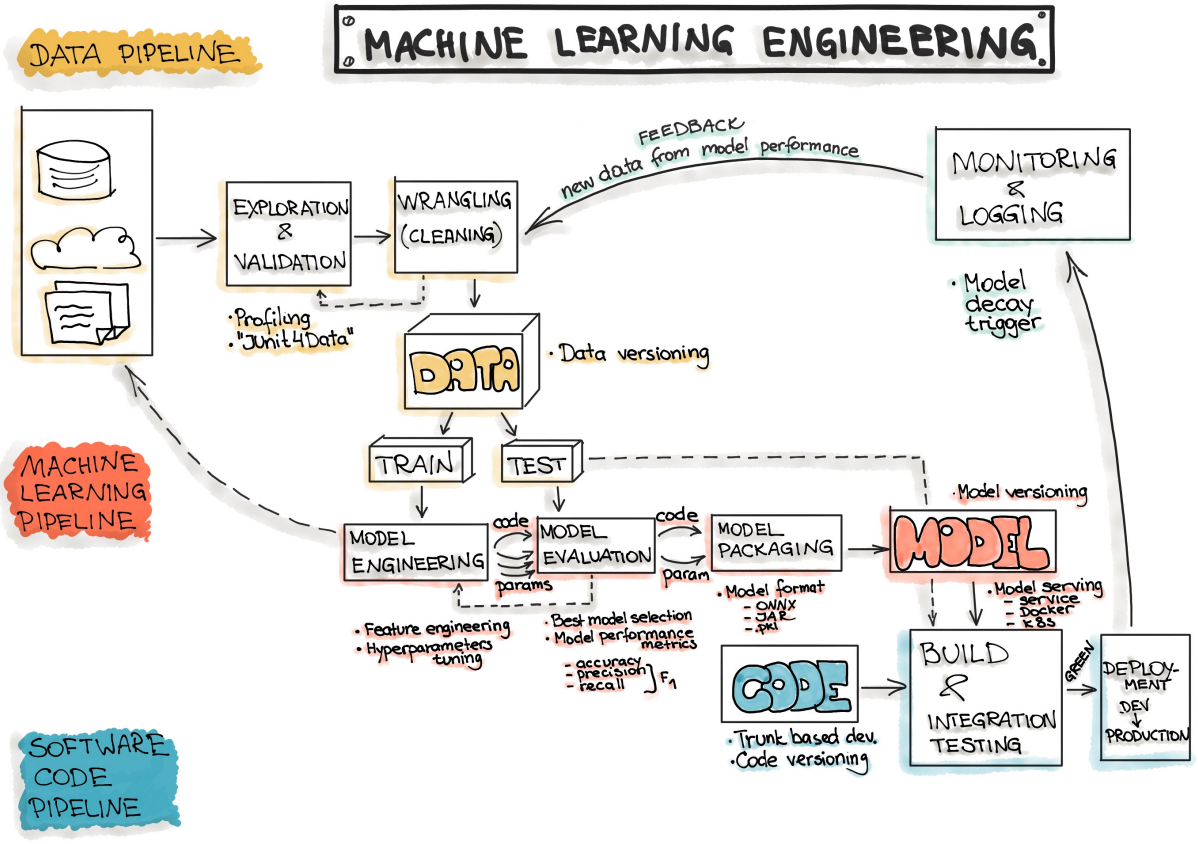
# Practical Machine Learning and Deep Learning. Lab 3. MLOps Tools.

## Practical Machine Learning and Deep Learning. Lab 3. MLOps Tools.

In the rapidly evolving landscape of machine learning and deep learning, the complexity of managing data pipelines, training models, deploying them into production, and continuously monitoring their performance has grown exponentially. As data science teams scale their efforts and deploy increasingly sophisticated models, the need for automation becomes obvious. Automation not only streamlines these processes but also ensures consistency, reproducibility, and scalability. By leveraging MLOps tools data scientists and engineers can significantly reduce the time and effort required to move from experimentation to production. This lab will introduce you to these tools.

## Automated Pipeline

The MLOps pipeline is a systematic approach to managing the lifecycle of machine learning models, from data acquisition and preprocessing to model training, deployment, and continuous monitoring. It integrates various stages of the machine learning workflow into a cohesive, automated process, ensuring that models are developed, tested, and deployed efficiently and reliably.




# Typical components of MLOps pipeline

## 1. Data Management

- **Data Collection**: Gathering raw data from various sources such as databases, APIs, or IoT devices.
- **Data Versioning**: Tracking changes in datasets over time, ensuring reproducibility.
- **Data Preprocessing**: Cleaning, transforming, and normalizing data to make it suitable for model training.

## 2. Model Development

- **Experiment Tracking**: Logging and comparing experiments, including hyperparameters, metrics, and artifacts.
- **Model Training**: Automating the training process.
- **Model Validation**: Performing cross-validation and testing to ensure the model's performance meets the required standards.

### 3. Model Deployment

- **Model Serving**: Deploying models and enabling real-time predictions.
- **Batch Processing**: Scheduling batch predictions real-time predictions are not necessary.

### 4. Monitoring and Maintenance

- **Performance Monitoring**: Monitoring model performance in production, including accuracy, latency, and resource utilization.
- **Drift Detection**: Continuously checking for data drift and concept drift to ensure the model remains relevant and accurate.
- **Retraining**: Automating the retraining process when new data becomes available or when the model's performance degrades.

