Platform for Machine Learning Lifecycle

Jules S. Damji
@2twitme

Outline – Introduction to MLflow: How to Use MLflow Tracking - Module 1

- Overview of ML development challenges
- How MLflow tackles these
- Concepts and Motivations
- MLFlow Components
  - MLflow Tracking
  - How to use MLflow Tracking APIs
  - Use Databricks Community Edition
  - Explore MLflow UI
  - Tutorials & Exercises
- Q & A

https://github.com/dmatrix/olt-mlflow

Machine Learning Development is Complex

Traditional Software vs. Machine Learning

Traditional Software

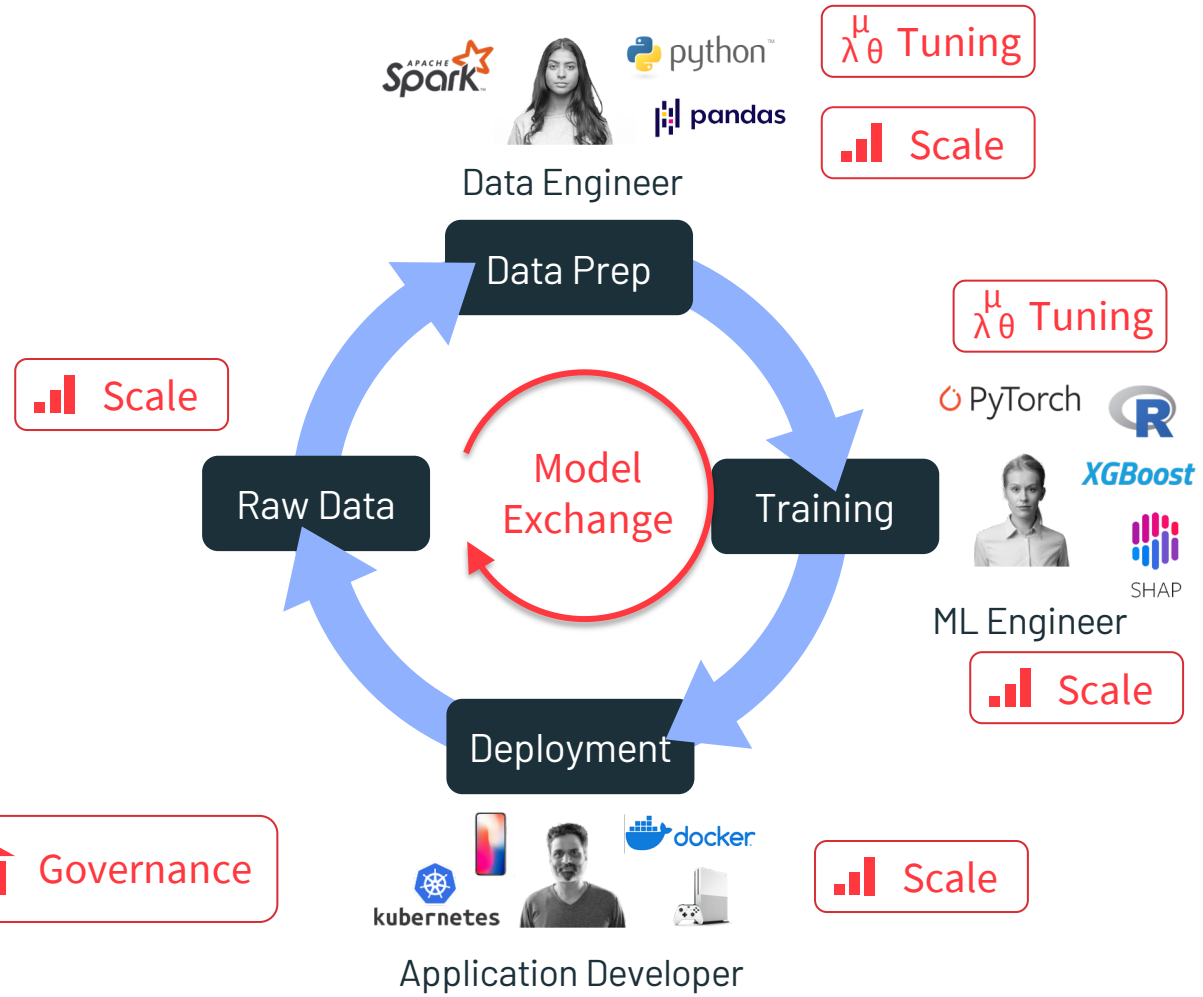
- Goal: Meet a functional specification
- Quality depends only on code
- Typically pick one software stack w/ fewer libraries and tools
- Limited deployment environments

