithm service, which generates new hyperparameter configurations. Based on these suggestions, the Experiment Controller creates corresponding **Trial** objects. Each Trial is executed as a Kubernetes pod using the Trial template, while its execution is monitored by the Trial Controller. After completion, the Metrics Collector retrieves and stores the reported metrics. These results are then fed back to the algorithm service, enabling the generation of improved suggestions and forming a closed optimization loop.

### 2.3 Methodological Framework: The "Student" Model

To define our training strategy, we adopt a "Student" analogy within a supervised learning curriculum:

- **Continuous Training:** Unlike standard models that restart from scratch, our model utilizes `warm_start`, allowing the "Student" to retain weights across incremental iterations.
- **Early Stopping (Dynamic Termination):** This acts as a graduation criteria. If the student's performance (Accuracy) plateaus for 5 consecutive rounds (Epochs), we terminate the session to prevent overfitting and save resources.

## 3 Project Structure

The project follows a modular structure where each file represents a distinct stage in the MLOps lifecycle.

### 3.1 File Organization

```
katib-nas-atelier/
|-- train_continuous_nas.py  # Core training logic with Early Stopping
|-- Dockerfile               # Environment blueprint for the training pod
|-- continuous_nas.yaml      # Katib Experiment definition (Search Space)
|-- tmp-pvc-pod.yaml         # Helper pod to extract models from PVC
|-- main.py                  # FastAPI inference service
|-- model/                   # Local mount point for PVC artifacts
```

### 3.2 Workflow

1. **Environment Packaging:** The training script is containerized using Docker.
2. **NAS Orchestration:** Katib interprets the YAML blueprint to launch multiple Trials.
3. **Metric Tracking:** A sidecar container captures accuracy logs from the pods.
4. **Artifact Serialization:** The best model is saved to the Persistent Volume.
5. **Model Serving:** FastAPI mounts the volume and exposes the model to the network.

### 3.3 Config of the hyperparams

### 3.4 Tableau des Hyperparamètres et Intervalles pour Iris et MNIST

## 4 Phase 1: Prerequisites & Installation

Before running the experiment, the infrastructure must be correctly provisioned.

| Dataset | Hyperparamètre | Type | Intervalle / Valeurs | Description |
|---|---|---|---|---|
| Iris | layers | int | 1 – 3 | Nombre de couches cachées du MLP |
| | units | int | 8 – 32 | Nombre de neurones par couche |
| | lr | double | 0.001 – 0.05 | Taux d'apprentissage initial |
| MNIST | layers | int | 2 – 4 | Nombre de couches cachées du MLP |
| | units | int | 64 – 256 | Nombre de neurones par couche |
| | lr | double | 0.0005 – 0.01 | Taux d'apprentissage initial |

Table 1: Configuration des hyperparamètres pour les expériences Iris et MNIST

### 4.1 MicroK8s Setup

```
sudo snap install microk8s --classic
microk8s status --wait-ready
microk8s enable dns storage
```

**Role:** *DNS* is critical for service discovery; *Storage* enables the Host-Path provisioner to create PVCs on our local disk.

### 4.2 Katib Installation (Standalone)

We install Katib by cloning the official repository and applying the standalone manifests.

```
# 1. Clone the repository
git clone https://github.com/kubeflow/katib.git

# 2. Navigate to the standalone installation directory
cd katib/manifests/v1beta1/installs/katib-standalone

# 3. Apply the manifests
microk8s kubectl apply -k .

# 4. Verify Katib Pods are Running
microk8s kubectl get pods -n kubeflow
```

### 4.3 Namespace Configuration

```
# Create Namespace
microk8s kubectl create namespace kubeflow

# Enable Metrics Injection
microk8s kubectl label namespace kubeflow katib.kubeflow.org/metrics-
    collector-injection=enabled
```

**Role:** This labels the namespace, signaling Katib to inject a "Metrics Collector" container into every trial pod to read the "accuracy=" string.

### 4.4 Essential Verification

After installation, verify that the core Katib components are running:

```
microk8s kubectl get pods -n kubeflow
```

You should see:

- `katib-controller`: Ready 1/1 & status Running
- `katib-db-manager`: Ready 1/1 & status Running
- `katib-mysql`: Ready 1/1 & status Running
- `katib-ui`: Ready 1/1 & status Running

```
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/mnist$ microk8s kubectl get pods -n kubeflow
NAME                                READY   STATUS      RESTARTS   AGE
katib-controller-85d6f4497d-mvnw9   1/1     Running     2          4d7h
katib-db-manager-6dbf658d74-qhz6l   1/1     Running     3          4d7h
katib-mysql-6d756b5657-vvl6v        1/1     Running     2          4d7h
katib-ui-5954947794-vmvhj           1/1     Running     2          4d7h
```

This ensures the environment is fully operational before launching any experiments.

