

Einführung in MLOps

08 MODEL PLATTFORMEN, REGISTRY, MLFLOW

Tobias Mérinat

teaching2025@fsck.ch

Lucerne University of
Applied Sciences and Arts

**HOCHSCHULE
LUZERN**

DEPARTMENT OF INFORMATION TECHNOLOGY
Lucerne University of Applied Sciences and Arts
6343 Rotkreuz, Switzerland

14. und 15. Februar 2025

Das Konzept *Model-Plattform* beschreibt, wie Machine Learning Modelle paketiert, gespeichert, verwaltet und verwendet werden. Eine Model Plattform besteht aus drei Hauptkomponenten:

- dem Model Deployment API
- der Model Registry und
- dem Model Server (auch Prediction Server/Service)

Das Tracken der Experimente mit verschiedenen Modellen und Hyperparametern während der Entwicklungsphase kann als vierter Teil der Modellplattform betrachtet werden.

Experiment Tracking dient einerseits der Chaos-Vermeidung während der Entwicklung, und hat andererseits die Reproduzierbarkeit der schliesslich verwendeten Modelle zum Ziel.

Das Deployment API Beschreibt, wie fertig trainierte Modelle paketiert werden. Modelle müssen inkl. trainierter Parameter serialisiert werden, sie müssen versioniert sein sowie alle Dependencies nennen oder mit pakettieren.

Das gleich vorgestellte Open Source Tool **MLFlow** definiert ein solches Deployment API. **ONNX** (Open Neural Network Exchange) ist ein Open Source Format für Neuronale Netze.

Die Aufgabe der *Model Registry* (auch als *Model Store* bezeichnet) ist es, Machine Learning Modelle versioniert zu speichern, zu verwalten und für die Prediction zur Verfügung zu stellen.

Neben den Modellen (Artefakten) speichert die Registry Metadaten wie die im Training verwendeten Parameter, Dependencies und Performancemetriken.

Die Registry ist eine zentrale Stelle, um Nachvollziehbarkeit, Reproduzierbarkeit, und automatisiertes Deployment zu ermöglichen.

Eine Model Registry besteht aus den folgenden Komponenten:

- Ein Artefakt Store, im allgemeinen ein Object Store um Modelle zu speichern
- Eine Datenbank für die Speicherung von Metadaten
- Ein GUI für die Verwaltung von Modellen
- Ein API für den Bezug von Modellen und Metadaten

Eine Model Registry sollte die folgenden Informationen speichern:

- Die komplette Environment (Dependencies und Python Version)
- Eine Referenz auf die Trainingsdaten und Commit Hashes des gesamten Feature Engineering und Training Codes
- Hyperparameter
- Performance-Metriken
- Das ausführbare Modell-Artefakt, zwecks deployment

- Experiment Tracking wird verwendet, um
 - die Übersicht während des Entwicklungsprozesses zu behalten
 - Experimente bzw. die entstandenen Modelle einfach vergleichen zu können
 - die relevanten Modelle aus der Entwicklungsphase reproduzieren zu können
- Die Model Registry wird benötigt,
 - um Modelle reproduzierbar in den produktiven Betrieb zu übergeben
 - Modelle nach Bedarf zu annotieren
 - den Model Lifecycle abzubilden

Der *Model Server* (auch *Prediction-Server*, *Inference-Server*) ist dafür verantwortlich, Modelle aus der Registry zu holen, entsprechend Ressourcen zu allozieren und einen REST oder gRPC Endpoint für Prediction-Anfragen zur Verfügung zu stellen.

Das Angebot an Model Servern ist breit. Einige bekanntere Model Server sind **MLFlow**, **Triton Inference Server**, **Seldon ML Server**, **Kserve**, **BentoML**, **TensorFlow Serving**, **TorchServe**, **Multi Model Server** und **OpenVINO Model Server**.

Model Server können auf spezifische Architekturen spezifiziert sein (z.B. Tensorflow Serving, TorchServe) oder mehrere Model Deployment APIs unterstützen (Triton, MLFlow, Seldon, KServe).