Machine Learning

- Goal: Optimize metric (e.g., accuracy. Constantly experiment to improve it
- Quality depends on input data and tuning parameters
- Over time data changes; models drift…
- Compare + combine many libraries, model
- Diverse deployment environments

But Building ML Applications is Complex

- Continuous, iterative process
- Dependent on data
- Many teams and systems involved



Custom Machine Learning Platforms

Some Big Data Companies

+ Standardize the data prep / training / deploy loop: if you work with the platform, you get these!
– Limited to a few algorithms or frameworks
– Tied to one company's infrastructure
– Out of luck if you left the company….

Can we provide similar benefits in an open manner?

mlflow: An Open Source ML Platform

mlflow TRACKING

Experiment management

mlflow PROJECTS

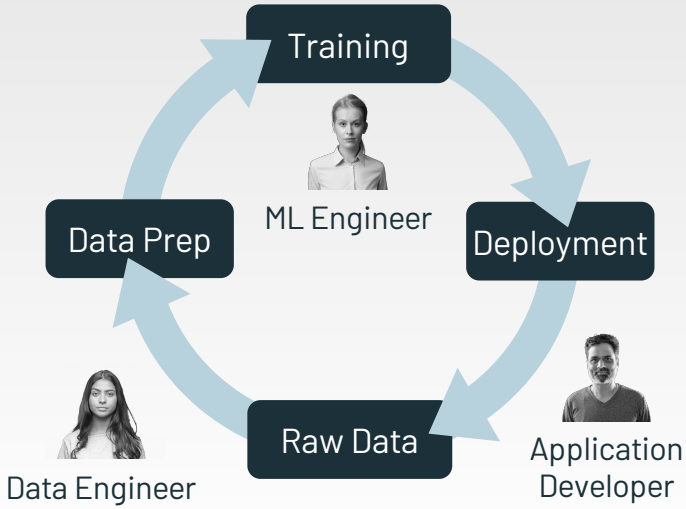
Reproducible runs

mlflow MODELS

Model packaging and deployment

mlflow MODEL REGISTRY

Model management



Any Language

python · R · Java · Scala

Any ML Library

TensorFlow · PyTorch · Keras · Spark · learn · H2O.ai

mleap · ONNX · GLUON · XGBoost · LightGBM · CONDA

kubernetes · docker · Amazon SageMaker · Azure Machine Learning · Google Cloud

MLflow Design Philosophy

API-First

- Submit runs, log models, metrics, etc. from popular library & language
- Abstract “model” lambda function that MLflow can then deploy in many places (Docker, Azure ML, Spark UDF)
- Open interface allows easy integration from the community

Key enabler: built around Programmatic APIs, REST APIs & CLI

Modular Design

- Allow different components individually (e.g., use MLflow’s project format but not its deployment tools
- Not monolithic
- But Distinctive and Selective

Key enabler: distinct components (Tracking/Projects/Models/Registry)

Introducing mlflow

Open machine learning platform

- Works with popular ML library & language
- Runs the same way anywhere (e.g., any cloud or locally)
- Designed to be useful for 1 or 1000+ person orgs
- Simple. Modular. Easy-to-use.
- Offers positive developer experience to get started!

Key Concepts in MLflow Tracking

Parameters: key-value inputs to your code

Metrics: numeric values (can update over time)

Tags and Notes: information about a run

Artifacts: files, data, and models

Source: what code ran?

Version: what of the code?

