D ONE

WE MAKE SENSE.

DATA DRIVEN VALUE CREATION

DATA SCIENCE & ANALYTICS | DATA MANAGEMENT | VISUALIZATION & DATA EXPERIENCE

D ONE Solutions AG, Sihlfeldstrasse 58, 8003 Zürich, d-one.ai

How to Use Machine Learning in Production (MLOps)

Steffen Terhaar, Tim Rohner, Bernhard Venneman,
Spyros Cavadias & Roman Moser

CD4ML Working Group @ D ONE

Workshop Case Study

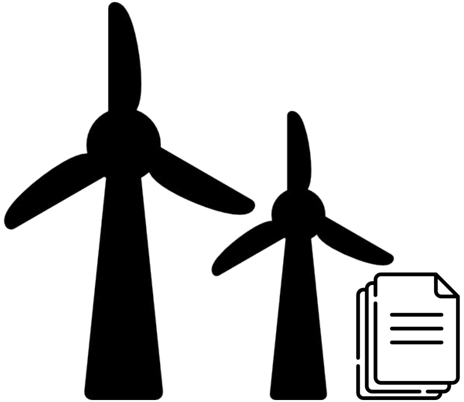
A Zurich-based, data-driven company that provides an analytics platform for wind turbine parks

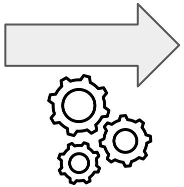
winji

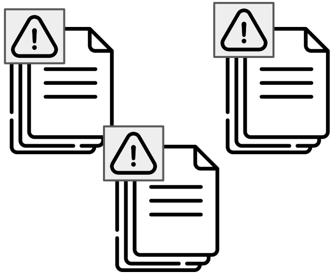
Goal

Build and productionalize a model to predict turbine malfunctions

ML problem statement





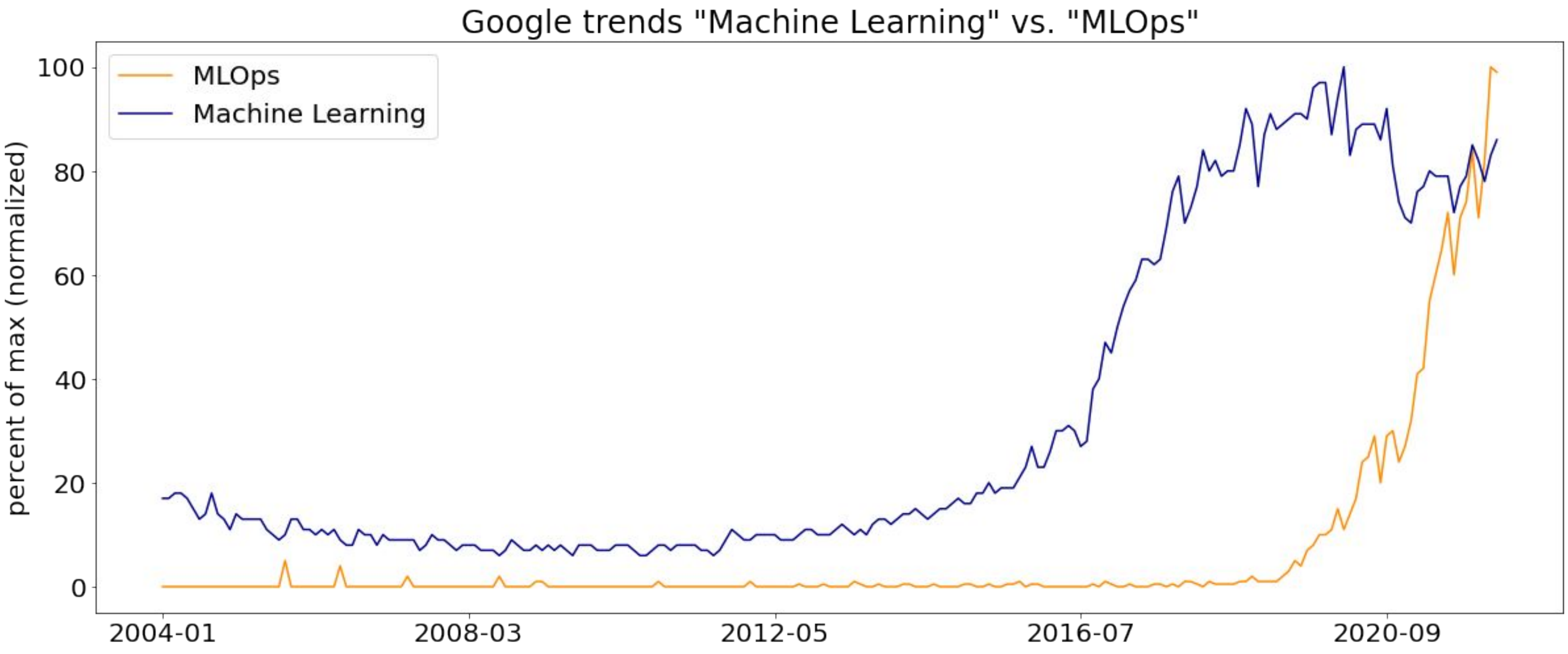


Given measurement data from wind turbines

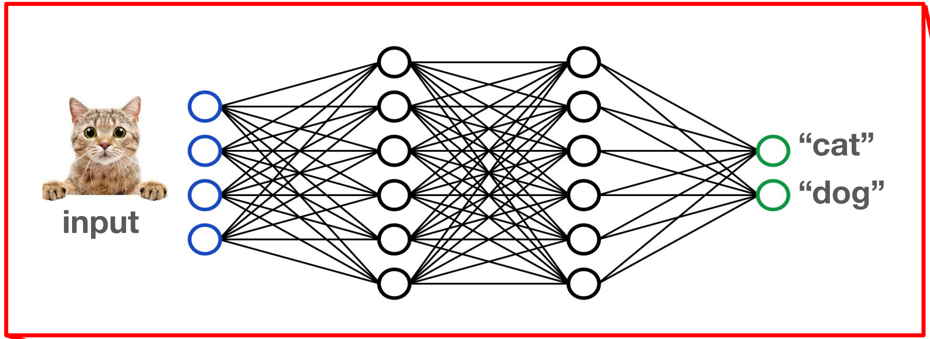
we would like to predict

error types appearing

Machine learning and MLOps popularity

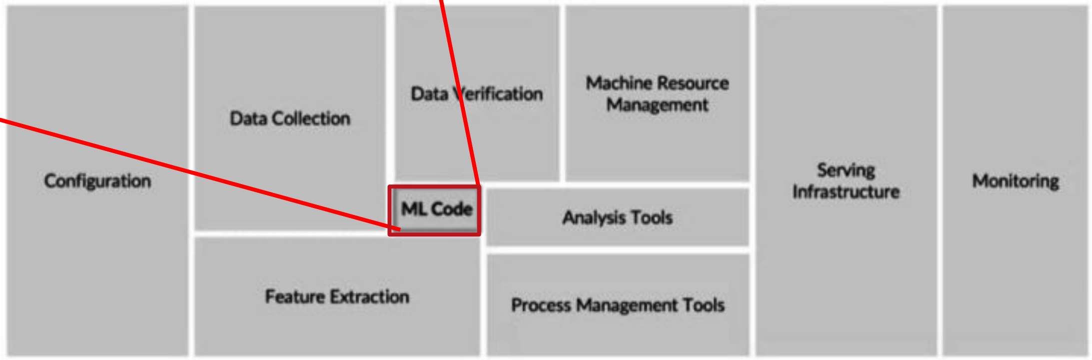


ML model development vs MLOps



Developing a machine learning model is challenging and fun …

… however, it is only one (small) piece of the puzzle

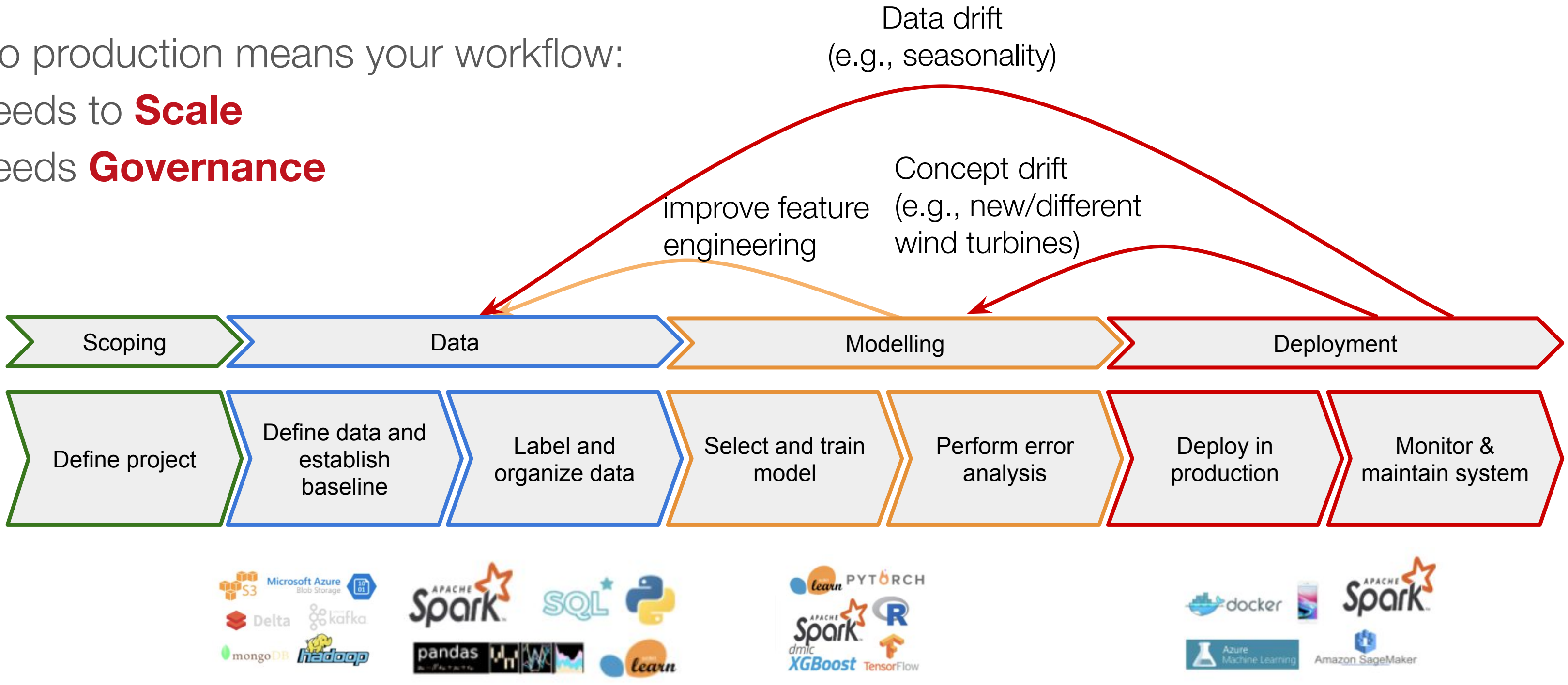


Source: Andrew Ng

Machine Learning Pipeline

Going to production means your workflow:

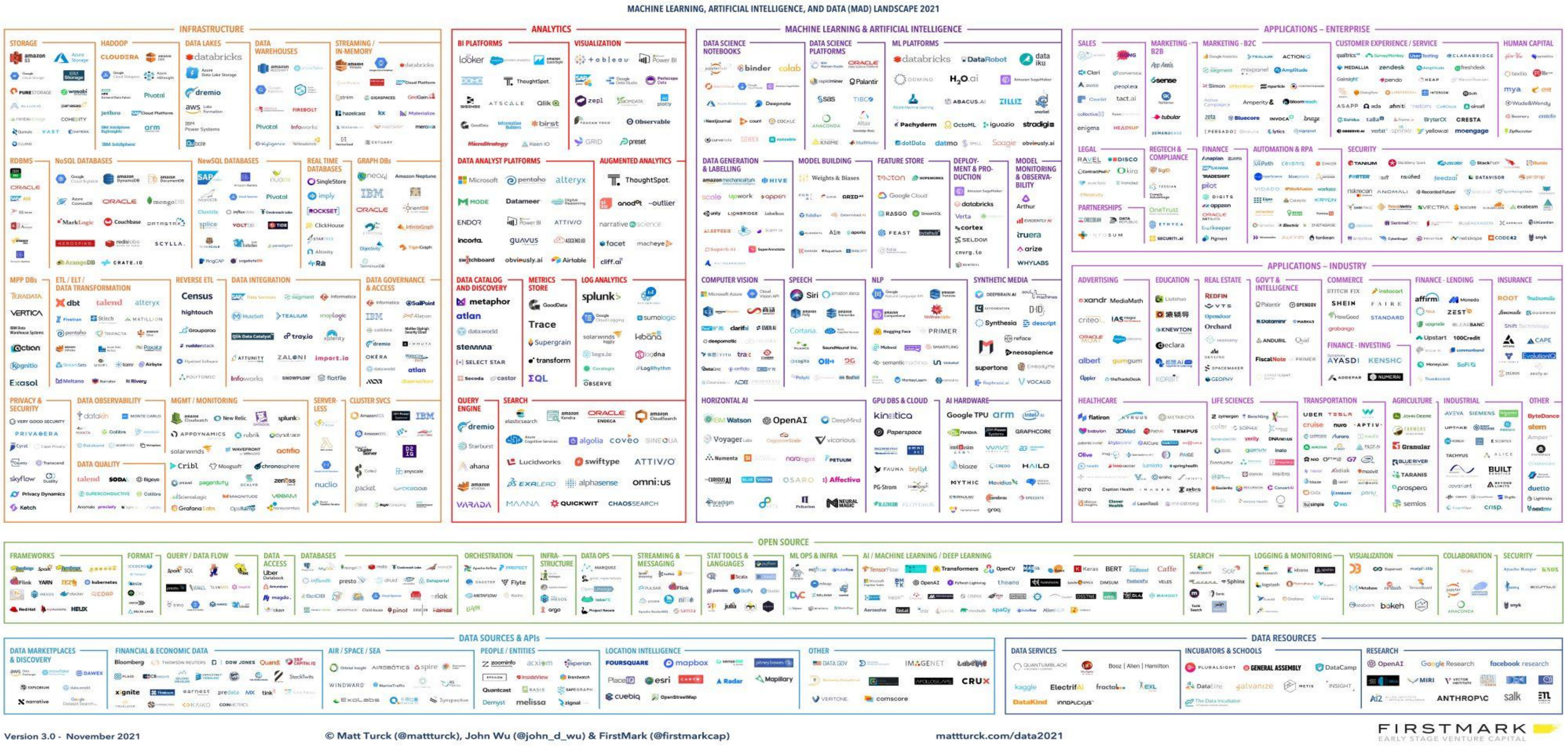
- Needs to **Scale**
- Needs **Governance**



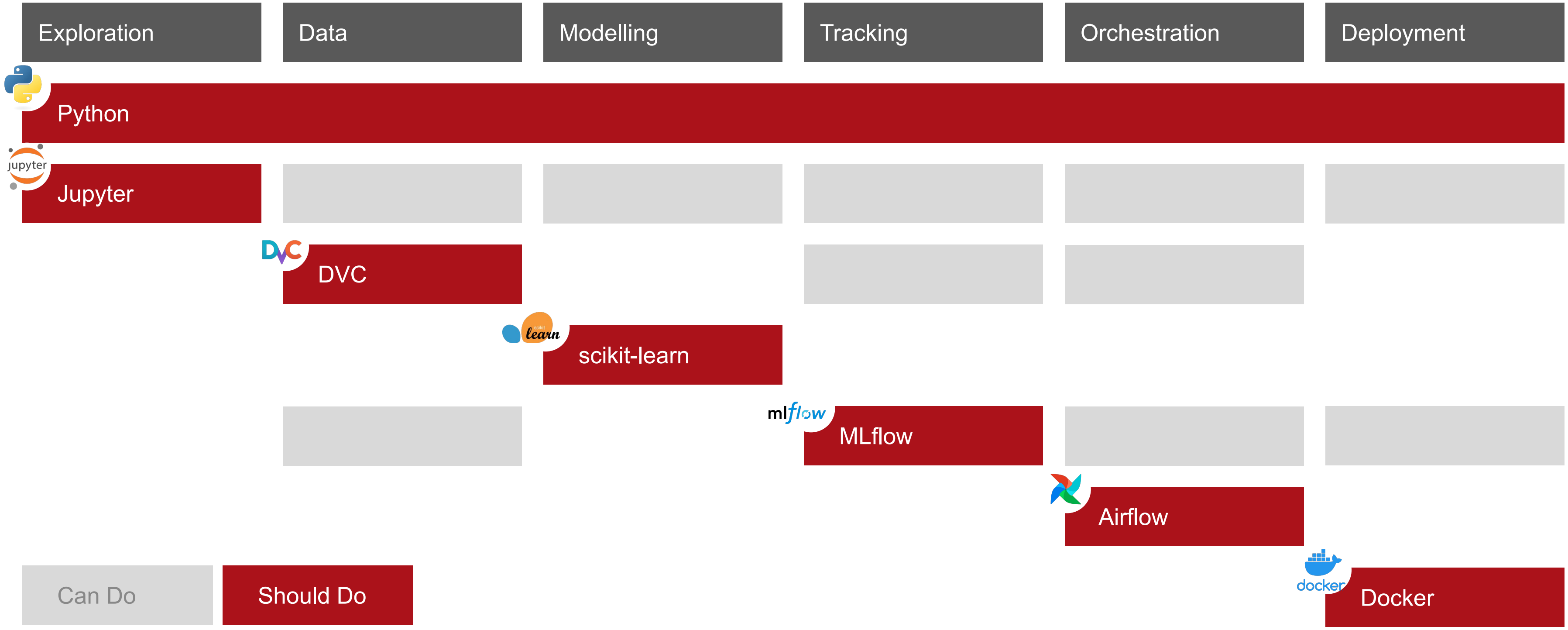
For this case study we use tools from the open source stack

- We want to provide a solution that:
  - Works on-premise
  - Works in the cloud (independent of the cloud provider)
  - Gives you complete control

Which tools are ideal?



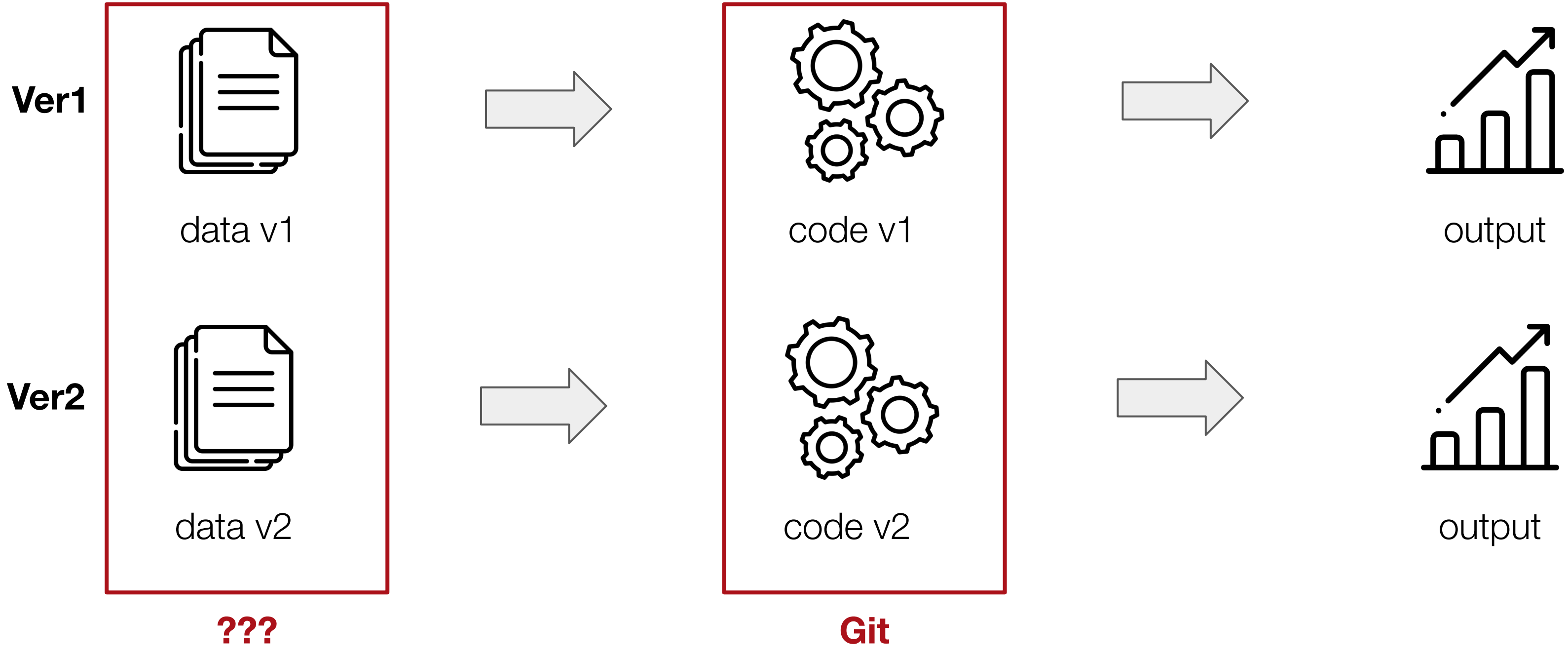
A Selection of an Open Source Stack



How To:

Data Tracking

Why data versioning?