Machine Learning Modelle können auf verschiedene Arten verwendet bzw. deployt werden:

- Mehrere Modelle/Modellversionen in einem zentralen Inference Server
- Jedes Modell zusammen mit einem REST/gRPC API in einem eigenen Container
- Direkt in eine Fachapplikation integriert
- In eine Batch- oder Streaming Pipeline integriert

Jede Variante hat Vor- und Nachteile.

Es gibt bei der Wahl der Deployment-Variante einiges zu Bedenken:

- Entkoppelung von Modell und Applikation
- Wie einfach ist es, ein Modell zu aktualisieren
- Können mehrere Versionen eines Modells gleichzeitig verwendet werden
- Ist für einen Anwendungsfall besser, ein Modell in den Speicher zu laden oder via ein API auszuführen?
- Wenn ein Modell in einen Container gekapselt wird, soll das Modell in den Container oder nachgeladen werden?

Im Rahmen des Kurses wird die OpenSource Software **MLFlow** verwendet. MLFlow deckt einige Anforderungen von MLOps ab und erlaubt es uns, viel selbst auszuprobieren.

MLFlow ist jedoch bei weitem nicht das einzige Tool und muss auch nicht die erste Wahl sein für eine *Model Plattform*. Es ist opinionated, die Dokumentation ist zwar umfangreich, aber auch recht chaotisch, und die Funktionalität von MLFlow lässt teilweise zu wünschen übrig.

Es geht im Kurs nicht darum, eine tiefe Einführung in MLFlow als Tool zu geben, sondern darum, die Konzepte, welche MLOps mehr oder weniger gut umsetzt, zu vermitteln.

MLFlow besteht aus einer Reihe von Komponenten. Wir verwenden die folgenden drei:

- **Tracking:** Ein API um Informationen, welche während des Trainings von Modellen anfallen, zu loggen und zu visualisieren
- **Models:** Ein Standard-Format, um Machine Learning Modelle zu paketieren
- **Model Registry:** Ein Dienst, um Modelle zu speichern, zugreifbar zu machen und ihren Lifecycle zu managen

MLFlow bietet die folgenden weitere Komponenten, welche wir hier nicht verwenden

- **Projects:** Konventionen, um die Reproduzierbarkeit von ML-Projekten zu verbessern
- **Recipes:** Eine Art Framework mit Templates, um rasch typische Standard-ML-Modelle entwickeln zu können
- **Dataset Tracking:** Ein Submodul von *Tracking*, um Datensätze zu loggen
- **LLM:** Funktionalität, um mit Cloud LLM APIs zu interagieren
- **Model Serving:** Einen Weg, um ein ML Modell und ein REST Api in einen Container zu verpacken.

MLFlow definiert ein **Paketformat** für ML Modelle, um deren Verwendung für die Inferenz zu vereinfachen. Es umfasst

- Eine Folder-Struktur als Speicherformat
- ein Metadatenfile namens *MLModel*
- Eine **Liste der Dependencies**

Folder-Struktur, wie sie `mlflow.sklearn.save_model()` schreibt:

```
my_model/  
|-- MLmodel  
|-- model.pkl  
|-- conda.yaml  
|-- python_env.yaml  
__ requirements.txt
```


Ein *MLModel* File, welches ein Modell in zwei Ausprägungen (*flavors* in MLFlow Sprache) definiert:

```
time_created: 2018-05-25T17:28:53.35
```

```
flavors:
```

```
  sklearn:
```

```
    sklearn_version: 0.19.1
```

```
    pickled_model: model.pkl
```

```
  python_function:
```

```
    loader_module: mlflow.sklearn
```

Es werden **viele Model Flavors** von MLFlow unterstützt (Sklearn, PyTorch, Keras, ONNX, XGBoost, Spacy, . . .). Auch ein generischer 'pyfunc' Flavor steht zur Verfügung.

Beim Loggen eines Modells kann eine *Model Signature* und/oder ein *Input Example* übergeben werden. Signaturen definieren Model Input, Output und Parameter für die Inferenz.

So können entsprechende Schemata forciert werden, was einerseits das Modell besser dokumentiert und dessen Verwendung einfacher macht und andererseits die Erkennung von Datenanomalien im Betrieb vereinfacht.

MLFlow Model Signaturen sind nur eine Möglichkeit für solche Datenchecks, mehr dazu kommt später im Kapitel *Data Validation*.