## Benefits of an MLOps Pipeline

- **Reproducibility**: Ensures that experiments and deployments can be replicated consistently.
- **Scalability**: Allows for the efficient handling of large datasets and complex models.
- **Automation**: Reduces manual effort and human error, speeding up the development and deployment process.
- **Continuous Improvement**: Facilitates ongoing model refinement and updates based on real-world performance.

# Data Management

## Data Version Control (DVC)

Data Version Control (DVC) is an open-source version control system designed specifically for machine learning projects. It extends Git's functionality to manage large datasets, ML models, and experiments efficiently.

DVC is technically not a version control system by itself. It manipulates `.dvc` files, whose contents define the data file versions. Git is already used to version your code, and now it can also version your data alongside it.

## Basic Use Cases

- Track and save data and machine learning models the same way you capture code
- Create and switch between versions of data and ML models easily
- Understand how datasets and ML artifacts were built in the first place
- Compare model metrics among experiments
- Adopt engineering tools and best practices in data science projects

## Installation

DVC is available in Linux, Windows, and MacOS and can be installed as a Python library. Before installation make sure that you have installed git. To install it with pip you can run:

```
pip install dvc
```

If you have troubles with installation, here is a full installation guide.

## Initialization

To use your Git repository as a DVC project, run the following command in the folder of your repository:

```
dvc init
```

Then, commit the changes to GIt:

```
git commit -m "Initialize DVC"
```

Now you're ready to DVC!

## Remote Storage Configuration

You can upload DVC-tracked data to a variety of storage systems (remote or local) referred to as remotes. For simplicity, we will use a "local remote" for this guide, which is just a directory in the local file system. Before pushing data to a remote we need to set it up:

```
mkdir dvcstore
dvc remote add -d myremote dvcstore
```

if you want to set up a the storage on a real remote server, please check this guide.

## Data Tracking

If you have any data that you want to track, you can run `dvc add` to start tracking. For example, if you have a file with data `data/data.csv` you can run:

```
dvc add data/data.csv
```

DVC stores information about the added file in a special `.dvc` file. This small, human-readable metadata file acts as a placeholder for the original data for the purpose of Git tracking.

Next, run the following commands to track changes in Git. In the example with `data/data.csv` it will be:

```
git add data/data.csv.dvc data/.gitignore
git commit -m "Add raw data"
```

Now the metadata about your data is versioned alongside your source code, while the original data file was added to .gitignore.

## Data Uploading

Run `dvc push` to upload data:

```
dvc push
```

Usually, we would also want to Git track any code changes that led to the data change ( `git add` , `git commit` and `git push` ).

## Data Retrieving

Once DVC-tracked data and models are stored remotely, they can be downloaded with `dvc pull` when needed (e.g. in other copies of this project). Usually, we run it after `git pull` or `git clone` .

Run the following command to retrieve data:

```
dvc pull
```

## Switching between versions

A commonly used workflow is to use `git checkout` to switch to a branch or checkout a specific `.dvc` file revision, followed by a `dvc checkout` to sync data into your workspace:

```
git checkout <...>
dvc checkout
```

## Return to a previous version of the dataset

Run checkout to go back to the original version of the data:

```
git checkout HEAD~1 data/data.csv.dvc
dvc checkout
```

Now commit it (no need to do `dvc push` this time since this original version of the dataset was already saved):

```
git commit data/data.csv.dvc -m "Revert dataset updates"
```

### Other features

Besides versioning, DVC also provides data pipelines, experiment tracking and model management. If you are interested, check the following guides:

- **Data Pipelines** - Use DVC as a build system for reproducible, data driven pipelines.
- **Experiment Management** - Easily track your experiments and their progress by only instrumenting your code, and collaborate on ML experiments like software engineers do for code.
- **Model Registry** - Use the DVC model registry to manage the lifecycle of your models in an auditable way. Easily access your models and integrate your model registry actions into CICD pipelines to follow GitOps best practices.

# Model Development

## MLFlow

MLflow, at its core, provides a suite of tools aimed at simplifying the ML workflow. It is tailored to assist ML practitioners throughout the various stages of ML development and deployment.

## Use Cases

- **Experiment Tracking**

  A data science team leverages MLflow Tracking to log parameters and metrics for experiments within a particular domain. Using the MLflow UI, they can compare results and fine-tune their solution approach. The outcomes of these experiments are preserved as MLflow models.

- **Model Selection and Deployment**

  MLOps engineers employ the MLflow UI to assess and pick the top-performing models. The chosen model is registered in the MLflow Registry, allowing for monitoring its real-world performance.

- **Model Performance Monitoring**

  Post deployment, MLOps engineers utilize the MLflow Registry to gauge the model's efficacy, juxtaposing it against other models in a live environment.

- **Collaborative Projects**

  Data scientists embarking on new ventures organize their work as an MLflow Project. This structure facilitates easy sharing and parameter modifications, promoting collaboration.

## Installation

You can easily install MLflow as a Python package:

```
pip install mlflow
```

This will install the `mlflow` package and `mlflow` command.

MLflow works on MacOS. If you run into issues with the default system Python on MacOS, try installing Python 3 through the Homebrew package manager using `brew install python`. (In this case, installing MLflow is now `pip3 install mlflow`).

## Tracking User Interface

MLflow is one of the common tools used to track and log ML models and experiments. By default, wherever you run your program, the tracking API writes data into files into a local ./mlruns directory.