# 5 Phase 2: Training & Orchestration

## 5.1 Defining the Logic and Environment

First, we define the training script that implements the Early Stopping logic.

```python
# ... imports ...
model = MLPClassifier(hidden_layer_sizes=tuple([args.units]*args.layers),
                      warm_start=True, max_iter=1)

# Continuous Training Loop
for epoch in range(50):
    model.fit(X_train, y_train)
    acc = accuracy_score(y_test, model.predict(X_test))
    print(f"accuracy={acc}") # Metric for Katib

    # Early Stopping Check
    if acc > best_acc:
        best_acc, wait = acc, 0
    else:
        wait += 1
    if wait >= patience: break

# Persist to PVC
with open(f"{args.export_path}/model.pkl", "wb") as f:
    pickle.dump(model, f)
```

Listing 1: train_continuous_nas.py

## 5.2 Docker Environment Definition

We define a lean container image including `scipy` and our training logic.

```
FROM python:3.10-slim

WORKDIR /app
COPY train_continuous_nas.py /app/train_continuous_nas.py

RUN pip install --no-cache-dir numpy scipy scikit-learn

# Corrected Entrypoint to match the copied file
ENTRYPOINT ["python3", "/app/train_continuous_nas.py"]
```

Listing 2: Dockerfile

## 5.3 Execution

We build the image, import it into the cluster registry, and submit the experiment manifest.

```
# 1. Build the Docker image
docker build -t katib-iris:v3 .

# 2. Import image to MicroK8s registry
docker save katib-iris:v3 | microk8s ctr image import -

# 3. Submit the Experiment to the Cluster
microk8s kubectl apply -f continuous_nas.yaml
```

**Role:** *Build* creates a portable environment; *Import* pushes that image into the MicroK8s internal registry; *Apply* triggers the Katib controller to start the "Bi-level optimization" process.

```
 ---> Removed intermediate container f4938f855724
 ---> 412a0d387866
Step 5/5 : ENTRYPOINT ["python3", "/app/train.py"]
 ---> Running in f1cf6f8527a4
 ---> Removed intermediate container f1cf6f8527a4
 ---> d9e0c4ff6d5e
Successfully built d9e0c4ff6d5e
Successfully tagged katib-iris:v3
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ docker save katib-iris:v3 | microk8s ctr image import -
[sudo] password for hp:
unpacking docker.io/library/katib-iris:v3 (sha256:b5c878ce89d7ab56d311115c2ea6c48cc7080c07b25f1f800d504231d126119d)...done
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl apply -f continuous_nas.yaml
experiment.kubeflow.org/iris-continuous-nas created
```

Figure 2: image building and submission of the expirement

## 5.4 Verification

```
# Check Pods
microk8s kubectl get pods -n kubeflow
# Check Trials
```

```
microk8s kubectl get trials -n kubeflow
# Access UI
microk8s kubectl port-forward -n kubeflow svc/katib-ui 8080:80
```

```
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl get pods -n kubeflow
NAME                                      READY   STATUS    RESTARTS   AGE
iris-continuous-nas-7988zrq2-hd5br        2/2     Running   0          8s
iris-continuous-nas-dw7gw4sl-f492d        2/2     Running   0          8s
iris-continuous-nas-random-66578c6b66-24bcz   1/1     Running   0          26s
katib-controller-85d6f4497d-mvnw9         1/1     Running   0          160m
katib-db-manager-6dbf658d74-qhz6l         1/1     Running   0          160m
katib-mysql-6d756b5657-vvl6v              1/1     Running   0          160m
katib-ui-5954947794-vmvhj                 1/1     Running   0          160m
```

Figure 3: Pods status verification

```
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl get trials -n kubeflow
NAME                          TYPE        STATUS   AGE
iris-continuous-nas-7988zrq2  Succeeded   True     23s
iris-continuous-nas-dw7gw4sl  Succeeded   True     23s
```