MLFlow Tracking besteht aus den folgenden Komponenten:

- **Tracking Server:**

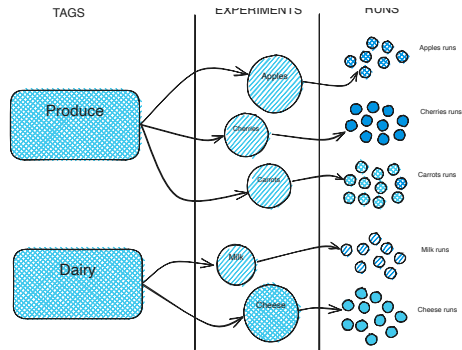
- REST API
- Web UI

- **Metadata Store:** File-Based oder SQL Database

- **Artifact Store:** Für die Ablage grosser Datenobjekte (File, S3)

Experimente enthalten *Runs*, welche wiederum *Child-Runs* enthalten können. Weitere Gliederung ist über frei vergebbare *Tags* möglich.

- **Experiment:** Alles, was die gleichen Input-Daten verwendet
- **Run:** Ein Trainings-Durchlauf. Die Idee ist, dass verschiedene Runs vergleichbar bleiben sollen.
- **Child-Run:** Für Hyperparameter-Sweeps verwendet



MLFlow Tracking: Logging Setup (1)

Um das MLFlow-API verwenden zu können, muss die Tracking-URI gesetzt werden oder die Umgebungsvariable `MLFLOW_TRACKING_URI` gesetzt sein:

```
mlflow.set_tracking_uri("http://127.0.0.1:8080")
```

Ist die **Authentifizierung** aktiviert, müssen zudem die Umgebungsvariablen `MLFLOW_TRACKING_USERNAME` und `MLFLOW_TRACKING_PASSWORD` gesetzt oder diese Werte via ein Credentials File hinterlegt sein.

MLFlow Tracking: Experiment festlegen (2)

Als nächstes sollte ein Experimentname gesetzt werden. Der Rückgabewert enthält Metadaten zum Experiment, man muss diesen nicht unbedingt speichern.

```
experiment_metadata = mlflow.set_experiment("Apple_Models")
```

Wird dieser Schritt ausgelassen, weist MLFlow die nachfolgenden Runs einem *Default-Experiment* zu. Das Default-Experiment sollte man nicht verwenden, es dient lediglich als Catch-All, damit keine Runs verloren gehen, wenn das Experiment nicht gesetzt wurde.

MLFlow Tracking: Training durchführen) (3)

Nun wird ein Modell trainiert, hier im Beispiel mit Scikit-Learn. Bei diesem Schritt gibt es keinen MLFlow-spezifischen Code. Die Vorbereitung der Datensets wird her nicht gezeigt.

```
1 params = {  
2     "n_estimators": 100,  
3     "max_depth": 3,  
4 }  
5 rf = RandomForestClassifier(**params)  
6 rf.fit(X_train, y_train)  
7 y_pred = rf.predict(X_val)  
8 metrics = { "acc": accuracy_score(y_val, y_pred) }
```

Die Modell-Signatur kann aus den Daten abgeleitet werden:

```
9  from mlflow.models import infer_signature
10 signature = infer_signature(model_input=X_train,
11                             model_output=y_train)
```


MLFlow Tracking: Manuelles loggen (5)

Und schliesslich werden die Resultate an den Tracking Server geschickt.

```
11 with mlflow.start_run(run_name="some run-specific name") as run:
12     mlflow.log_params(params)
13     mlflow.log_metrics(metrics)
14     mlflow.sklearn.log_model(
15         sk_model=rf, signature=signature,
16         artifact_path="relative/to/artifact/root"
17     )
```

MLFlow Tracking: Automatisches loggen

Für eine Vielzahl an ML Bibliotheken (Sklearn, PyTorch, LightGBM, . . .) kann auch automatisch geloggt werden.

```
mlflow.set_experiment("Apple_Models")
mlflow.autolog()  # must be activated BEFORE metrics import
from sklearn.metrics import accuracy_score
rf = RandomForestClassifier(n_estimators=100, max_depth=3)
# ONLY logs fit(), fit_transform(), fit_predict()
# and not other functions like crossvalscore()
rf.fit(X_train, y_train)
```

Dabei werden Metriken, Parameter, Artefakte und die Model Signature automatisch geloggt.