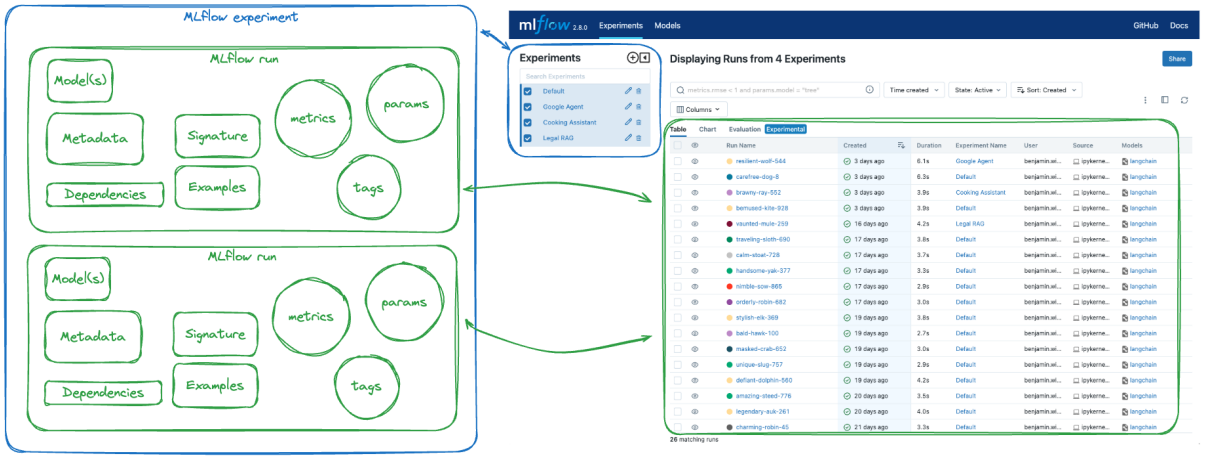
You can then run MLflow's Tracking UI by running:

```
mflow ui
```

or by running:

```
mlflow server -h localhost -p 5000
```

Then, you can open the user interface by this URL: http://localhost:5000/




## Autologging

Auto logging is a powerful feature that allows you to log metrics, parameters, and models without the need for explicit log statements. All you need to do is to call `mlflow.autolog()` before your training code.

```
import mlflow

# Enable autlogging
mlflow.autolog()

# Your ML modeling code is here.
```

This will enable MLflow to automatically log various information about your run, including:

- **Metrics** - MLflow pre-selects a set of metrics to log, based on what model and library you use
- **Parameters** - hyper params specified for the training, plus default values provided by the library if not explicitly set

- **Model Signature** - logs Model signature instance, which describes input and output schema of the model
- **Artifacts** - e.g. model checkpoints
- **Dataset** - dataset object used for training (if applicable), such as tensorflow.data.Dataset

Here are the steps to do autologging:

1. Insert `mlflow.autolog` in your training code

```
import mlflow

from sklearn.model_selection import train_test_split
from sklearn.datasets import load_diabetes
from sklearn.ensemble import RandomForestRegressor

mlflow.autolog()

db = load_diabetes()
X_train, X_test, y_train, y_test = train_test_split(db.dat
a, db.target)

rf = RandomForestRegressor(n_estimators=100, max_depth=6, m
ax_features=3)
# MLflow triggers logging automatically upon model fitting
rf.fit(X_train, y_train)
```

1. Execute your code

```
python YOUR_ML_CODE.py
```

2. View Your Results in the MLflow UI

```
mlflow ui
```

The more detailed description is available in this guide

## MLflow Models

An *MLflow Model* is a standard format for packaging machine learning models that can be used in a variety of downstream tools—for example, real-time serving through a REST API. The format defines a convention that lets you save a model in different **“flavors”** that can be understood by different downstream tools.

Each MLflow Model is a directory containing arbitrary files, together with an MLmodel file in the root of the directory that can define multiple flavors that the model can be viewed in.

**Logging sklearn models**

```
import mlflow
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_scor
e, recall_score, f1_score
from mlflow.models import infer_signature
import mlflow.sklearn
import mlflow.exceptions

# Load the Iris dataset
X, y = datasets.load_iris(return_X_y=True)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Define the model hyperparameters
params = {
    "solver": "lbfgs",
    "max_iter": 1000, # Use hydra for configuration managem
ent
    "random_state": 8888,
}

# Train the model
```

```
lr = LogisticRegression(**params)
lr.fit(X_train, y_train)

# Predict on the test set
y_pred = lr.predict(X_test)

# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="macr
o")
recall = recall_score(y_test, y_pred, average="macro")
f1 = f1_score(y_test, y_pred, average="macro")
print(accuracy, precision, recall, f1)


experiment_name = "MLflow experiment 01"
run_name = "run 01"
try:
    # Create a new MLflow Experiment
    experiment_id = mlflow.create_experiment(name=experimen
t_name)
except mlflow.exceptions.MlflowException as e:
    experiment_id = mlflow.get_experiment_by_name(experimen
t_name).experiment_id

print(experiment_id)

with mlflow.start_run(run_name=run_name, experiment_id=expe
riment_id) as run:

    # Log the hyperparameters
    mlflow.log_params(params=params)

    # Log the performance metrics
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("f1", f1)
    mlflow.log_metrics({
        "accuracy": accuracy,
```

```
        "f1": f1
    })

    # Set a tag that we can use to remind ourselves what th
is run was for
    mlflow.set_tag("Training Info", "Basic LR model for iri
s data")

        # Infer the model signature
    signature = infer_signature(X_test, y_test)


    # Log the model
    model_info = mlflow.sklearn.log_model(
        sk_model=lr,
        artifact_path="iris_model",
        signature=signature,
        input_example=X_test,
        registered_model_name="LR_model_01",
        pyfunc_predict_fn = "predict_proba"
    )

    sk_pyfunc = mlflow.sklearn.load_model(model_uri=model_i
nfo.model_uri)

    predictions = sk_pyfunc.predict(X_test)
    print(predictions)



    eval_data = pd.DataFrame(y_test)
    eval_data.columns = ["label"]
    eval_data["predictions"] = predictions

    results = mlflow.evaluate(
        data=eval_data,
        model_type="classifier",
        targets= "label",
```

```
        predictions="predictions",
        evaluators = ["default"]
    )

    print(f"metrics:\\n{results.metrics}")
    print(f"artifacts:\\n{results.artifacts}")
```