Run: an instance of code that runs by MLflow

Experiment: {Run, … Run}

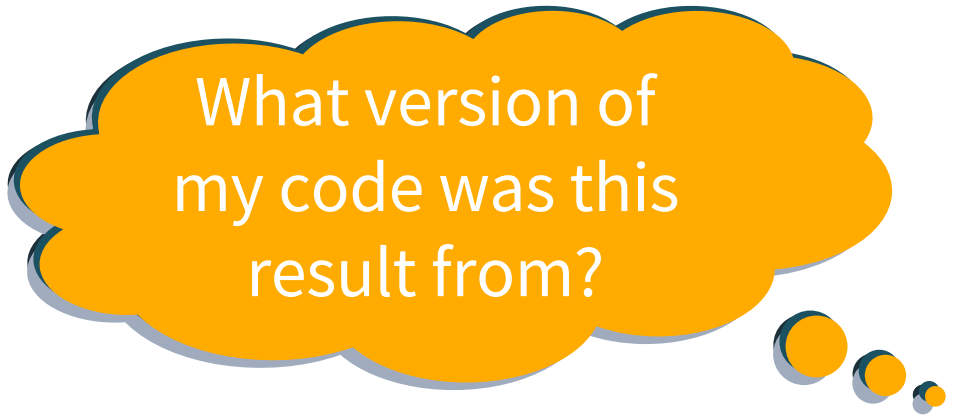
Model Development *without* MLflow Tracking

```
data   = load_text(file_name=file)
ngrams = extract_ngrams(data, N=n)
model  = train_model(ngrams,
                     learning_rate=lr)
score  = compute_accuracy(model)

print("For n=%d, lr=%f: accuracy=%f"
      % (n, lr, score))

pickle.dump(model, open("model.pkl"))
```

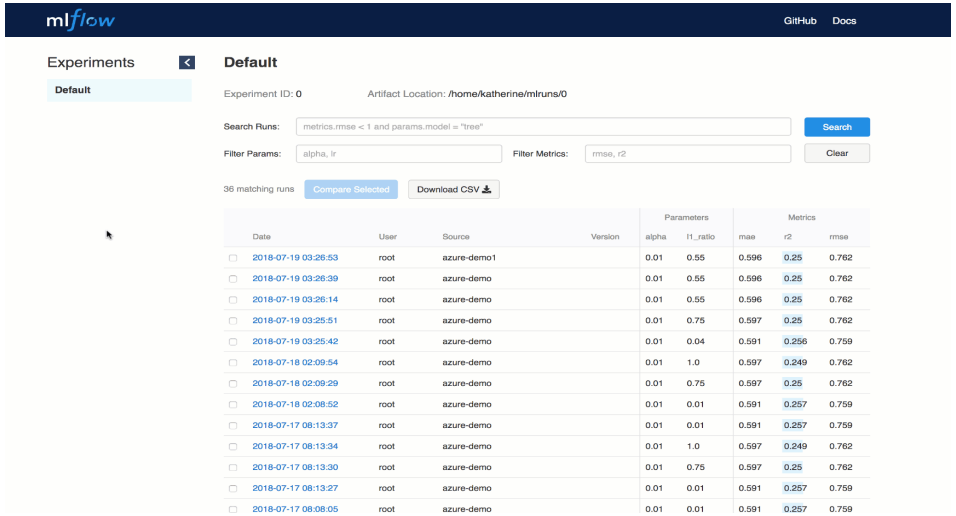
```
For n=2, lr=0.1: accuracy=0.71
For n=2, lr=0.2: accuracy=0.79
For n=2, lr=0.5: accuracy=0.83
For n=2, lr=0.9: accuracy=0.79
For n=3, lr=0.1: accuracy=0.83
For n=3, lr=0.2: accuracy=0.82
For n=4, lr=0.5: accuracy=0.75
...
```



Model Development *with* MLflow is *Simple!*

```
import mlflow
data   = load_text(file_name=file)
ngrams = extract_ngrams(data, N=n)
model  = train_model(ngrams,
                     learning_rate=lr)
score  = compute_accuracy(model)
with mlflow.start_run():
  mlflow.log_param("data_file", file)
  mlflow.log_param("n", n)
  mlflow.log_param("learn_rate", lr)
  mlflow.log_metric("score", score)
  mlflow.sklearn.log_model(model)
```

```
$ mlflow ui
```