Figure 4: Trials List and Status

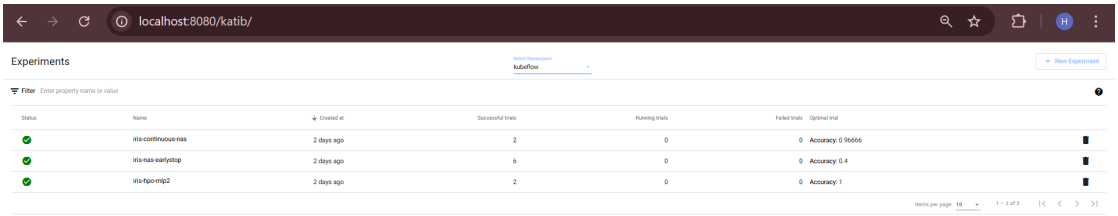


Figure 5: Katib UI

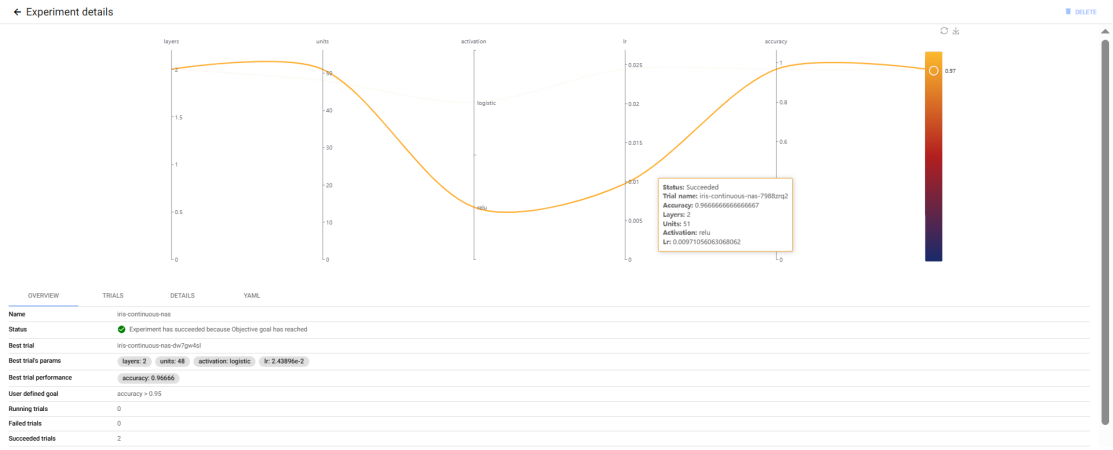


Figure 6: Expiremnet details


Figure 7: The complete set of trials, where the best performing iteration is marked in yellow

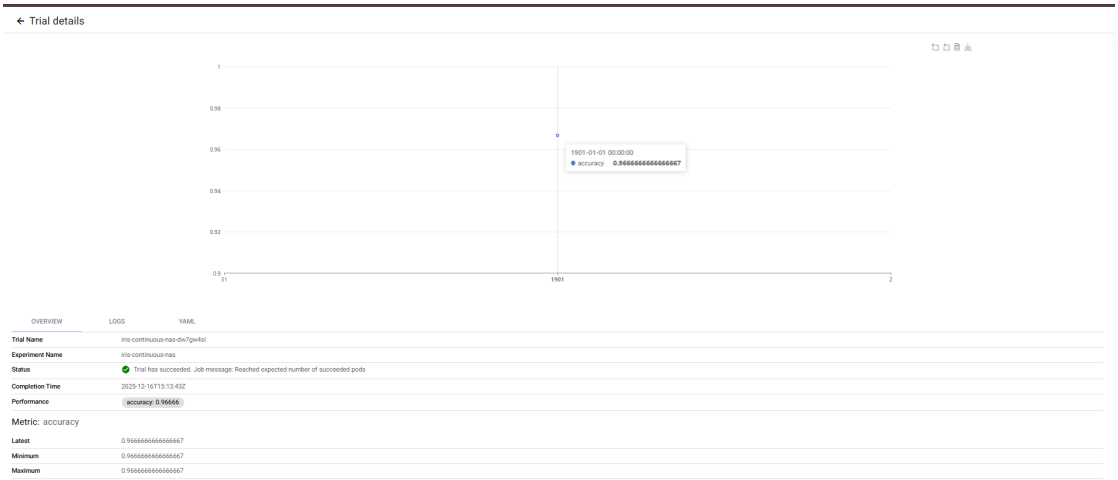


Figure 8: Best Trial details

# 6 Phase 3: Persistence and Serving

```
microk8s kubectl get pvc -n kubeflow
```

**Role:** Confirms that our "Virtual Disk" is **Bound**. Without this, the training script has no place to save the `.pkl` file.