### Logging Pytorch models

```
import numpy as np
import mlflow
from mlflow.models import infer_signature
import torch
from torch import nn
import pandas as  pd

net = nn.Linear(10, 1)
loss_function = nn.L1Loss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-4)

X = torch.randn(100, 10)
y = torch.randn(100, 1)

print(X.shape, y.shape)

epochs = 5
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = net(X)

    loss = loss_function(outputs, y)
    loss.backward()

    optimizer.step()

with mlflow.start_run() as run:
    signature = infer_signature(X.numpy(), net(X).detach().
numpy())
```

```
    model_info = mlflow.pytorch.log_model(
        pytorch_model = net,
        artifact_path = "pytorch model",
        signature=signature,
        input_example=X.numpy(),
        registered_model_name="pytorch_model"
        )

    pytorch_pyfunc = mlflow.pyfunc.load_model(model_uri=mod
el_info.model_uri)

    X_test = torch.randn(20, 10).numpy()
    predictions = pytorch_pyfunc.predict(X_test)
    print(predictions)



    eval_data = pd.DataFrame(X.numpy())
    eval_data = pd.DataFrame(y.numpy())
    print(eval_data)
    eval_data.columns = ["label"]
    eval_data["predictions"] = net(X).detach().numpy()
    print(eval_data.shape)


    results = mlflow.evaluate(
        data=eval_data,
        model_type="regressor",
        targets= "label",
        predictions="predictions",
        evaluators = ["default"]
    )

print(f"metrics:\\n{results.metrics}")
print(f"artifacts:\\n{results.artifacts}")
```

**Fetch models from model registry**

```
import mlflow.pyfunc
from mlflow import MlflowClient

run_id = "e389609f9f1b44678ea7fea020453f94"
model_artifact_path = "pytorch model"

model = mlflow.pyfunc.load_model(model_uri=f"runs:/{run_i
d}/{model_artifact_path}")

print(model.metadata)

# OR

model_name = "pytorch_model"
model_version = 1

model = mlflow.pyfunc.load_model(model_uri=f"models:/{model
_name}/{model_version}")

print(model.metadata)

# OR

client = MlflowClient()
client.set_registered_model_alias(name = model_name, alias
= "staging", version = "1")

model_name = "pytorch_model"
model_alias = "staging"

model = mlflow.pyfunc.load_model(model_uri=f"models:/{model
_name}@{model_alias}")

print(model.metadata)
```

A comprehensive tutorial on working with Model Registry is available here.

# Deployment

In this section we will learn how to deploy a ML model with FastAPI and Docker.

## FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.


### Installation

FastAPI can be easily installed as a Python package

```
pip install "fastapi[standard]"
```

Note: Make sure you put "fastapi[standard]" in quotes to ensure it works in all terminals.

## Docker

Docker is an open-source platform designed for building, deploying, and managing applications in lightweight, portable containers. Containers encapsulate an application along with all its dependencies, including libraries and configuration files, allowing it to run consistently across different computing environments without the need for complex setups.


## Key Features

- **Containerization**

  Docker uses OS-level virtualization to create containers, which are isolated from each other and share the same operating system kernel. This makes containers more resource-efficient compared to traditional virtual machines, which require separate operating systems for each instance.

- **Docker Engine**

  The core component of Docker is the Docker Engine, which consists of a server (Docker daemon), a REST API, and a command-line interface (CLI). The daemon manages the containers and images, while the CLI allows users to interact with the daemon.

- **Images and Containers**

  A Docker image is a read-only template used to create containers. When an image is executed, it becomes a container, which is a running instance of that image. Containers can be easily created, started, stopped, and deleted, making them highly flexible for development and deployment.

- **Docker Hub**

  This is a cloud-based repository where users can share and manage Docker images. It provides access to thousands of pre-built images, facilitating easy deployment and collaboration among developers.

- **Microservices and Scalability**

  Docker is particularly popular for microservices architecture, allowing developers to deploy applications as a suite of small, independent services that can be scaled easily. This modular approach enhances the ability to update and maintain applications without downtime.

- **Cross-Platform Compatibility**

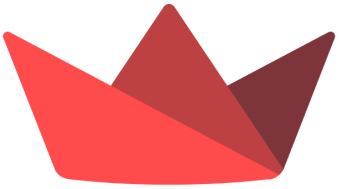
  Docker can run on various operating systems, including Linux, Windows, and macOS, making it versatile for development across different environments.