What DVC offers

- **Data and model versioning**
- **Remote data storage**
- Data pipelines
- Experiments and metrics tracking

DVC features

Git-compatible

Storage agnostic

Reproducible

Low friction branching

Metric tracking

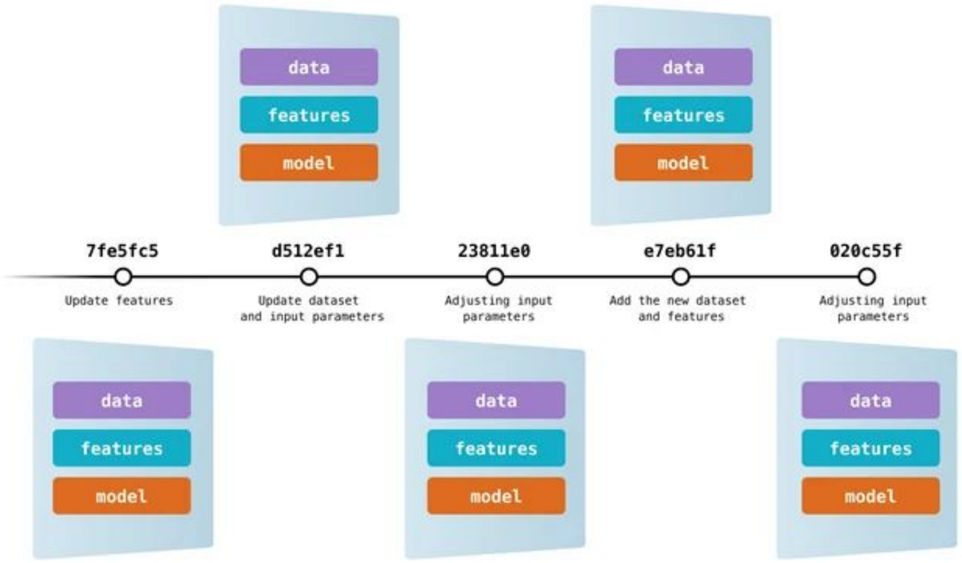
ML pipeline framework

Language- & framework-agnostic

HDFS, Hive & Apache Spark

Track failures

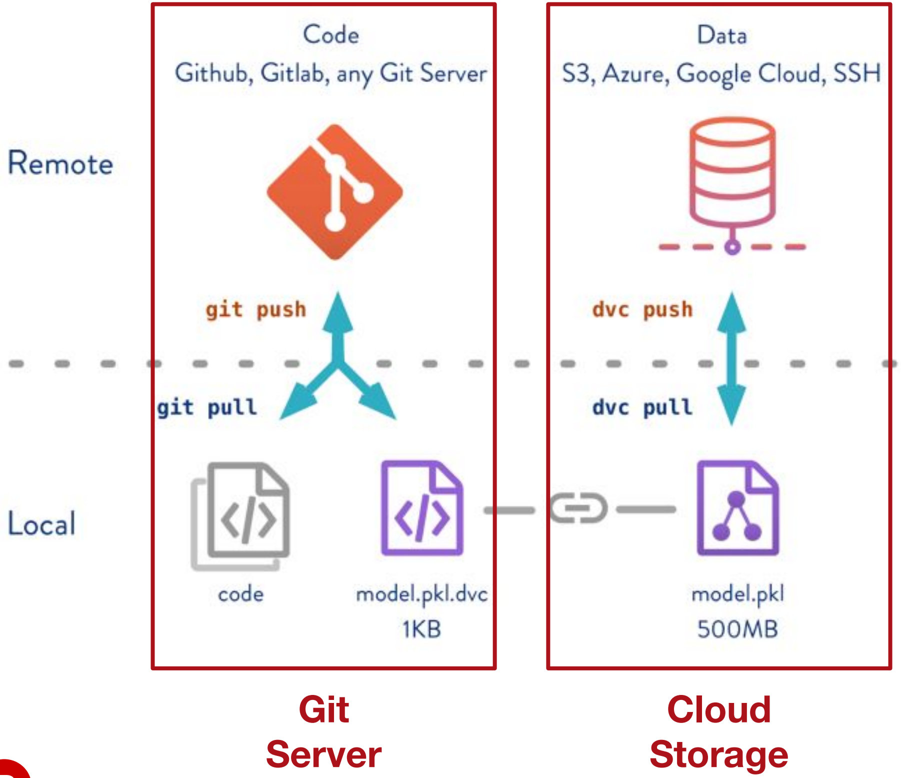
How DVC data versioning works



Basic idea

Keep versions not only of your code, but also of your data, pipeline and models.

How DVC data versioning works



Problem with Git and data

Git is not suitable for tracking large files. Thus, tracking entire datasets and large models is not feasible with Git.

Solution

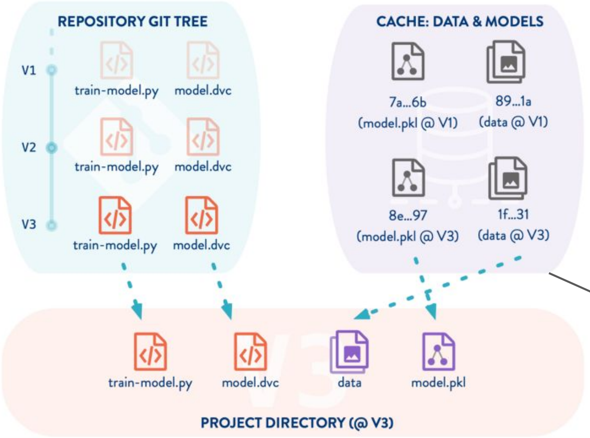
DVC!

image source: dvc.org

Advantages over Git LFS

DVC was developed with Data Science in mind

- No special server infrastructure needed
- Avoids GitHub repository/file size limit of 2-5GB
- Works with any cloud storage (also on-prem)



- Amazon S3
- Microsoft Azure Blob
- Google Drive
- Google Cloud Storage
- Aliyun OSS
- SSH
- HDFS
- Web HDFS
- HTTP
- WebDAV
- Local remote

image source: dvc.org

Installing DVC

A Python package

- Basic:

  ```
  $ pip install dvc
  ```

  or

  ```
  $ conda install -c conda-forge dvc
  ```

- Additional backends:
  - dvc[all]: all available backends
  - [s3], [azure], [gdrive], [gs], [oss], [ssh]: Individual backends

Setting up repo for data versioning

1. Initialize git repository
   `$ git init`
2. Initialize DVC
   `$ dvc init`
3. Commit the created files to the git repo
   `$ git commit -m "Initialize DVC"`
4. Add the data file for DVC tracking
   `$ dvc add [filename]`
5. Commit the newly created files to the git repo
   `$ git add .gitignore [filename].dvc`
   `$ git commit -m "Initial dataset commit"`
6. Optional: Set up remote storage and push data to it
   `$ dvc remote add [name] [url], $ dvc push`

Setting up repo for data versioning

1. **Initialize git repository**

   `$ git init`

2. Initialize DVC

   `$ dvc init`

3. Commit the created files to the git repo

   `$ git commit -m "Initialize DVC"`

4. Add the data file for DVC tracking

   `$ dvc add [filename]`

5. Commit the newly created files to the git repo

   `$ git add .gitignore [filename].dvc`

   `$ git commit -m "Initial dataset commit"`

6. Optional: Set up remote storage and push data to it

   `$ dvc remote add [name] [url]`

Initializes a git repository and creates

- **.git/**
  Directory used for git tracking

Setting up repo for data versioning

1. Initialize git repository
   `$ git init`
2. **Initialize DVC**
   `$ dvc init`
3. Commit the created files to the git repo
   `$ git commit -m "Initialize DVC"`
4. Add the data file for DVC tracking
   `$ dvc add [filename]`
5. Commit the newly created files to the git repo
   `$ git add .gitignore [filename].dvc`
   `$ git commit -m "Initial dataset commit"`
6. Optional: Set up remote storage and push data to it
   `$ dvc remote add [name] [url]`

Initializes a DVC project with the following files

- **.dvcignore**
  Used to exclude files from DVC tracking (similar to .gitignore)
- **.dvc/.gitignore**
  Used to exclude DVC-internal files from git tracking
- **.dvc/config**
  Configuration file which may be edited manually or with `$ dvc config`
- **.dvc/plots/**
  Directory for the plotting functionality of DVC