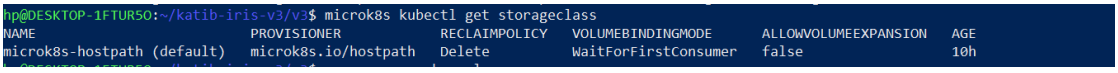


Figure 9: Verification of Persistent Volume Claim status.

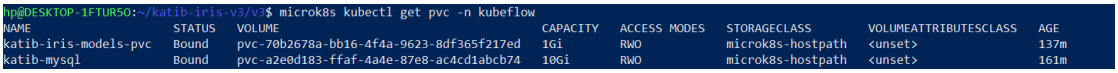


Figure 10: PVC verification in "Bound" state

## 6.1 Model Recuperation: The "Bridge" Pod Strategy

Since Katib trial pods are ephemeral (they terminate upon completion), direct access to the generated artifacts is lost once the trial finishes. To recuperate the trained model from the **Persistent Volume Claim (PVC)**, we must employ a temporary "Bridge" Pod.

### 6.1.1 Defining the Inspection Pod

We define a lightweight Pod configuration (`tmp-pvc-pod.yaml`) whose sole purpose is to mount the shared storage volume and sleep, allowing us time to access the file system.

```
apiVersion: v1
kind: Pod
metadata:
  name: tmp-pvc-pod
  namespace: kubeflow
spec:
  containers:
    - name: pvc-container
      image: python:3.10-slim
      # Sleep keeps the pod alive so we can enter it
      command: ["sleep", "3600"]
      volumeMounts:
        - name: model-volume
          mountPath: /model
  volumes:
    - name: model-volume
      persistentVolumeClaim:
```

```
        # Must match the PVC name used in the Experiment
        claimName: katib-iris-models-pvc
```

Listing 3: tmp-pvc-pod.yaml

### 6.1.2 Execution and Extraction

Once the manifest is defined, we deploy the pod and use `kubectl cp` to transfer the artifact from the Kubernetes cluster to the local development environment.

```
# 1. Deploy the bridge pod
microk8s kubectl apply -f tmp-pvc-pod.yaml

# 2. Wait for the Pod to be 'Running', then verify artifact existence
microk8s kubectl exec -n kubeflow -it tmp-pvc-pod -- ls -l /model/

# 3. Recuperate the model (Copy from Cluster to Local Host)
microk8s kubectl cp kubeflow/tmp-pvc-pod:/model/model.pkl ./model.pkl
```

```
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl apply -f tmp-pvc-pod.yaml
pod/tmp-pvc-pod created
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl get pods -n kubeflow
NAME                                READY   STATUS    RESTARTS   AGE
katib-controller-85d6f4497d-mvnw9   1/1     Running   0          171m
katib-db-manager-6dbf658d74-qhz6l   1/1     Running   0          171m
katib-mysql-6d756b5657-vvl6v        1/1     Running   0          171m
katib-ui-5954947794-vmvhj           1/1     Running   0          171m
tmp-pvc-pod                         1/1     Running   0          8s
hp@DESKTOP-1FTUR5O:~/katib-iris-v3/v3$ microk8s kubectl exec -n kubeflow -it tmp-pvc-pod -- /bin/sh
/model/
# ls /model/
best_model.pkl  model.pkl
```

Figure 11: tmp-pvc-pod creation and model artifact existence verification

```
hp@DESKTOP-1FTUR5O:/mnt/c/Users/hp$ microk8s kubectl cp kubeflow/tmp-pvc-pod:/model/best_model.pkl best_model2.pkl
hp@DESKTOP-1FTUR5O:/mnt/c/Users/hp$ microk8s kubectl cp kubeflow/tmp-pvc-pod:/model/model.pkl model2.pkl
```

Figure 12: Model recuperation from cluster to local host

*Note: After the file is successfully copied, the temporary pod can be deleted to free up resources.*

## 6.2 FastAPI Deployment and Testing

The following logic ensures the model is available for external requests.

```
from fastapi import FastAPI
import pickle, numpy as np

app = FastAPI()
model = pickle.load(open("model.pkl", "rb"))