### Installation

Installation of Docker is highly OS-dependent and contains many steps. Please, check this installation guide and follow the steps of installation that correspond to your OS.

## Streamlit

Streamlit is an open-source Python framework designed to enable data scientists and machine learning engineers to quickly create and share interactive web applications. It simplifies the web app development process, allowing users to build visually appealing applications with minimal coding effort.



# Streamlit

### Use Cases

- **Data Science Applications**: Building dashboards and interactive reports that visualize data analysis results.
- **Machine Learning Demos:** Creating applications to demonstrate machine learning models, allowing users to interact with the models in real-time.
- **Prototyping:** Rapidly developing prototypes for data-centric applications, facilitating quick feedback and iteration.

### Installation

Streamlit can be installed as a Python package:

```
pip install streamlit
```

## Model Deployment with FastAPI, Docker and Streamlit

Deploying a machine learning model using FastAPI, Docker and Streamlit involves creating an API for your model, containerizing it for easy deployment, and creating a frontend web application for interactive use. Before doing these steps make sure that Docker is installed.

## Step 1: Create the FastAPI Application

Create a Python file called `app.py` for your FastAPI app.

```
# app.py
from fastapi import FastAPI
from pydantic import BaseModel
import pickle

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

# Define the FastAPI app
app = FastAPI()

# Define the input data schema
class IrisInput(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float

# Define the prediction endpoint
@app.post("/predict")
def predict(input_data: IrisInput):
    data = [[
        input_data.sepal_length,
        input_data.sepal_width,
        input_data.petal_length,
        input_data.petal_width
    ]]

    prediction = model.predict(data)
    return {"prediction": int(prediction[0])}
```

This app accepts JSON input through a POST request at /predict, feeds it to the loaded model, and returns the prediction.

## Step 2: Create Streamlit App for Frontend

Create a file called `streamlit_app.py`:

```
# streamlit_app.py
import streamlit as st
import requests

# FastAPI endpoint
FASTAPI_URL = "http://fastapi:8000/predict"

# Streamlit app UI
st.title("Iris Flower Classifier")

# Input fields for the Iris flower data
sepal_length = st.number_input("Sepal Length", min_value=0.
0)
sepal_width = st.number_input("Sepal Width", min_value=0.0)
petal_length = st.number_input("Petal Length", min_value=0.
0)
petal_width = st.number_input("Petal Width", min_value=0.0)

# Make prediction when the button is clicked
if st.button("Predict"):
    # Prepare the data for the API request
    input_data = {
        "sepal_length": sepal_length,
        "sepal_width": sepal_width,
        "petal_length": petal_length,
        "petal_width": petal_width
    }

    # Send a request to the FastAPI prediction endpoint
    response = requests.post(FASTAPI_URL, json=input_data)
    prediction = response.json()["prediction"]

    # Display the result
    st.success(f"The model predicts class: {prediction}")
```

This creates a simple interface where users can input data for the Iris flower classification task, and when they click the "Predict" button, it sends the input data to the FastAPI backend for prediction.

## Step 3: Create Dockerfile for FastAPI

A `Dockerfile` is used to define the Docker image for your FastAPI app. Create a file called `Dockerfile` in the same directory as `app.py`.

```
# Use an official Python runtime as a parent image
FROM python:3.9

# Set the working directory
WORKDIR /app

# Copy the current directory contents into the container at
/app
COPY . /app

# Install the necessary packages
RUN pip install fastapi uvicorn scikit-learn

# Expose port 80 to the outside world
EXPOSE 80

# Run the FastAPI server
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port",
"80"]
```

This Dockerfile defines the environment where your app will run, installs dependencies, and starts the FastAPI app with Uvicorn.

## Step 4: Create Dockerfile for Streamlit

Similarly, create a Dockerfile for the Streamlit app:

```
# Streamlit Dockerfile
FROM python:3.9

# Set the working directory
```

```
WORKDIR /app

# Copy the current directory contents into the container at
/app
COPY . /app

# Install the dependencies
RUN pip install streamlit requests

# Expose the default Streamlit port
EXPOSE 8501

# Command to run the Streamlit app
CMD ["streamlit", "run", "streamlit_app.py", "--server.port
=8501", "--server.address=0.0.0.0"]
```

This Dockerfile defines the environment for running the Streamlit frontend. It installs the required dependencies and exposes port 8501 (the default port for Streamlit apps).

## Step 5: Create Docker Compose File

To orchestrate the FastAPI and Streamlit containers together, we will use Docker Compose. Create a `docker-compose.yml` file:

```
version: '3'

services:
  fastapi:
    build:
      context: .
      dockerfile: Dockerfile.fastapi
    container_name: fastapi
    ports:
      - "8000:8000"

  streamlit:
    build:
      context: .
```

```
      dockerfile: Dockerfile.streamlit
    container_name: streamlit
    ports:
      - "8501:8501"
    depends_on:
      - fastapi
```

Ensure the Dockerfiles for FastAPI and Streamlit are named as `Dockerfile.fastapi` and `Dockerfile.streamlit` respectively.

### Step 6: Build and Run with Docker Compose

To build and run the services, use the following commands in the directory where the `docker-compose.yml` file is located:

```
# Build the containers
docker-compose build

# Run the containers
docker-compose up
```

Docker Compose will build and start both the FastAPI and Streamlit containers. FastAPI will be available at http://localhost:8000, and Streamlit at http://localhost:8501.