Setting up repo for data versioning

1. Initialize git repository

   `$ git init`

2. Initialize DVC

   `$ dvc init`

3. **Commit the created files to the git repo**

   `$ git commit -m “Initialize DVC”`

4. Add the data file for DVC tracking

   `$ dvc add [filename]`

5. Commit the newly created files to the git repo

   `$ git add .gitignore [filename].dvc`

   `$ git commit -m “Initial dataset commit”`

6. Optional: Set up remote storage and push data to it

   `$ dvc remote add [name] [url]`

Commits the (automatically staged) DVC files to the git repository

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   config
        new file:   plots/confusion.json
        new file:   plots/confusion_normalized.json
        new file:   plots/default.json
        new file:   plots/linear.json
        new file:   plots/scatter.json
        new file:   plots/smooth.json
        new file:   ../.dvcignore
```

Setting up repo for data versioning

1. Initialize git repository

   `$ git init`

2. Initialize DVC

   `$ dvc init`

3. Commit the created files to the git repo

   `$ git commit -m "Initialize DVC"`

4. **Add the data file for DVC tracking**

   `$ dvc add [filename]`

5. Commit the newly created files to the git repo

   `$ git add .gitignore [filename].dvc`

   `$ git commit -m "Initial dataset commit"`

6. Optional: Set up remote storage and push data to it

   `$ dvc remote add [name] [url]`

Add data files or directories for DVC tracking (analogous to the `git add` command).

Command creates a new file

- **[filename].dvc**
  Functions as placeholder for the data which is versioned with git

```
outs:
  - md5: a304afb96060aad90176268345e10355
    path: data.xml
    desc: Cats and dogs dataset
    remote: myremote

# Comments and user metadata are supported.
meta:
  name: 'Devee Bird'
  email: devee@dvc.org
```

Setting up repo for data versioning

1. Initialize git repository

```
$ git init
```

2. Initialize DVC

```
$ dvc init
```

3. Commit the created files to the git repo

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

4. Add the data file for DVC tracking

```
$ dvc add [filename]
```

5. **Commit the newly created files to the git repo**

```
$ git add .gitignore [filename].dvc
$ git commit -m "Initial dataset commit"
```

6. Optional: Set up remote storage and push data to it

```
$ dvc remote add [name] [url]
```

Stages the newly created files and commits them to the git repository.

The **.gitignore** was updated when running `dvc add` to exclude that data file from git versioning (because we are versioning the data.dvc file, not the data file itself).

Setting up repo for data versioning

1. Initialize git repository
   `$ git init`
2. Initialize DVC
   `$ dvc init`
3. Commit the created files to the git repo
   `$ git commit -m "Initialize DVC"`
4. Add the data file for DVC tracking
   `$ dvc add [filename]`
5. Commit the newly created files to the git repo
   `$ git add .gitignore [filename].dvc`
   `$ git commit -m "Initial dataset commit"`
6. **Optional: Set up remote storage and push data to it**
   `$ dvc remote add [name] [url], $ dvc push`

Adding remote storage backend.

Backend-specific instructions can be found here:

dvc.org/doc/command-reference/remote/add

Hands-on Exercise: DVC

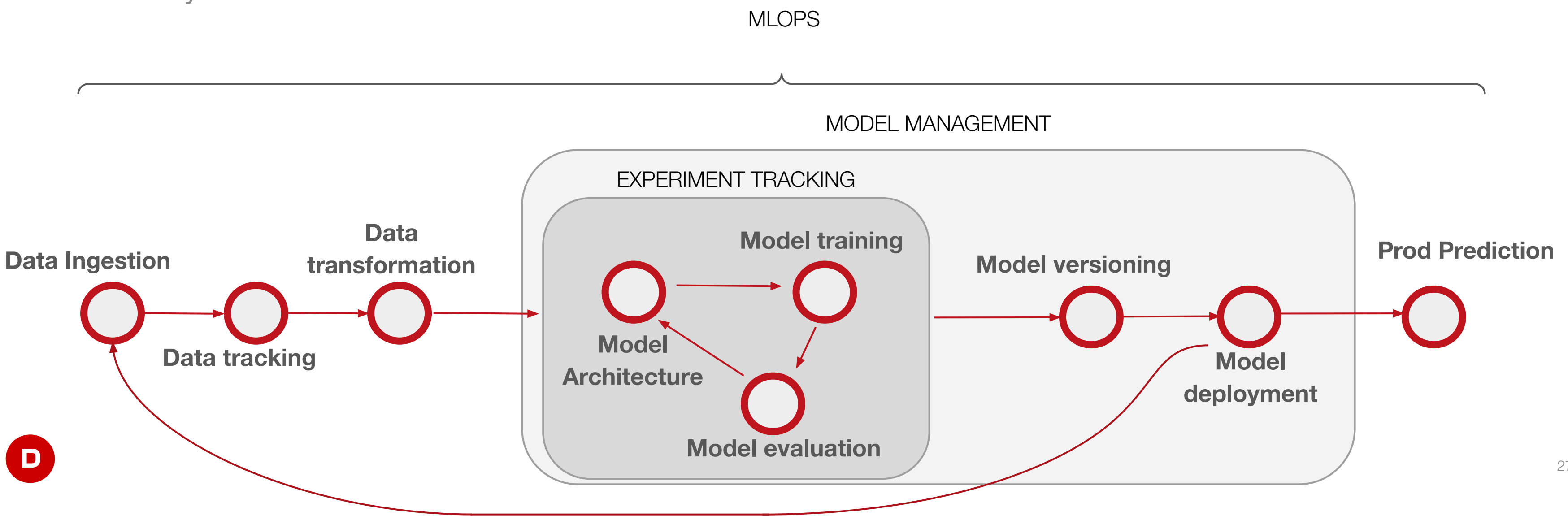
Go to http://localhost:8888/ to access the remote instance of Jupyter Lab, and then navigate to ***/notebooks/*** and open ***dvc-exercise.ipynb*** notebook.

How To:

Experiment Tracking

Why tracking?

- Experiment tracking is the process of saving all experiment related information that you care about for every experiment you run.
- Experiment tracking is **different** from ML model management, as it focuses on the iterative model development phase when you try many things to get your model performance to the level you need.



Machine Learning Platforms

- Big Data Companies:
  - Deal with this with proprietary solutions that standardize data prep/training/deploy loop, so as long as you are in the ecosystem all is good!
  - Tied to the company's infrastructure
  - Out of luck if you want to migrate somewhere else
  - May be limited to certain algorithms or frameworks

- An Open Source Solution is **needed**

Tracking tools (open source)

Tool	*Description*	*creator*	*language*	*Github* ⭐
MLflow	**focus on entire ML lifecycle, works with any ML library**	**Databricks**	**Python**	**11.5k**
DVC	focus on Data Versioning	Iterative	Python	9.5k
Kubeflow	Use with Kubernetes but define tasks in Python, difficult set-up (preferably with GCP)	Google	Python	11.3k
ClearML	complicated installation, unstable	Allegro.AI	Python	3.1k
Guild AI	easy to start with, immature, lightweight, basic interface	Open-source community	Python	678

Machine Learning Platforms

- Big Data Companies:
  - Deal with this with proprietary solutions that standardize data prep/training/deploy loop, so as long as you are in the ecosystem all is good!
  - Tied to the company's infrastructure
  - Out of luck if you want to migrate somewhere else
  - May be limited to certain algorithms or frameworks

- An Open Source Solution is **needed**

mlflow

Why MLflow

- Open source, not tied to a particular platform/company
- Runs the same way everywhere (locally or in the cloud)
- Useful from 1 developer to 100+ developers

- Design philosophy:
  1. API-First
  2. Integration with popular libraries
  3. high level “model” function that can be deployed everywhere
  4. Open interface that enables contributions from the community
  5. Modular design (can use DISTINCT components separately)

Installing MLflow

A Python package

- Basic:

  `$ pip install mlflow`

  or

  `$ conda install -c conda-forge mlflow`

- Additional:

  `$ pip install mlflow-skinny` (lower dependency subset)

MLflow Components

MLflow Tracking

Record and query experiments: code, data, config, and results

Read more

MLflow Projects

Package data science code in a format to reproduce runs on any platform

!! not used

Read more

MLflow Models

Deploy machine learning models in diverse serving environments

Read more

Model Registry

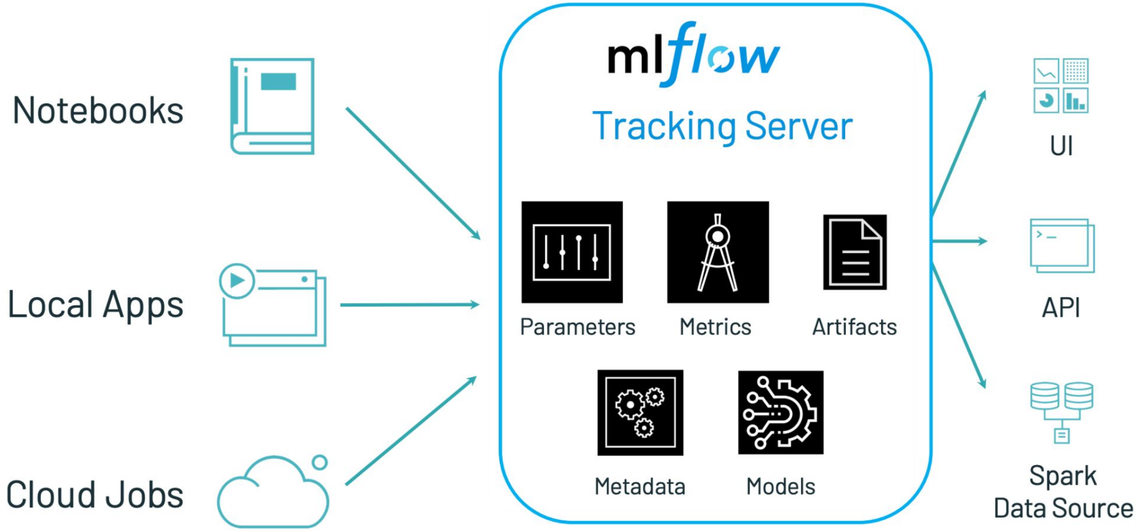
Store, annotate, discover, and manage models in a central repository

Read more

Useful links:

- www.mlflow.org
- www.github.com/mlflow
- www.databricks.com/mlflow

MLflow Tracking



set tracking location