```

```python
@app.post("/predict/")
def predict(data: list):
    # Converts input JSON into NumPy array for inference
    prediction = model.predict([np.array(data)])
    return {"prediction": prediction.tolist()}
```

Listing 4: main.py Deployment Logic

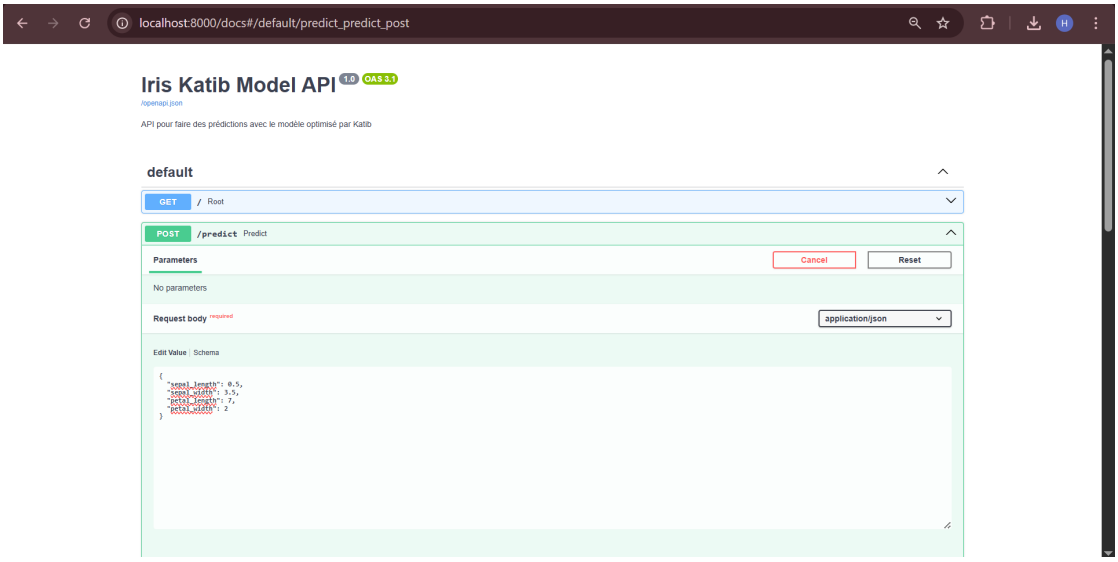


Figure 13: FastAPI

### 6.2.1 Testing the API

Once the FastAPI server is running (`uvicorn main:app -reload`), we verify it using `curl`:

```
curl -X POST "http://localhost:8000/predict/" \
     -H "Content-Type: application/json" \
     -d "[5.1, 3.5, 1.4, 0.2]"
```

*Expected Output: A JSON response containing the class prediction.*

```
PS C:\Users\hp> Invoke-RestMethod `
>>   -Uri http://localhost:8000/predict `
>>   -Method POST `
>>   -ContentType "application/json" `
>>   -Body '{ "sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.2 }'

prediction
----------
         1
```

Figure 14: API testing using PowerShell

# 7 Conclusion

This atelier demonstrated the complete lifecycle of an MLOps project. We successfully:

1. **Installed and Configured** a Kubernetes environment with Kubeflow Katib.
2. **Implemented Continuous NAS**, automating the search for the optimal neural network architecture using Early Stopping.
3. **Orchestrated Persistence** using PVCs, ensuring valuable model artifacts were not lost.
4. **Deployed the Solution** using FastAPI, bridging the gap between research (training) and production (serving).

This workflow ensures that our ML solutions are reproducible, scalable, and ready for real-world deployment.

# 8 Part 2: MNIST Training and Trial Management

## 8.1 Introduction

In this phase, we train a Fully Connected Neural Network (FCNN) on the MNIST dataset. The goal is to demonstrate the use of Katib for automatic hyperparameter optimization and trial tracking. Each trial corresponds to a different combination of parameters, allowing us to identify the model with the best performance.

## 8.2 Objectives

- Create a parameterized Python script for training the MNIST model.
- Define a Dockerfile to execute this script on Kubernetes.
- Configure a Katib Experiment to launch multiple trials and track performance.
- Identify the best model and save its weights to the persistent volume.