# Monitoring

## Evidently AI

Evidently helps evaluate and track quality of ML-based systems, from experimentation to production.

Evidently is both a library of 100+ ready-made evaluations, and a framework to easily implement yours: from Python functions to LLM judges.

Evidently has a modular architecture, and you can start with ad hoc checks without complex installations. There are 3 interfaces: you can get a visual `Report` to see a summary of evaluation metrics, run conditional checks with a `TestSuite` to get a pass/fail outcome, or plot the evaluation results over time on a `Monitoring Dashboard`.


**For the sake of simplicity of the lab and the assignment we will not use this tool**. However, if you are interested in it, welcome to the documentation of Evidently AI!

# Workflow Orchestration

## Apache Airflow

Apache Airflow is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. The main characteristic of Airflow workflows is that all workflows are defined in Python code, such that "Workflows as code".

Airflow is a platform that lets you build and run **workflows**. A workflow is represented as a **DAG** (a Directed Acyclic Graph), and contains individual pieces of work called **Tasks**, arranged with dependencies and "data" flows taken into account. Airflow is a **batch workflow orchestration platform**.

## Installation and Setup

There are two ways of Apache Airflow installation: with `pip` and with `docker`. Airflow does not support Windows, so use Docker or WSL. Check this guide if you want to install Airflow with docker

### Installation with pip for Linux

Create a new virtual environment using Python 3.11

```
# Create .venv
python3.11 -m venv .venv

# Activate it
source .venv/bin/activate

# You can deactivate it at anytime
# deactivate
```

Install Airflow to your new virtual environment:

```
pip install apache-airflow
```

Before running Airflow we need to customize the Airflow config folder as follows:

```
# PWD is project folder path
export AIRFLOW_HOME=$PWD/services/airflow
```

Make sure that you set this environment variable before you work with Airflow. This will store the configs and metadata in the specified folder.

Add this to your ~/.bashrc file as a permanent variable but replace $PWD with your absolute project folder path:

```
# REPLACE <project-folder-path> with your project folder pa
th
cd <project-folder-path>
echo "export AIRFLOW_HOME=$PWD/services/airflow" >> ~/.bash
rc

# Run/Load the file content
source ~/.bashrc

# Activate the virtual environment again
source .venv/bin/activate
```

Now we will use a minimal setup of Metadata database and Executor (SequentialExecutor + SQLite). Initialize the metadata database:

```
# Clean it
airflow db reset

# initialize the metadata database
airflow db init
```

Then, we setup Airflow webserver by creating a new user and add it to the database:

```
# Here we are creating admin user with Admin role
airflow users create \
--role Admin \
--username admin \
--email admin@example.org \
--firstname admin \
--lastname admin \
--password admin
```

After that we run Airflow components. Firstly, create folders for logs:

```
# Create folders and files for logging the output of compon
ents
mkdir -p $AIRFLOW_HOME/logs $AIRFLOW_HOME/dags
```

```
echo > $AIRFLOW_HOME/logs/scheduler.log
echo > $AIRFLOW_HOME/logs/triggerer.log
echo > $AIRFLOW_HOME/logs/webserver.log

# Add log files to .gitignore
echo *.log >> $AIRFLOW_HOME/logs/.gitignore
```

Start the scheduler component:

```
airflow scheduler --daemon --log-file services/airflow/logs/scheduler.log
```

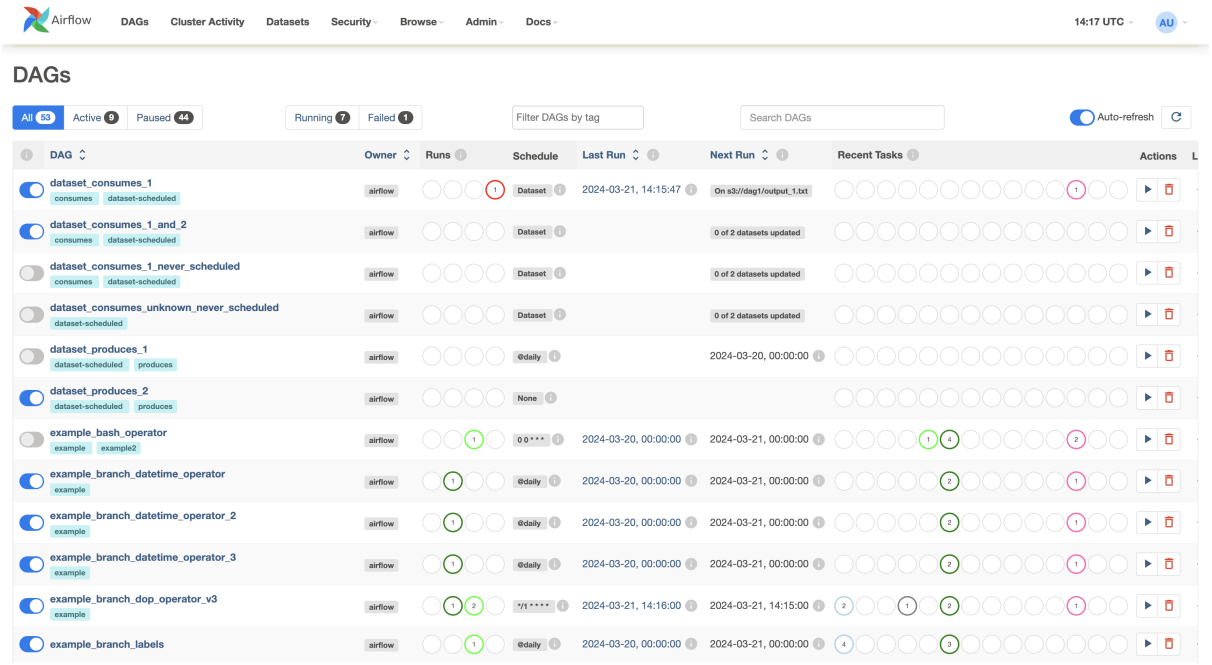
Start the webserver component:

```
airflow webserver --daemon --log-file services/airflow/logs/webserver.log
```