Track parameters, metrics, artifacts, output files & code version

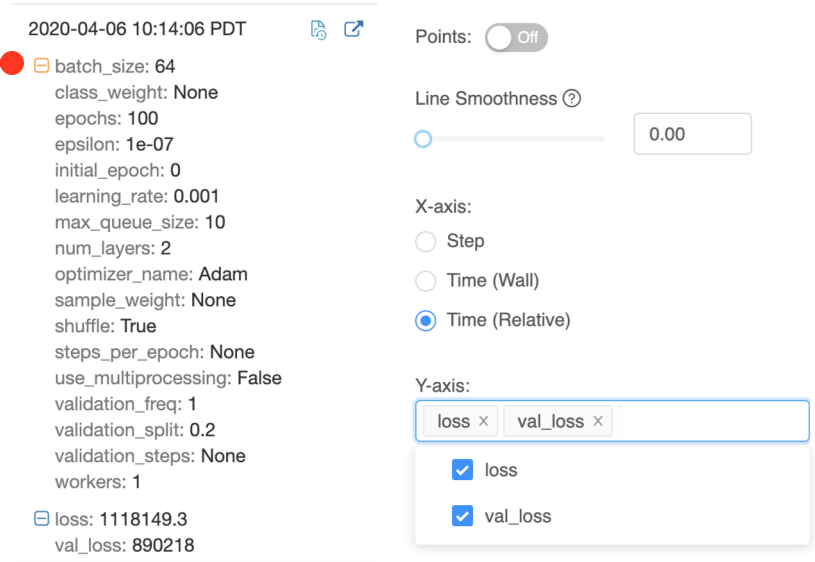
mlflow Tracking for ML Experiments

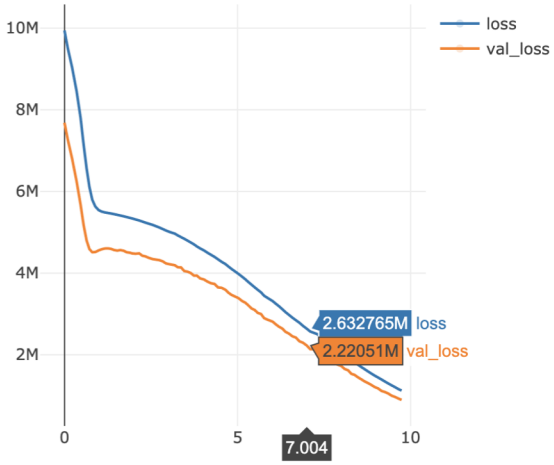
Easily track parameters, metrics, and artifacts in popular ML libraries

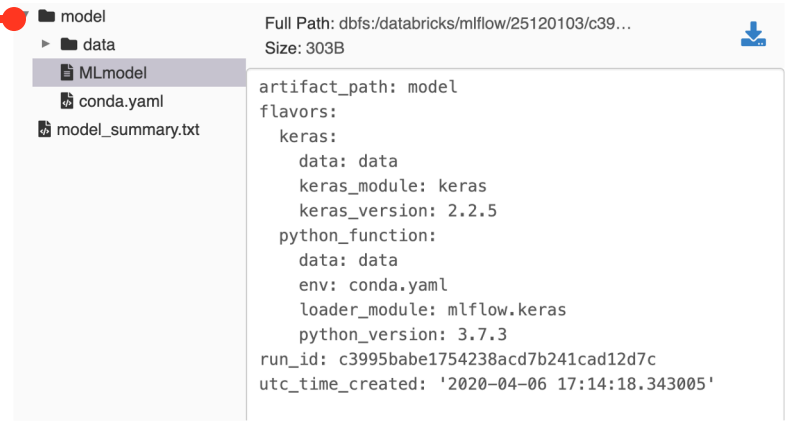
```
with mlflow.start_run(run_name='keras'):
  # log model and datasource
  mlflow.keras.autolog()
```