## 8.3 Python Script: `train_mnist.py`

This script utilizes PyTorch to train the model, dynamically accepting hyperparameters via command-line arguments.

```
import argparse
import torch
from torch import nn, optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import pickle

# Argument Parsing
```

```python
parser = argparse.ArgumentParser()
parser.add_argument("--layers", type=int, required=True)
parser.add_argument("--units", type=int, required=True)
parser.add_argument("--lr", type=float, required=True)
parser.add_argument("--export_path", type=str, required=True)
args = parser.parse_args()

# Data Loading
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000)

# Model Definition
layers = [784] + [args.units]*args.layers + [10]
model = nn.Sequential(*[nn.Linear(layers[i], layers[i+1]) for i in range(len(layers)-1)])
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=args.lr)

# Training with Early Stopping Logic
best_acc = 0
patience = 5
wait = 0

for epoch in range(20):
    model.train()
    for X, y in train_loader:
        X = X.view(X.size(0), -1)
        optimizer.zero_grad()
        output = model(X)
        loss = loss_fn(output, y)
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for X, y in test_loader:
            X = X.view(X.size(0), -1)
            pred = model(X).argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.size(0)
    acc = correct / total
    print(f"accuracy={acc}") # Metric for Katib

    if acc > best_acc:
        best_acc, wait = acc, 0
    else:
```

```
            wait += 1
        if wait >= patience:
            break

# Save Model
with open(f"{args.export_path}/mnist_model.pkl", "wb") as f:
    pickle.dump(model, f)
```

Listing 5: train_mnist.py

## 8.4 Dockerfile for MNIST

```
FROM python:3.10-slim

WORKDIR /app
COPY train_mnist.py /app/train_mnist.py

RUN pip install --no-cache-dir torch torchvision numpy

ENTRYPOINT ["python3", "/app/train_mnist.py"]
```

Listing 6: Dockerfile

## 8.5 Katib Experiment YAML: `mnist-nas.yaml`

```
apiVersion: kubeflow.org/v1beta1
kind: Experiment
metadata:
  name: mnist-nas
  namespace: kubeflow
spec:
  maxTrialCount: 5
  parallelTrialCount: 2
  maxFailedTrialCount: 3
  objective:
    type: maximize
    goal: 0.98
    objectiveMetricName: accuracy
  algorithm:
    algorithmName: random
  parameters:
  - name: layers
    parameterType: int
    feasibleSpace:
      min: "1"
      max: "3"
  - name: units
    parameterType: int
    feasibleSpace:
      min: "64"
      max: "256"
  - name: lr
```

```
    parameterType: double
    feasibleSpace:
      min: "0.0005"
      max: "0.01"
  trialTemplate:
    primaryContainerName: training-container
    trialParameters:
    - name: layers
      description: Number of hidden layers
      reference: "{{.TrialParameters.layers}}"
    - name: units
      description: Units per layer
      reference: "{{.TrialParameters.units}}"
    - name: lr
      description: Learning rate
      reference: "{{.TrialParameters.lr}}"
    trialSpec:
      apiVersion: batch/v1
      kind: Job
      spec:
        backoffLimit: 0
        template:
          spec:
            restartPolicy: Never
            containers:
            - name: training-container
              image: katib-mnist:latest
              command:
              - "python3"
              - "/app/train_mnist.py"
              args:
              - "--layers={{.TrialParameters.layers}}"
              - "--units={{.TrialParameters.units}}"
              - "--lr={{.TrialParameters.lr}}"
              - "--export_path=/model"
```

Listing 7: mnist-nas.yaml

## 8.6 Execution and Trial Tracking

We build the specific MNIST image and launch the experiment.

```
# 1. Build Docker image
docker build -t katib-mnist:latest .

# 2. Import image into MicroK8s registry
docker save katib-mnist:latest | microk8s ctr image import -

# 3. Launch the Katib Experiment
microk8s kubectl apply -f mnist-nas.yaml

# 4. Track trials in real-time
microk8s kubectl get trials -n kubeflow -w
```