```
$ export_MLFLOW_TRACKING_URI <URI>
mlflow.set_tracking_uri()
```

URI: Uniform Resource Identifier

MLflow Tracking

What do we track?

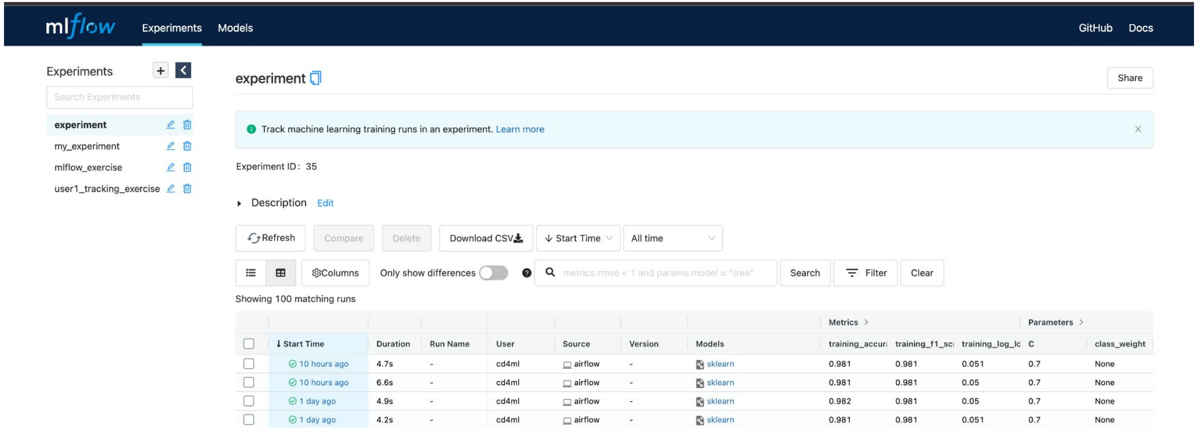
- Parameters : inputs to our code `mlflow.log_param()..`
- Metrics : numeric values to access our models `mlflow.log_metric()..`
- Tags/Notes: info about the run `mlflow.set_tag()..`
- Artifacts: files,data and models produced `mlflow.log_artifact(), mlflow.log_artifacts()..`
- Source: what code run
- Version: what version of the code run (github)
- Run: the particular code instance (id) captured by MLflow `mlflow.start_run()..`
- Experiment: the set of runs `mlflow.create_experiment(), mlflow.set_experiment()..`

More on : https://www.mlflow.org/docs/latest/tracking.html

MLflow UI

MLflow UI is used to compare the models that you have produced

- connect to the MLflow tracking server where you set `mlflow.set_tracking_uri()`
- or connect to local MLflow tracking server instance by running the CLI command `mlflow ui` in the same working directory as the one that contains the `mlruns` folder



URI: Uniform Resource Identifier
UI: User Interface

MLflow Model

- Standard format for packaging machine learning models in MLflow
- Defines a convention that lets you save a model in different “flavors” that can be understood by different downstream tools

```
# Directory written by
mlflow.sklearn.save_model(model,
"my_model")

my_model/
├── MLmodel
├── model.pkl
├── conda.yaml
└── requirements.txt
```

```
# in MLmodel file

time_created:
2021-10-25T17:28:53.35

flavors:
  sklearn:
    sklearn_version: 0.24.1
    pickled_model: model.pkl
  python_function:
    loader_module: mlflow.sklearn
```

python_function model flavor

- `python_function` is the default model interface for MLflow Python
- It:
  1. allows any Python model to be productionized in a variety of environments
  2. has generic filesystem model format
  3. is self-contained (includes all the information necessary to load and use a model)

```
# LOG
from mlflow import pyfunc
# Log model as artifact
mlflow.pyfunc.log_model(param)
# or Save model in storage
mlflow.pyfunc.save_model((param)
```

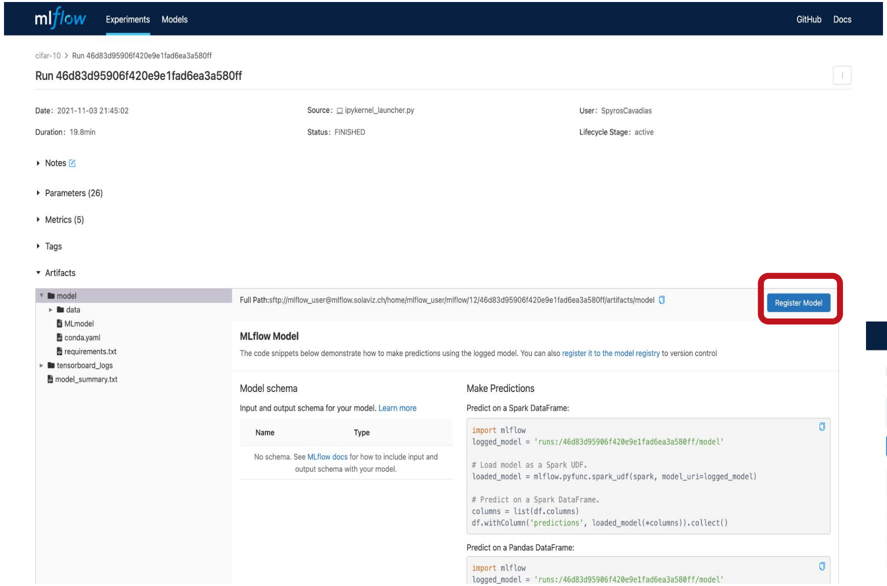
```
# LOAD
from mlflow import pyfunc
# load model
pyfunc_model = pyfunc.load_model(param)
# predict on model
y_pred = pyfunc_model.predict(y_test)
```

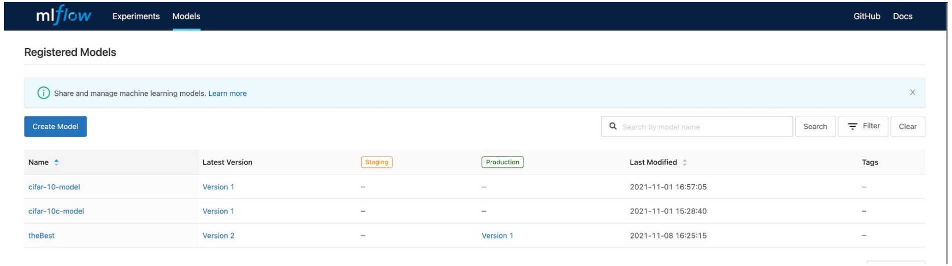
MLflow Model Registry

- It is a centralized model store, set of APIs, and UI, to collaboratively manage the full lifecycle of an MLflow Model.
- Provides **model lineage** (which MLflow experiment and run produced the model), **model versioning**, **stage transitions** (for example from staging to production), and **annotations**.
- You register a model through:
  1. API Workflow
  2. UI Workflow

```
# register model
res = mlflow.register_model(my_model_uri, "my_model")
```

MLflow Model Registry





Hands-on Exercise: Tracking

Go to http://localhost:8888/ to access the remote instance of Jupyter Lab, and then navigate to ***/cd4ml-workshop/notebooks/*** and open ***mlflow-exerise.ipynb*** notebook.

How To:

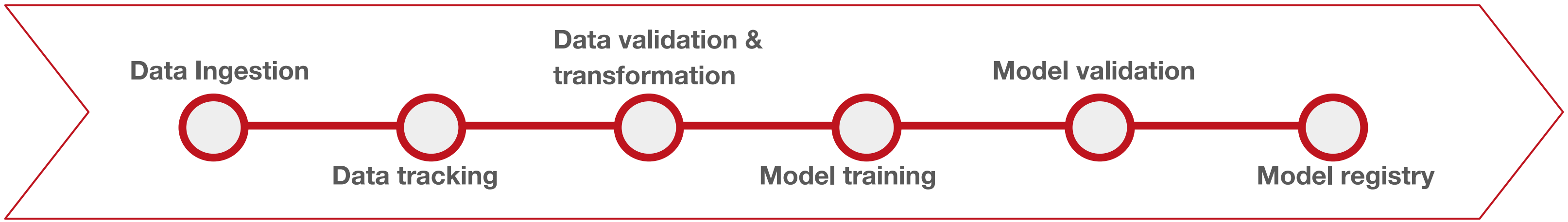
Orchestration

How do we bring everything together?

- Up to this point we have built the core components of our ML training pipeline
- Now we want to combine these components to run the pipeline end-to-end
- This could be done in a simple bash/Python script that runs all the components sequentially..
- However, we would like to:
  - Have a scheduler that is aware how the components interact and when to run each component as the pipeline grows in complexity
  - Be able to re-start the pipeline from a certain component after a crash
  - Visually inspect and monitor our pipeline

Meet pipeline orchestration

- To combine the components and add the desired functionalities, we need a **pipeline orchestrator**, which:
  - Helps to automate and execute workflows
  - Schedules different components and coordinates dependencies among them
  - Abstracts away details of computing environment and provides a UI to monitor and interact with pipeline runs



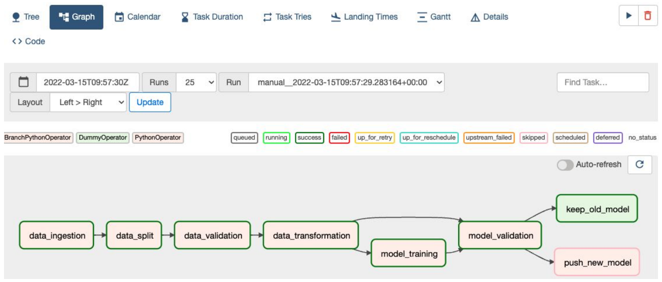
Orchestration tools

Tool	*Description*	*first release*	*creator*	*language*	*Github* ⭐
Luigi	Easy to get started, low complexity	2015	Spotify	Python	15.5k
Apache Airflow	**Includes UI, feature-rich, mature, complex**	**2016**	**Airbnb**	**Python**	**25.1k**
Kubeflow	Use with Kubernetes but define tasks in Python, difficult set-up (preferably with GCP)	2018	Google	Python	11.3k
Argo Workflows	Use with Kubernetes (Alternative to Kubeflow), define tasks with YAML	2017	Open-source community	YAML	10.6k
Prefect	Easy to get started for Python programmers, immature, open-core	2019	Prefect	Python	8.5k

→ We chose Airflow due to its maturity, popularity (support of open source community), flexibility, and easy to use UI

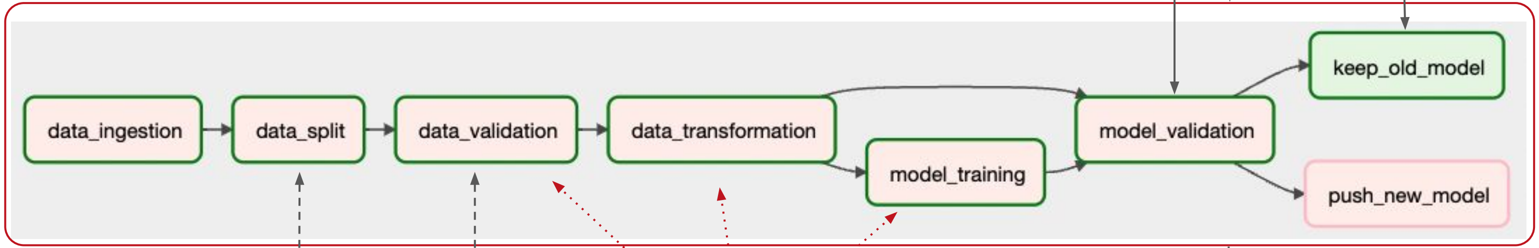
Airflow *concepts* and ***components***

Airflow Webserver
- Hosts Airflow UI (Flask App)
- Monitor runs, investigate logs, etc.



DAG (ML Pipeline)
- Built with Operators
- Triggered:
  - automatically by *Airflow Scheduler*, or
  - manually from Airflow UI

Airflow Scheduler
- Multithreaded Python process
- Schedules workflows and tasks



Metadata database *(SQLAlchemy)*
- Tracks how components interact
- All process read from / write to it

Operators (Tasks)
- Defined in Python, Bash, SQL, etc.
- Run on *Executor*

Executor (specified in airflow.cfg)
- local executors: run tasks inside scheduler
- remote executors: run tasks remotely (e.g. Kubernetes)

How to set up Airflow

- With **Docker** - convenient to manage dependencies & isolate Airflow components (documentation)
- With **Docker-Compose**:
  - Easy to get started (documentation), but requires Docker Compose expertise for customized & production ready deployment
- With **Helm Chart** package manager (documentation)
  - Docker-based deployment using Kubernetes, supported by Airflow community
- **Locally** in a virtual environment (documentation)
  - Set AIRFLOW_HOME environment variable
  - Install Airflow using PyPI
  - Initialize the metadata database airflow initdb
  - Start the Airflow scheduler
  - Start the Airflow webserver