Start the triggerer component:

```
airflow triggerer --daemon --log-file services/airflow/logs/triggerer.log
```

You can access the webserver UI in (http://localhost:8080). The password is `admin` and username is `admin` if you did not change it when you created the admin user.

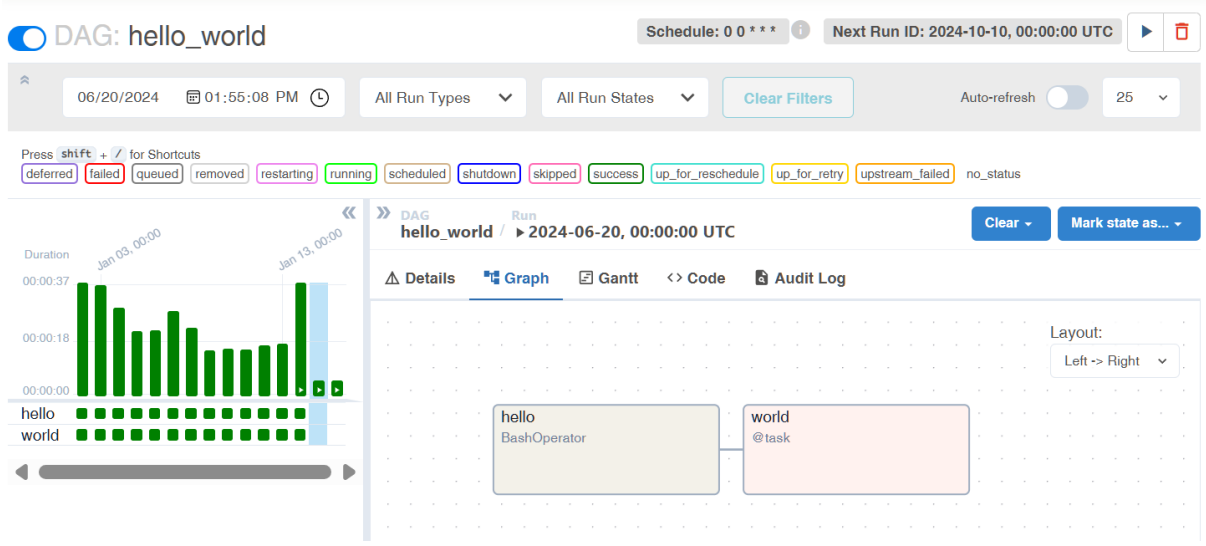



## Hello World DAG

```
# services/airflow/dags/hello_world.py

from datetime import datetime

from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
# This DAG is scheduled to print 'hello world' every minute
starting from 01.01.2022.
with DAG(dag_id="hello_world",
         start_date=datetime(2022, 1, 1),
         schedule="* * * * *") as dag:

    # Tasks are represented as operators
    # Use Bash operator to create a Bash task
    hello = BashOperator(task_id="hello", bash_command="ech
o hello")

    # Python task
    @task()
```

```
    def world():
        print("world")

    # Set dependencies between tasks
    # First is hello task then world task
    hello >> world()
```

In this workflow, we can see two tasks. The first one is defined using operators and the second one is defined using @task decorator.




## DAGs Scheduling

DAG (Directed Acyclic Graph) is essentially a collection of tasks that are arranged with dependencies and schedules. The DAG scheduling in Airflow defines when and how often these tasks should run.

### Schedule Interval

The `schedule_interval` parameter defines how often the DAG should be triggered. This can be set to different values like:

- Cron expressions: `0 12 * * *` (run every day at 12:00 PM)
- Preset strings: `@daily` Once every day (midnight)
- timedelta object: e.g., `timedelta(days=1)` for daily runs.

### Start Date

The `start_date` parameter defines the earliest date and time when the DAG should be first triggered. Airflow will not start the DAG earlier than this date,

regardless of the schedule interval.

### Catchup

When Airflow runs, it looks at the last run and compares it to the current time. If there are any intervals where the DAG should have run but didn't (based on the `schedule_interval` and `start_date`), Airflow will run these past intervals if `catchup=True` (which is the default). If you don't want to backfill, you can set `catchup=False`.

### Execution Date

The execution date refers to the logical date the DAG run is meant to process. It is typically in the past, even though the DAG runs in the present. For example, a daily DAG that runs at midnight on July 2nd will have an execution date of July 1st.

### Timezone Awareness

By default, Airflow uses UTC for scheduling, but you can configure it to use other time zones. This ensures that your DAGs are triggered according to the correct timezone you prefer.