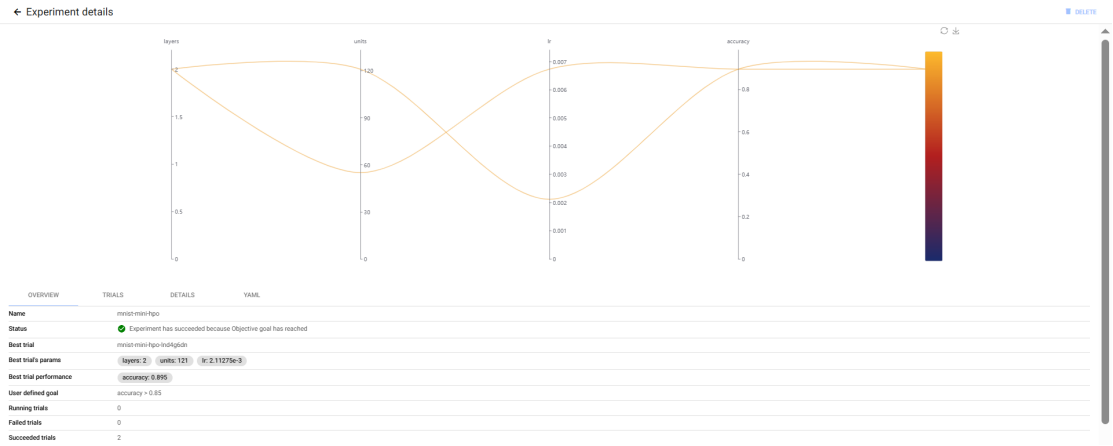


Figure 15: expirement details

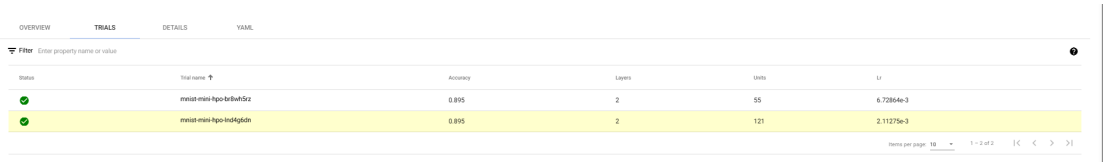


Figure 16: Trials

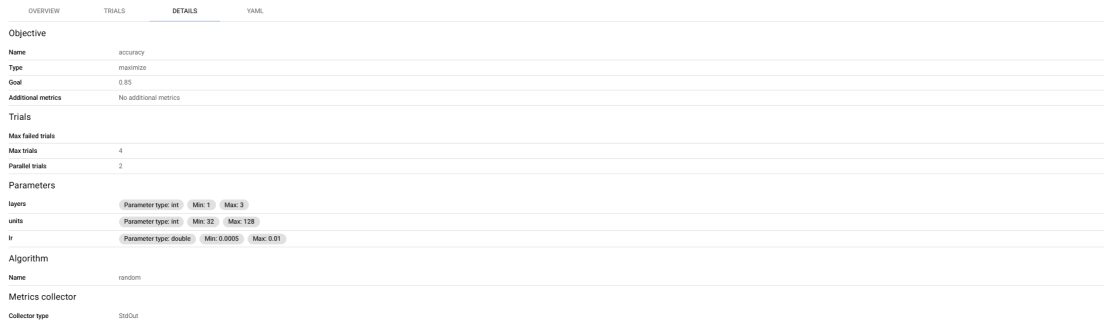


Figure 17: Config details

## 8.7 Conclusion

Through this phase, we have:

- Developed a flexible Python script for training MNIST using PyTorch.

- Created a Dockerfile to execute the model within Kubernetes.
- Utilized Katib to launch multiple trials and optimize hyperparameters.
- Saved the best-performing model for future use.

Cette approche illustre le workflow complet d'optimisation de modèles dans un environnement Kubernetes avec Katib.

# 9 MNIST Batch Inference Using Katib

After exploring the Iris dataset with continuous NAS, we extend the workflow to the MNIST dataset for batch inference.

## 9.1 Model Preparation

The MNIST model was pre-trained and stored as `model.pkl` in the PVC:

```
-rw-r--r-- 1 root root 3558862 Dec 20 18:26 model.pkl
```

**Role:** This model will be loaded by a Job for batch inference.

## 9.2 Bridge Pod for Model Access

Since pods are ephemeral, we use a temporary pod to access the model:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mnist-bridge-pod
  namespace: kubeflow
spec:
  containers:
    - name: mnist-container
      image: python:3.10-slim
      command: ["sleep", "3600"]
      volumeMounts:
        - name: model-volume
          mountPath: /model
  volumes:
    - name: model-volume
      persistentVolumeClaim:
        claimName: katib-iris-models-pvc
```

Listing 8: tmp-pvc-pod.yaml for MNIST

## 9.3 Batch Inference Job

We define a Kubernetes Job to run predictions on MNIST samples:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: mnist-batch-inference
  namespace: kubeflow
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: mnist-inference
        image: python:3.10-slim
        command: ["python3", "/app/mnist_inference.py"]
        volumeMounts:
        - name: model-volume
          mountPath: /model
      volumes:
      - name: model-volume
        persistentVolumeClaim:
          claimName: katib-iris-models-pvc
```