```
export AIRFLOW_HOME=~/airflow
pip install apache-airflow
airflow db init
airflow webserver --port 8080
airflow scheduler
```

How to create an Airflow DAG

- Create a Python file for the Airflow DAG, including:
  1. Define an Airflow DAG object with configuration settings
  2. Define Airflow operators for the individual tasks. Airflow offers pre-defined operators such as:
     - Python operators: calling a Python function
     - Bash operators: executing a bash command
     - SQL operators, MySQL operators, etc.
  3. Set instructions on how to connect the various Airflow operators
- Save the DAG python file in ~/AIRFLOW_HOME/dags

How to create an Airflow DAG

- Create a Python file for the Airflow DAG, including:
  1. Define an Airflow DAG object with configuration settings
  2. Define Airflow operators for the individual tasks. Airflow offers pre-defined operators such as:
     - Python operators: calling a Python function
     - Bash operators: executing a bash command
     - SQL operators, MySQL operators, etc.
  3. Set instructions on how to connect the various Airflow operators
- Save the DAG python file in ~/AIRFLOW_HOME/dags

```python
dag = DAG(
    'example_dag',
    default_args=default_args,
    description='example dag',
    schedule_interval=timedelta(days=1),
)
```

How to create an Airflow DAG

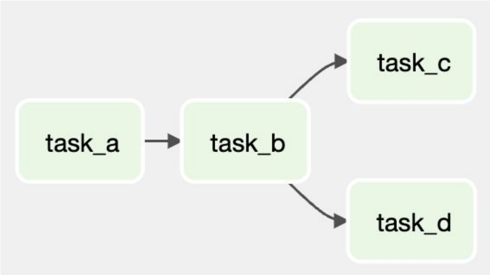
- Create a Python file for the Airflow DAG, including:
  1. Define an Airflow DAG object with configuration settings
  2. **Define Airflow operators for the individual tasks. Airflow offers pre-defined operators such as:**
     - Python operators: calling a Python function
     - Bash operators: executing a bash command
     - SQL operators, MySQL operators, etc.
  3. Set instructions on how to connect the various Airflow operators
- Save the DAG python file in ~/AIRFLOW_HOME/dags

```python
dag = DAG(
    'example_dag',
    default_args=default_args,
    description='example dag',
    schedule_interval=timedelta(days=1),
)

with dag:
    data_ingestion = PythonOperator(
        task_id='task_a',
        python_callable=ingest_data,
    )
    # ...
```

How to create an Airflow DAG

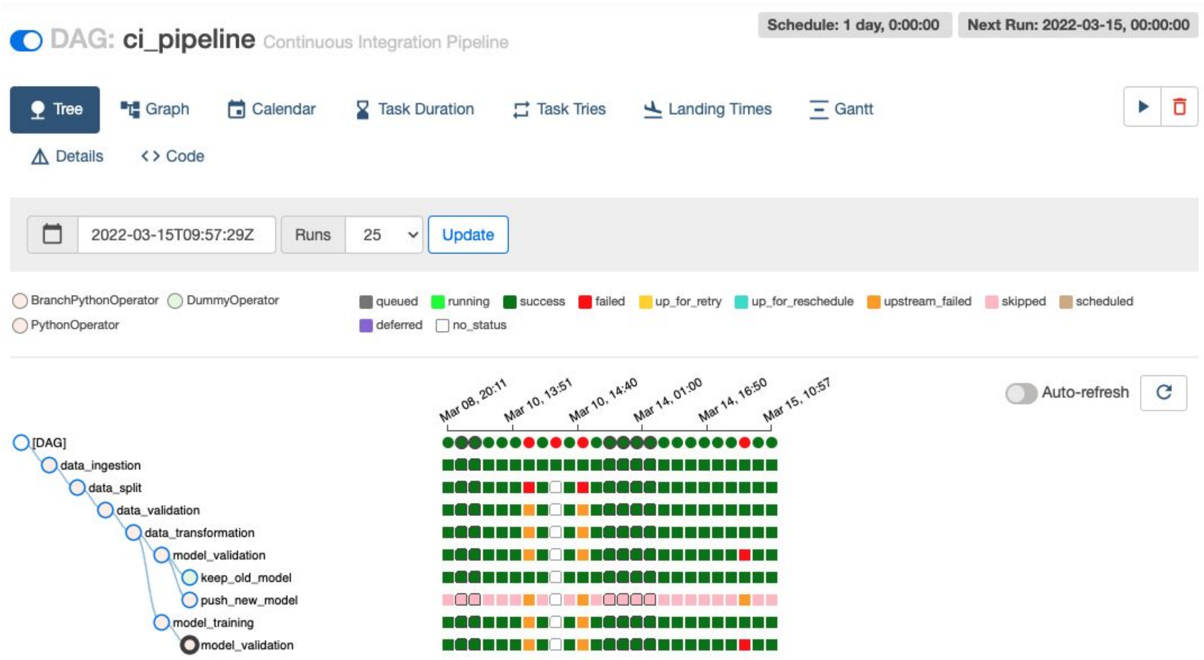
- Create a Python file for the Airflow DAG, including:
  1. Define an Airflow DAG object with configuration settings
  2. Define Airflow operators for the individual tasks. Airflow offers pre-defined operators such as:
     - Python operators: calling a Python function
     - Bash operators: executing a bash command
     - SQL operators, MySQL operators, etc.
  3. **Set instructions on how to connect the various Airflow operators**
- Save the DAG python file in ~/AIRFLOW_HOME/dags

```python
dag = DAG(
    'example_dag',
    default_args=default_args,
    description='example dag',
    schedule_interval=timedelta(days=1),
)

with dag:
    data_ingestion = PythonOperator(
        task_id='task_a',
        python_callable=ingest_data,
    )
    # ...