## DAG Examples

### Traditional API as a variable

```
from pendulum import datetime
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

# A DAG represents a workflow, a collection of tasks
# It is a variable
dag = DAG(dag_id="hello_dag1",
          start_date=datetime(2022, 1, 1, tz="UTC"),
          schedule=None,
          catchup=False)

# Tasks here are created via instantating operators

# You need to pass the dag for all tasks
```

```
# Bash task
hello = BashOperator(task_id="hello1",
                     bash_command="echo hello ",
                     dag = dag)


def msg():
    print("airflow!")

# Python task
msg = PythonOperator(task_id="msg1",
                     python_callable=msg,
                     dag=dag)

# Set dependencies between tasks
hello >> msg
```

**Traditional API as a context**

```
from pendulum import datetime
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

# A DAG represents a workflow, a collection of tasks
# DAG is defined as a context
with DAG(dag_id="hello_dag2",
         start_date=datetime(2022, 1, 1, tz="UTC"),
         schedule=None,
         catchup=False) as dag:


    # WE do NOT pass dag here

    hello = BashOperator(task_id="hello2",
                         bash_command="echo hello ")

    def msg():
```

```
        print("airflow!")

    called_msg = PythonOperator(task_id="msg2",
                        python_callable=msg)


    # Set dependencies between tasks
    # hello >> msg
    hello.set_downstream(called_msg)
    # OR
    # msg.set_upstream(hello)
```

**TaskFlow API**

```
from pendulum import datetime
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.models.baseoperator import chain

@dag(
    dag_id="hello_dag3",
    start_date=datetime(2022, 1, 1, tz="UTC"),
    schedule=None,
    catchup=False
)
def print_hello():

    # Tasks here are created via instantating operators

    # Bash task is defined using Traditional API
    # There is a decorator @task.bash in Airflow 2.9+ as a
replacement for this
    hello = BashOperator(task_id="hello3", bash_command="ec
ho hello ")

    # Python task is defined sing TaskFlow API
    @task(
```

```
        task_id = "msg31"
    )
    def msg():
        """Prints a message"""
        print("airflow!")

    called_msg = PythonOperator(task_id="msg32", python_cal
lable=msg)



    # Set dependencies between tasks
    # hello >> msg
    # OR
    # hello.set_downstream(msg)
    # OR
    # msg.set_upstream(hello)
    # OR
    chain(hello, msg)

# Call the pipeline since it is defined as a function
print_hello()
```

**ExternalTaskSensor**

```
import pendulum

from airflow.models.dag import DAG
from airflow.operators.empty import EmptyOperator
from airflow.operators.bash import BashOperator
from airflow.sensors.external_task import ExternalTaskMarke
r, ExternalTaskSensor
from datetime import timedelta
from airflow.decorators import task

start_date = pendulum.datetime(2024, 6, 27, 19, 30, tz = "E
urope/Moscow")
```

```
with DAG(
    dag_id="example_external_task_sensor_parent",
    start_date=start_date,
    catchup=False,
    schedule=timedelta(minutes=1),
    tags=["example2"],
) as parent_dag:

    parent_task = BashOperator(
        task_id = "parent_task",
        bash_command="echo Run this before! ",
        cwd="/"
    )


with DAG(
    dag_id="example_external_task_sensor_child",
    start_date=start_date,
    schedule=timedelta(minutes=1),
    catchup=False,
    tags=["example2"],
) as child_dag:

    child_task1 = ExternalTaskSensor(
        task_id="child_task1",
        external_dag_id=parent_dag.dag_id,
        external_task_id=parent_task.task_id,
        timeout=600
    )

    @task(task_id = "child_task2")
    def run_this_after():
        print("I am running!")

    child_task2 = run_this_after()


    child_task1 >> child_task2
```

### Trigger Dag Run operator

It triggers a DAG run for a specified `dag_id`.

```
import pendulum

from airflow.models.dag import DAG
from airflow.operators.trigger_dagrun import TriggerDagRunO
perator

with DAG(
    dag_id="example_trigger_controller_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    schedule="@once",
    tags=["example"],
) as dag:
    trigger = TriggerDagRunOperator(
        task_id="test_trigger_dagrun",
        # Ensure this equals the dag_id of the DAG to trigg
er
        trigger_dag_id="example_trigger_target_dag",
    )
```

```
import pendulum

from airflow.decorators import task
from airflow.models.dag import DAG
from airflow.operators.bash import BashOperator


@task(task_id="run_this")
def run_this_func(dag_run=None):
    """
    Print the payload "message" passed to the DagRun conf a
ttribute.

    :param dag_run: The DagRun object
```

```
    """
    print("triggerred task!")


with DAG(
    dag_id="example_trigger_target_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    schedule=None,
    tags=["example"],
) as dag:
    run_this = run_this_func()

    bash_task = BashOperator(
        task_id="bash_task",
        bash_command='sleep 60 && echo "Run this after the
target"'
    )
```

### DAG Testing

You can test a pipeline from airflow CLI as follows:

```
airflow dags test <dag-id>
```

This will run the dag only for one time. This is not scheduling dags.

You can also test it in VS code by adding the code snippet below to the end of the dag definition file:

```
if __name__ == "__main__":
    dag.test()
    # dag is an instance of DAG
```

# It may be useful for you

- DVC Get Started Guide
- MLFlow Get Started Guide

- FastAPI Website
- Docker Tutorial for Beginners
- Evidently AI Tutorials
- Apache Airflow Tutorial
- Get Started with Streamlit

# References


- F. Jolha “MLOps Engineering Course“
- DVC Linux Installation
- MLflow Quickstart
- MLflow autologging
- FastAPI Installation