MLflow Tracking: UI Übersicht

mlflow 2.14.1 Experiments Models 🔌 ⚙️ [GitHub](#) [Docs](#)

Experiments

☐

 Default ✎ 🗑

☒ MLflow Example ✎ 🗑

MLflow Example

ℹ️ [Provide Feedback](#) [↗](#) [Add Description](#) [Share](#)

ℹ️ ⋮ 🔄 [+ New run](#)

Time created ▾

State: Active ▾

Datasets ▾

⚙ Sort: Created ▾

📄 Columns ▾

📊 Group by ▾

Table

Chart

Evaluation

Experimental

Traces

Experimental

<input type="checkbox"/>	Run Name	Created	Dataset
<input type="checkbox"/>	🔴 upset-bug-224	✅ 26 seconds ago	-

MLFlow Tracking: Run View (1/2)

[MLflow Example](#) >

upset-bug-224

⋮ [Model registered](#) [↗](#)

Overview

[Model metrics](#)

[System metrics](#)

[Artifacts](#)





Description [✎](#)

No description


Details

Created at	2024-07-25 16:36:36
Created by	root
Experiment ID	1 ↗
Status	✅ Finished
Run ID	7180c89f925946b0b03338663640810d ↗
Duration	4.1s
Datasets used	—
Tags	Training Info: Basic RF classifier model for iris ... ✎
Source	🏠 ipykernel_launcher.py


MLFlow Tracking: Run View (2/2)

Tags	Training Info: Basic RF classifier model for iris ... 
Source	 ipykernel_launcher.py
Logged models	 sklearn
Registered models	 tracking-quickstart v1

Parameters (2)

 Search parameters	
Parameter	Value
n_estimators	100
max_depth	3

Metrics (1)

 Search metrics	
Metric	Value
acc	1

MLFlow Tracking: Zweiter Run

mlflow 2.14.1 Experiments Models 🔍 ⚙️ [GitHub](#) [Docs](#)

Experiments

☐

 Default ✎️ 🗑️

☒ MLflow Example ✎️ 🗑️

MLflow Example

ℹ️ [Provide Feedback](#) [🔗](#) [Share](#)

ℹ️ ⋮ 🔄 [+ New run](#)

Time created ▾

State: Active ▾

Datasets ▾

⌵ Sort: Created ▾

📊 Columns ▾

📅 Group by ▾

Table

Chart

Evaluation

Experimental

Traces

Experimental

<input type="checkbox"/>	Run Name	Created	⌵	Dataset
<input type="checkbox"/>	🔴 second run	✅ 7 seconds ago		-
<input type="checkbox"/>	🔴 upset-bug-224	✅ 8 minutes ago		-

MLflow Tracking: Runs vergleichen

MLflow Example ⓘ Provide Feedback ⓘ Add Description

Share

Q metrics.rmse < 1 and params.model = "tree" ⓘ

Time created ▾

State: Active ▾

Datasets ▾

Sort: Created ▾

⋮

↻

+ New run

Group by ▾

Table **Chart** Evaluation **Experimental**

Traces **Experimental**

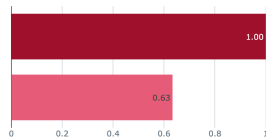
<input type="checkbox"/>	<input type="checkbox"/>	Run Name
<input type="checkbox"/>	<input checked="" type="checkbox"/>	second run
<input type="checkbox"/>	<input checked="" type="checkbox"/>	upset-bug-224

Q Search metric charts

Model metrics (2)

+ Add chart

acc



second run upset-bug-224

accuracy



second run upset-bug-224

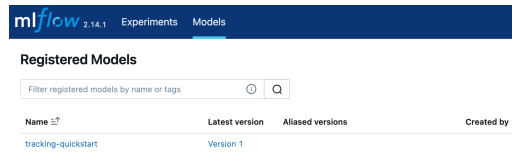
System metrics (0)

+ Add chart



In der *Model Registry* werden Modelle gehalten und verwaltet. Sie erlaubt das Verwenden, Speichern, Löschen, Versionieren, Taggen und Annotieren von Modellen.