    task_a >> task_b >> [task_c, task_d]
```



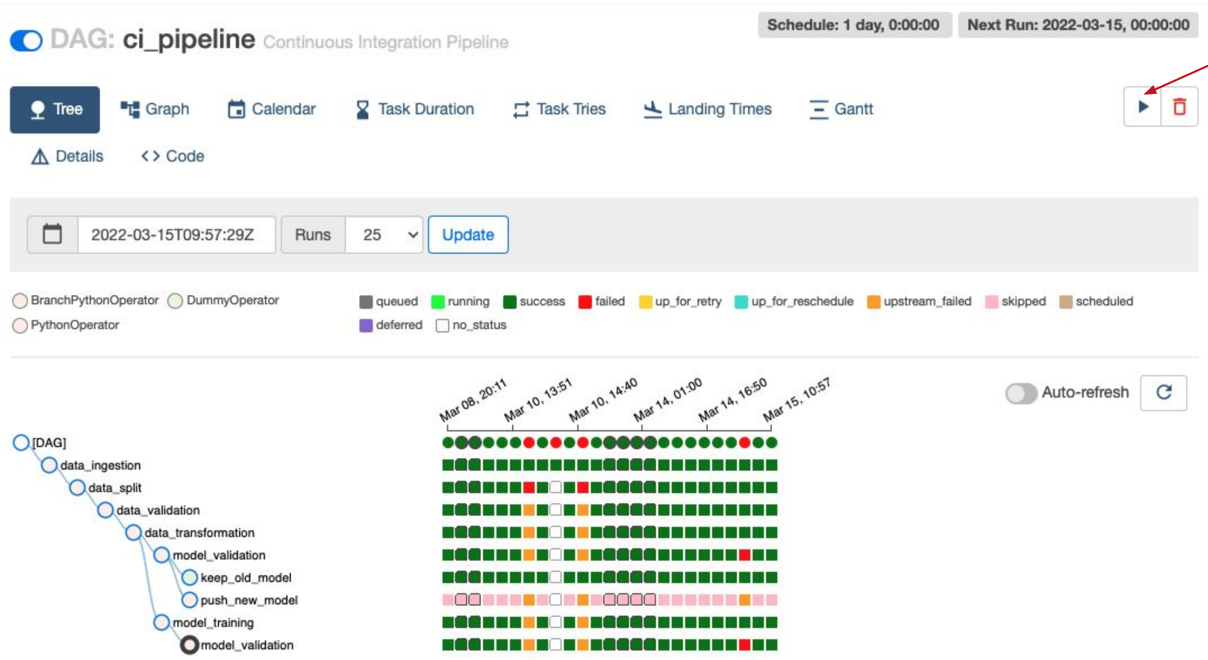
Interact with the Airflow UI

- The best way to get familiar with the UI is to experiment yourself…



Interact with the Airflow UI

- The best way to get familiar with the UI is to experiment yourself…

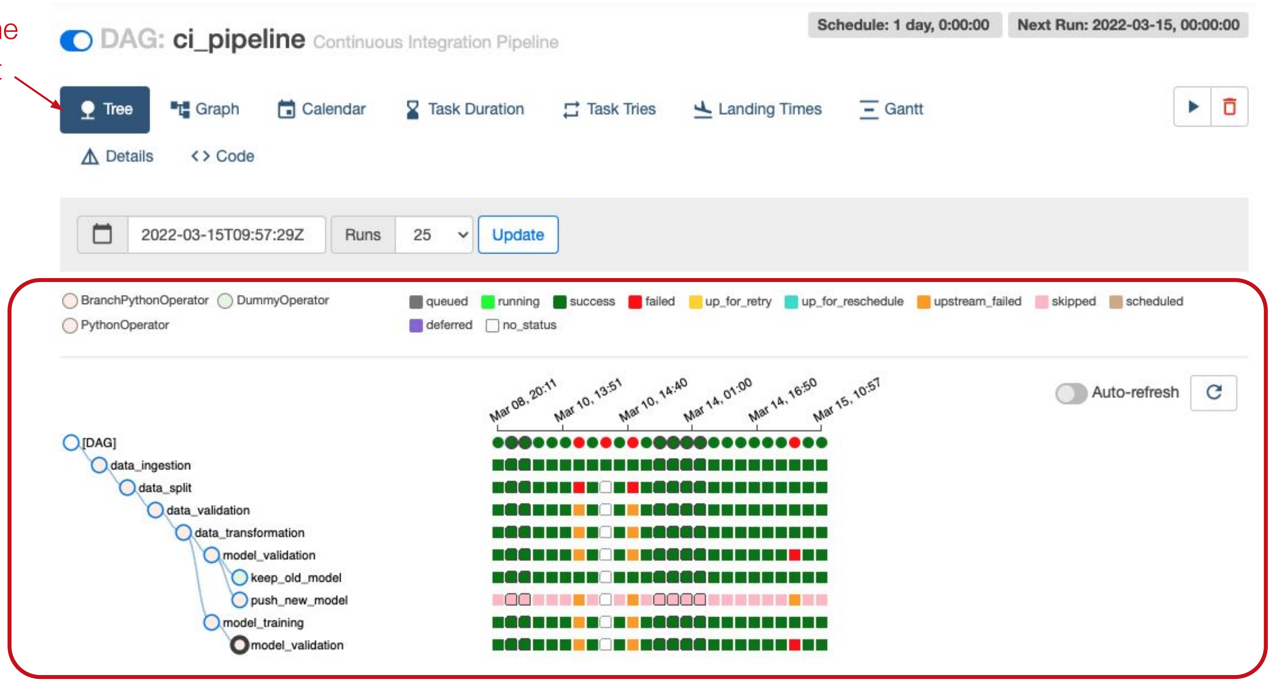


Trigger a DAG run

Interact with the Airflow UI

- The best way to get familiar with the UI is to experiment yourself…

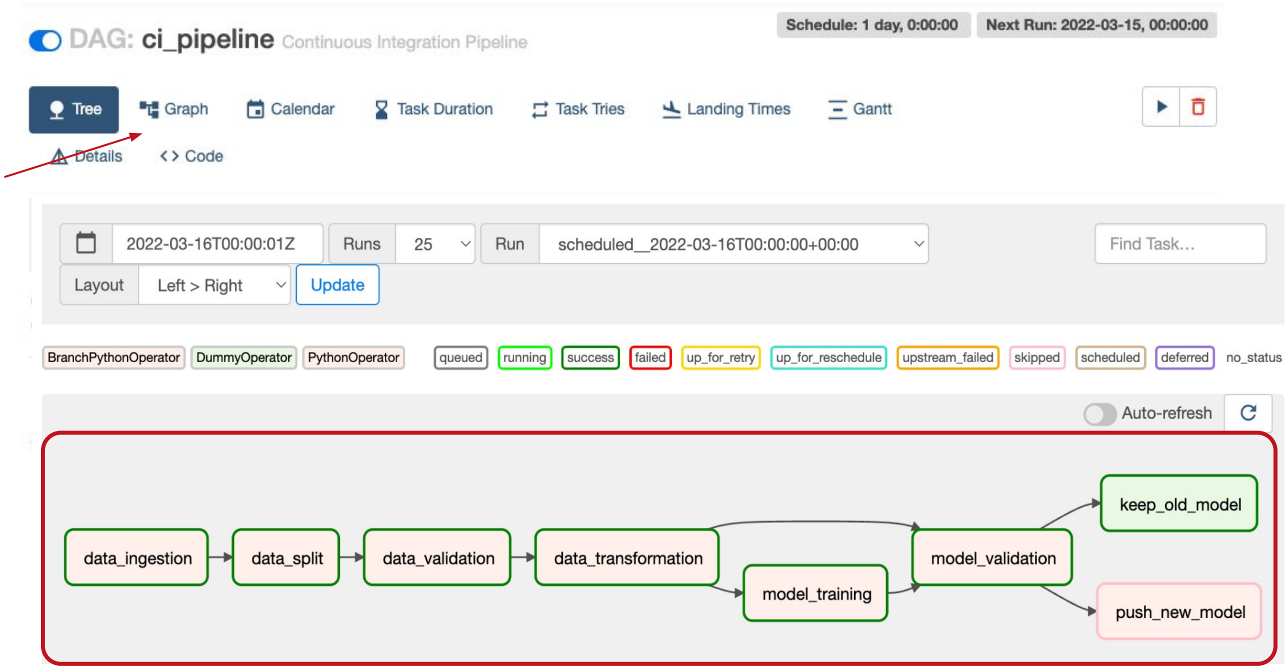
Monitor the pipeline workflow and past runs



Interact with the Airflow UI

- The best way to get familiar with the UI is to experiment yourself…

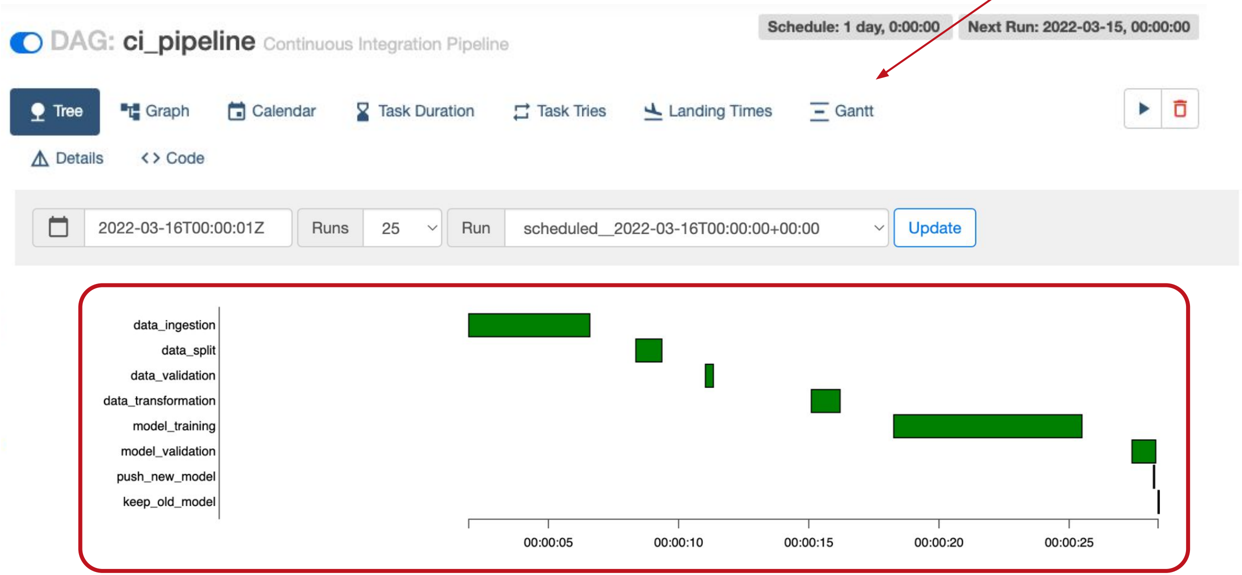
Inspect the DAG graph in its operators



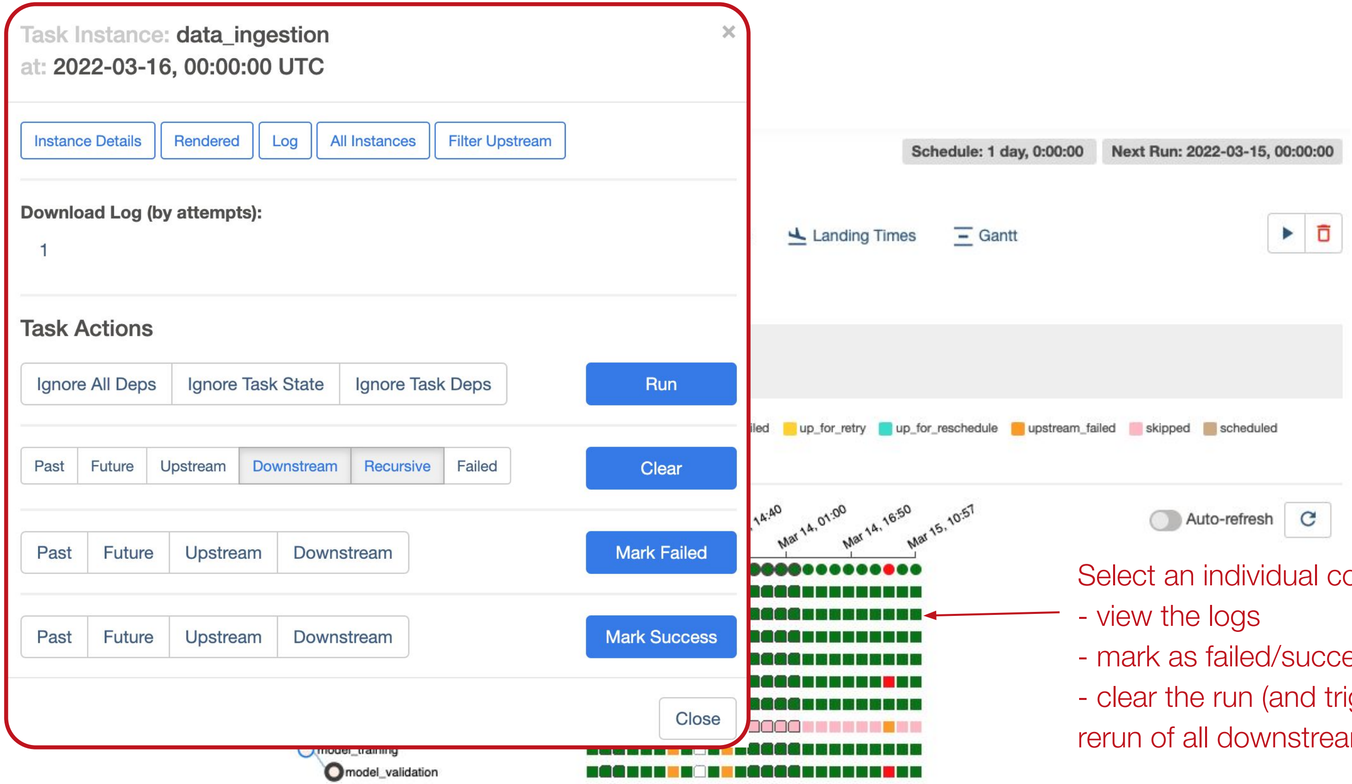
Interact with the Airflow UI

- The best way to get familiar with the UI is to experiment yourself…

Monitor running times



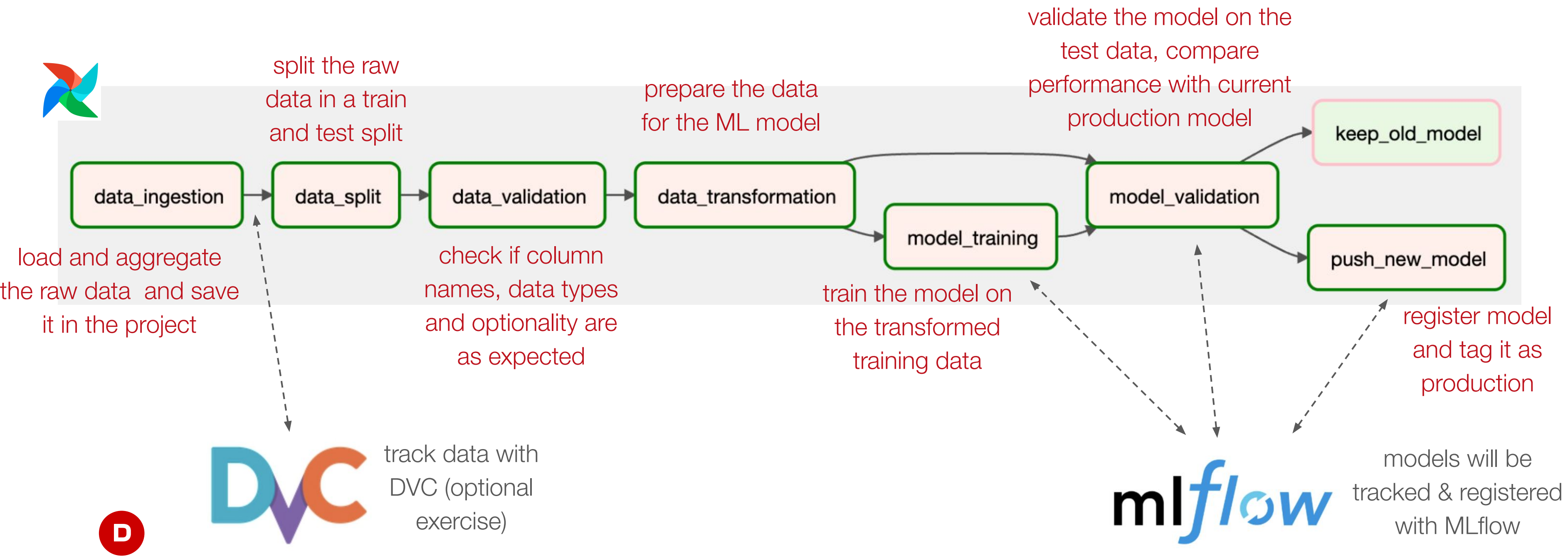
Interact with the Airflow UI



Select an individual component, to
- view the logs
- mark as failed/success
- clear the run (and trigger automatic rerun of all downstream components)

Now it is your turn!

- In the next exercise you will run the complete ML training pipeline in Airflow
- For reference, here is a summary what the individual components do:



Hands-on Exercise: Pipeline

Follow the instructions of the pipeline tasks in the **handout.md**

[EXTRA] How To:

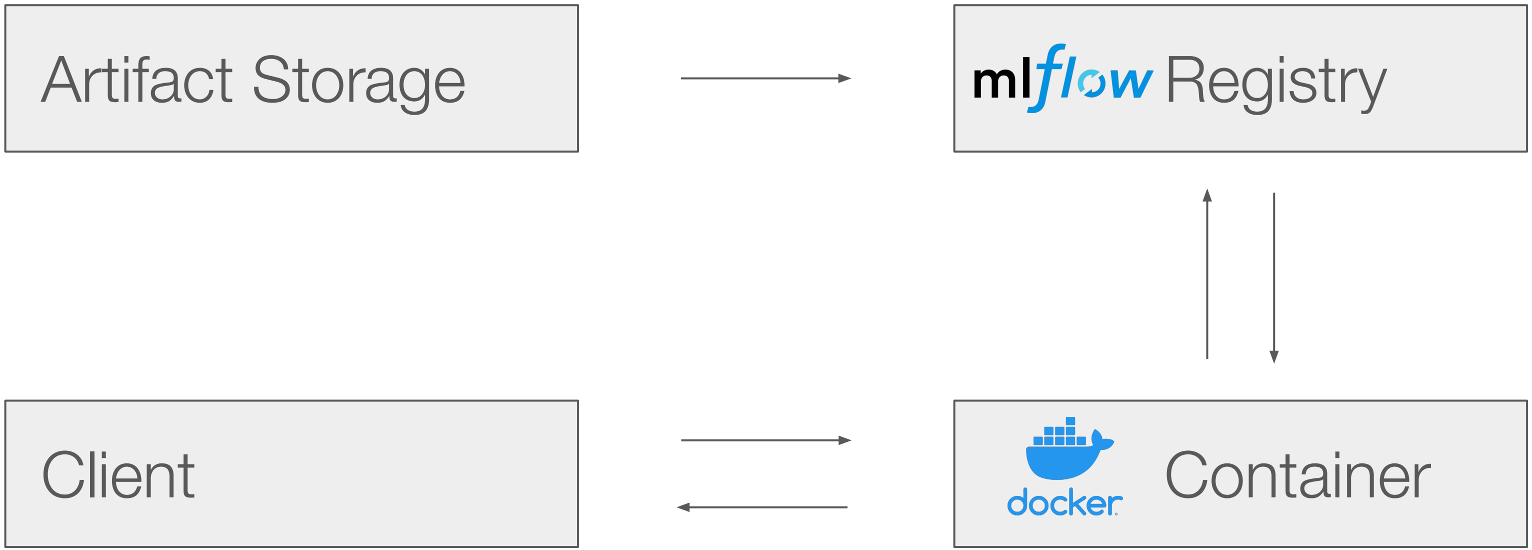
Deploy

Deployment

Batch Prediction

Online Prediction

Deploy a Web API



`run_id: 2deba13c50a84a9699acb15fda196afd`

Deployment with Docker and Flask