Library integrations:

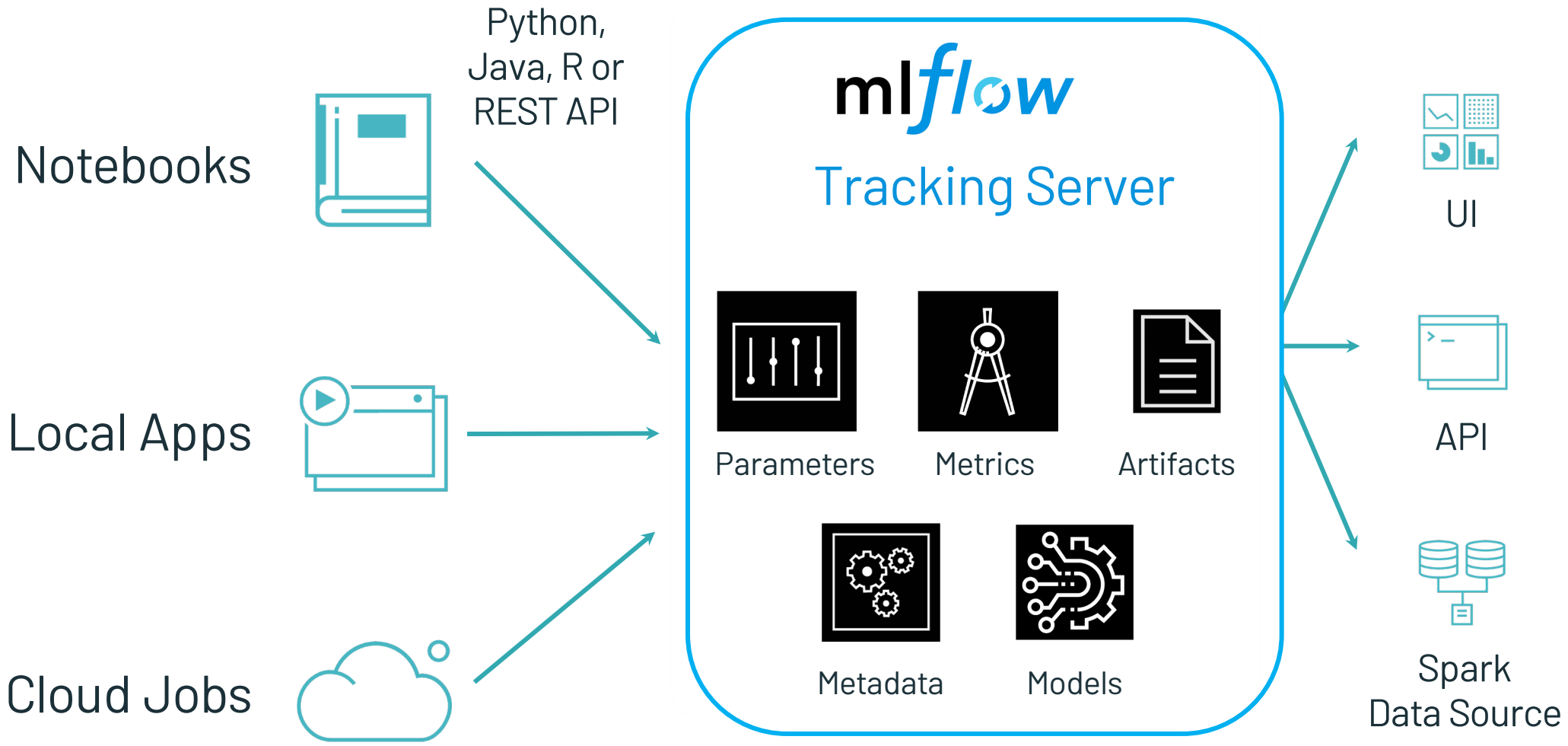
scikit learn · PyTorch · TensorFlow · Keras · Apache Spark · RAPIDS · H2O.ai · GLUON · XGBoost · LightGBM · spaCy · fast.ai · PyCaret · SHAP · Optuna · Delta Lake