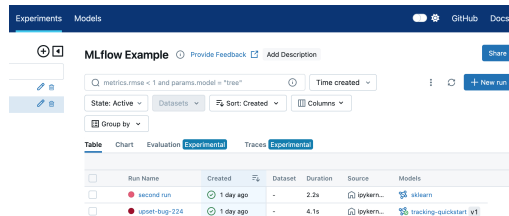
Via den *Model* Link im Header kann man auf die Model Registry zugreifen.



MLFlow Registry: Modelle registrieren

Nicht jedes geloggte Modell muss auch registriert werden. Modelle können entweder direkt beim loggen via optionalen Parameter aus `mlflow.*.log_model()` auch registriert werden, oder sie werden später via UI oder API registriert.

`mlflow.autolog()` registriert die geloggten Modelle nicht.



Run Name	Created	Dataset	Duration	Source	Models
second-run	1 day ago	-	2.2s	ipykern...	sklearn
upset-bug-224	1 day ago	-	4.1s	ipykern...	tracking-quickstart v1

Registrierte Modelle können eine optionale Beschreibung erhalten, können mit Tags versehen werden und sind versioniert.

[Registered Models](#) >

tracking-quickstart

Created Time: 2024-07-25 16:36:40

Last Modified: 2024-07-27 08:07:17

▼ Description [Edit](#)

Coolset Model Ever!

▼ Tags

Name	Value	Actions
purpose	teaching	✎ 🗑
<input type="text" value="Name"/>	<input type="text" value="Value"/>	<input type="button" value="Add"/>

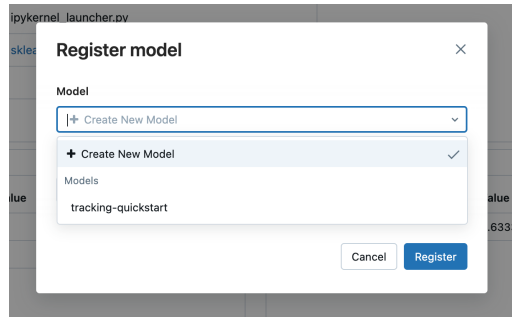
▼ Versions

Version	Registered at ↕	Created by	Tags	Aliases
✔ Version 1	2024-07-25 16:36:40		Add	Add

MLFlow Registry: Versionierung

Bei der Registrierung wird angegeben, ob das zu registrierende Modell als neues Modell registriert werden soll, oder ob es als bestehendes Modell mit einer neuen Version registriert werden soll.

Die Versionierung startet mit 1 und wird automatisch erhöht, wenn ein Modell mit gleichem Namen registriert wird.

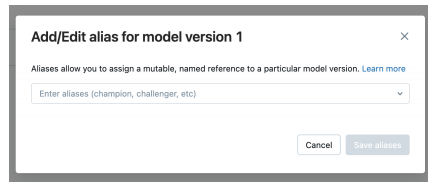


Modell-Versionen können im Sinne einer *mutable named reference* mit einem Alias versehen werden.

Aliase werden vor allem verwendet, um

- den ausführenden Code von der Modell-Version zu entkoppeln
- um einen Deployment Status abzubilden.

In der Produktion kann zum Beispiel eine Modell-Version mit dem Alias *champion* verwendet werden, gleichzeitig wird an einer neuen Version mit dem Alias *challenger* gearbeitet. Beide Versionen können gleichzeitig verwendet werden, z.B. für *A/B Tests* oder für Deployment Patterns wie *Extract and Contract*.



The screenshot shows a modal dialog box titled "Add/Edit alias for model version 1" with a close button (X) in the top right corner. Below the title, there is a descriptive text: "Aliases allow you to assign a mutable, named reference to a particular model version. [Learn more](#)". Underneath this text is a text input field with the placeholder text "Enter aliases (champion, challenger, etc)". At the bottom right of the dialog, there are two buttons: "Cancel" and "Save aliases".