```
docker_build_context
  score
    __init__.py
    score.py
  app.py
  Dockerfile
  requirements.txt
__init__.py
launch_api_endpoint.py
```

score.py	contains the actual scoring (inference) logic and two functions `init()` and `run(input)`.
app.py	Launches the API in a Flask Web application.
Dockerfile	All the configuration for the build of the Docker image.

Inference

```python
import requests
import numpy as np
import pandas as pd

host = "http://localhost:5000"

dat = pd.DataFrame(np.random.uniform(0, 1, (10, 12)),
                   columns=[
    "wind_speed",
    "power",
    "nacelle_direction",
    "wind_direction",
    "rotor_speed",
    "generator_speed",
    "temp_environment",
    "temp_hydraulic_oil",
    "temp_gear_bearing",
    "cosphi",
    "blade_angle_avg",
    "hydraulic_pressure"
]
)

data = {"data": dat.to_json()}

print(requests.post(host, json=data).json())
```

Sending Requests to the Rest API

RECAP

Recap

- We presented you with a real-world case how to bring ML to production
- You learned, how to:
  - Track your data
  - Track your models and experiments
  - Orchestrate your ML pipeline
  - Deploy
- We presented just one specific MLOps-approach – There are many different tools and practices to achieve the same goal
- We hope we provided you with some valuable practices and inspiration to put your own ML project to production

If you are curious about what we are doing at D ONE and would like to stay in touch…