Listing 9: mnist-batch-job.yaml

## 9.4 Inference Script

The Python script `mnist_inference.py` loads the model and runs batch predictions:

```python
import pickle
import numpy as np
from tensorflow.keras.datasets import mnist

# Load MNIST data
(_, _), (X_test, y_test) = mnist.load_data()
X_test = X_test.reshape(len(X_test), -1) / 255.0

# Load trained model
with open("/model/model.pkl", "rb") as f:
    model = pickle.load(f)

# Run batch predictions
preds = model.predict(X_test[:100])
print("Predictions:", preds)
```

## 9.5 Execution

Deploy the Job and check logs:

```
microk8s kubectl apply -f mnist-batch-job.yaml
microk8s kubectl get pods -n kubeflow
microk8s kubectl logs -n kubeflow <mnist-pod-name>
```

## 9.6 Visualization

Optionally, the Kubernetes Dashboard can be used to monitor pod status, logs, and PVC mounting:

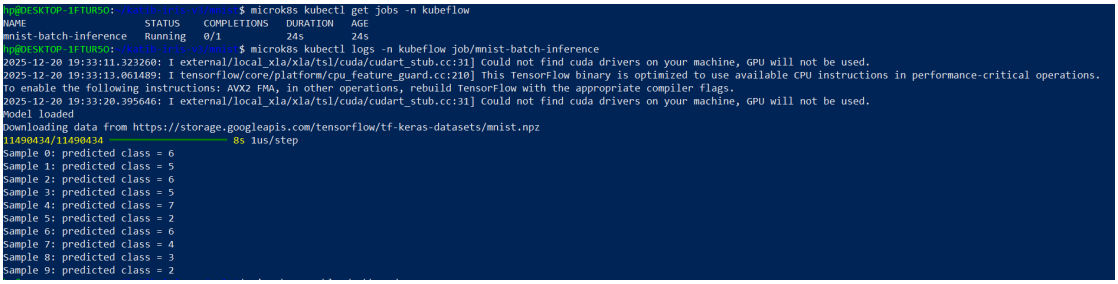


Figure 18: MNIST Batch Job running in the Kubernetes Dashboard.

## 9.7 Notes

- The model artifact is shared across experiments via the PVC.
- Using a bridge pod ensures safe retrieval and batch processing without losing data when pods terminate.
- This workflow can be extended to larger datasets and other models.

# 10 General Conclusion

This laboratory successfully demonstrated the implementation of a complete, production-ready MLOps pipeline, bridging the gap between static model development and dynamic, automated orchestration. By leveraging MicroK8s and Kubeflow Katib, we transitioned from manual hyperparameter tuning to an automated Neural Architecture Search (NAS).

Throughout this project, we explored two distinct use cases that highlighted the versatility of our infrastructure:

1. **The Iris Experiment (Part 1):** Established the fundamental workflow using Scikit-Learn. We successfully automated the search for an optimal MLP architecture, implemented **Early Stopping** to conserve resources, and deployed the final model as a **FastAPI** microservice.
2. **The MNIST Experiment (Part 2):** Demonstrated the scalability of the pipeline by applying it to a Deep Learning task using PyTorch. This phase confirmed that our containerized architecture is framework-agnostic and capable of handling complex image classification tasks.

### 10.1 Key Competencies Acquired

This atelier provided practical experience in the essential pillars of modern AI engineering:

- **Containerization (Docker):** Encapsulating dependencies to ensure that "it works on my machine" translates to "it works on the cluster."
- **Orchestration (Kubernetes/Katib):** Managing the lifecycle of multiple parallel trials without manual intervention.
- **Persistence (PVC):** Solving the challenge of ephemeral pods by implementing robust storage strategies for model artifacts.
- **Serving (FastAPI):** Exposing trained models as consumable APIs, completing the loop from research to production.

In conclusion, this architecture provides a robust foundation for scaling machine learning workflows. It ensures that models are not only accurate but also reproducible, deployable, and easier to maintain in a real-world industrial context.

## 11 References

`https://blog.kubeflow.org/katib/`