Modelle können auf drei Arten referenziert und somit verwendet werden.

```
# via *run id* aus dem Tracking  
logged_model = 'runs:/7180c89f925946...663640810d/iris_model'
```

```
# via model name und version  
model_uri = "models:/tracking-quickstart/1"  
model_uri = "models:/tracking-quickstart/latest"
```

```
# via model name und version-alias  
model_uri = f"models:/tracking-quickstart@champion"
```

Hat man eine *model_uri*", kann mittels einer Variante der `load_model()` Funktion das Modell geladen werden.

```
# entweder
loaded_model = mlflow.sklearn.load_model(model_uri)
# oder
loaded_model = mlflow.pyfunc.load_model(model_uri)

loaded_model.predict(df)
```

Während der Entwicklungsphase kann mittels der *run_id* auch ein noch nicht registriertes Modell verwendet werden.

MLFlow hat kein brauchbares Konzept von Deployment Stages. Ein Modell kann *promoted* werden, diese Funktionalität beschränkt sich jedoch auf Kopieren, Renamen und Taggen eines Modells mit optionaler Access Control.

Mehr Infos in der offiziellen Doku: [Promoting an MLflow Model across environments](#)

Registrierte Modelle können mit MLFlow *geservet* werden:

```
mlflow models serve -m "models:/tracking-quickstart/1"
```

Dies startet einen Python Flask Server in der aktuellen Umgebung, sofern die notwendigen Dependencies vorhanden sind.

Neben dem vorgestellten high-level *MLFlow Fluent API* existiert ein zweites, lower-level API, der `MlflowClient`.

Der MLFlow Client stellt ein Python CRUD interface zur Verfügung, um Experimente, Runs, Model Versions, und registrierte Modelle zu verwalten.

```
from mlflow import MlflowClient
```

```
client = MlflowClient()  
experiment_id = client.create_experiment("New Experiment")  
client.delete_experiment(experiment_id)
```

MLFlow ist aus meiner Sicht nicht das ideale Tool. Für den Kurs hat es den Vorteil, dass wir es kostenfrei lokal installieren können, und es die zentralen Komponenten Tracking, Registry und Model API abdeckt.

■ Pro

- Free/Open Source und On-Premise möglich
- Grundfunktionalität relativ schnell verständlich

■ Con

- Doku chaotisch, keine verständliche Hierarchie
- Gliederung wirr (Models, Projects, Recipes), überlappende Verantwortlichkeiten
- Funktionalität der zentralen Komponenten limitiert
- Staging-Funktionalität wirr umgesetzt

- Aim
- DVC
- Sacred mit Omnilboard
- Tensorboard
- **Weights and Biases** (WANDB) bietet ausgezeichnetes Tracking. Idealerweise in der Cloud, bietet aber auch On-Premise Lösungen.
- Azure, AWS, Vertex AI, bieten Tracking, ebenso Comet, Dagshub, ClearML

- **Weights and Biases** (WANDB) bietet neben Tracking auch eine Registry
- **Hopsworks** und **Neptune.ai** bieten eine Model Registry
- Azure, AWS, Vertex AI, bieten Model Registries, ebenso DataRobot, Dataiku, Comet

- Model Plattform =

- Model Plattform =
 - Deployment API +

- Model Plattform =
 - Deployment API +
 - Model Registry +

- Model Plattform =
 - Deployment API +
 - Model Registry +
 - Model Server

- Model Plattform =
 - Deployment API +
 - Model Registry +
 - Model Server
 - (+ Experiment Tracking)

- Model Plattform =
 - Deployment API +
 - Model Registry +
 - Model Server
 - (+ Experiment Tracking)
- Die Inferenz kann auf verschiedene Arten durchgeführt werden

- Model Plattform =
 - Deployment API +
 - Model Registry +
 - Model Server
 - (+ Experiment Tracking)
- Die Inferenz kann auf verschiedene Arten durchgeführt werden
 - Stark abhängig vom Anwendungsfall und von der Infrastruktur

- Model Plattform =
 - Deployment API +
 - Model Registry +
 - Model Server
 - (+ Experiment Tracking)
- Die Inferenz kann auf verschiedene Arten durchgeführt werden
 - Stark abhängig vom Anwendungsfall und von der Infrastruktur
- Im Kurs verwenden wir MLFlow, es gibt aber auch viele gute Alternativen