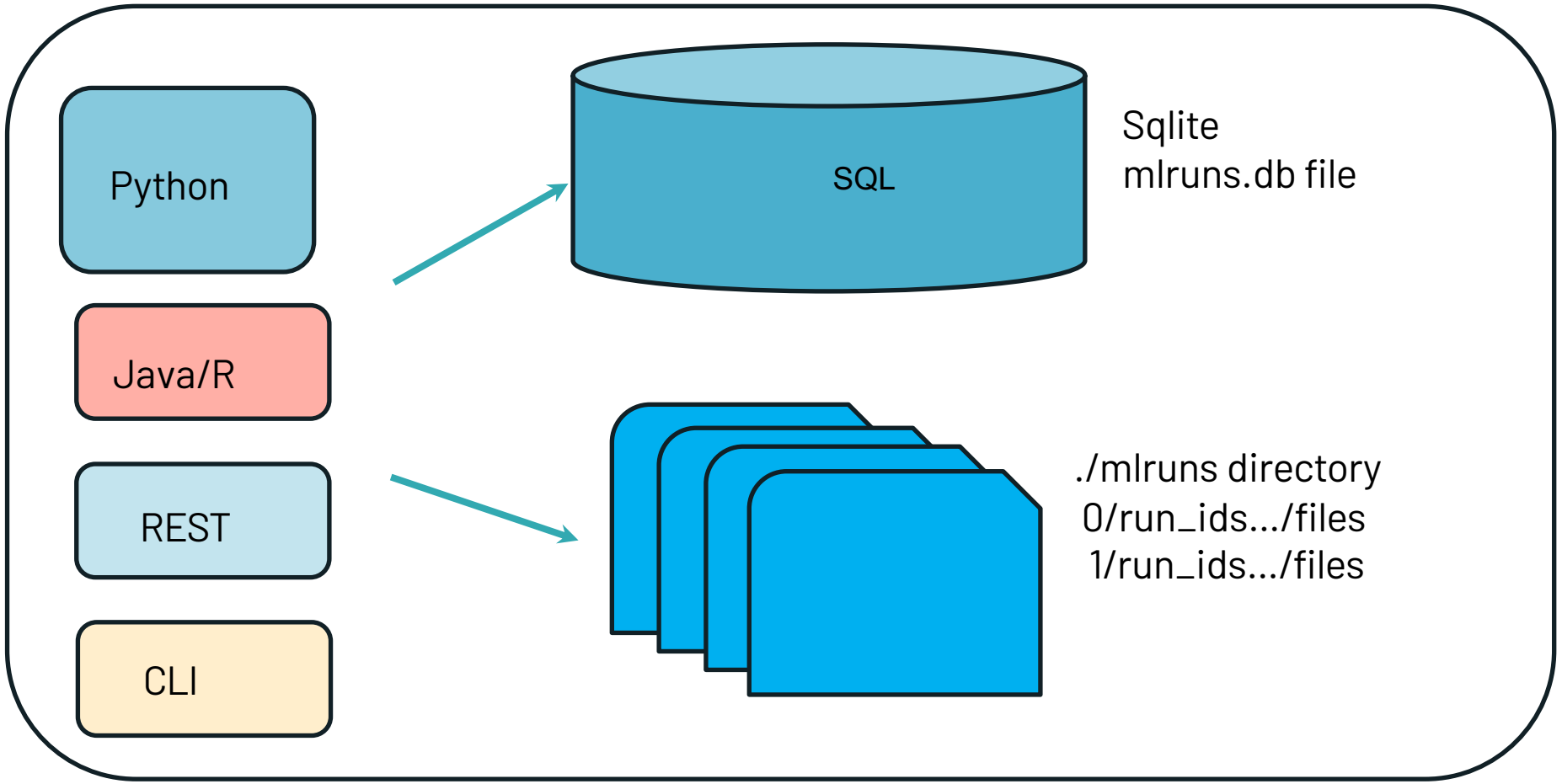


MLflow Tracking Server

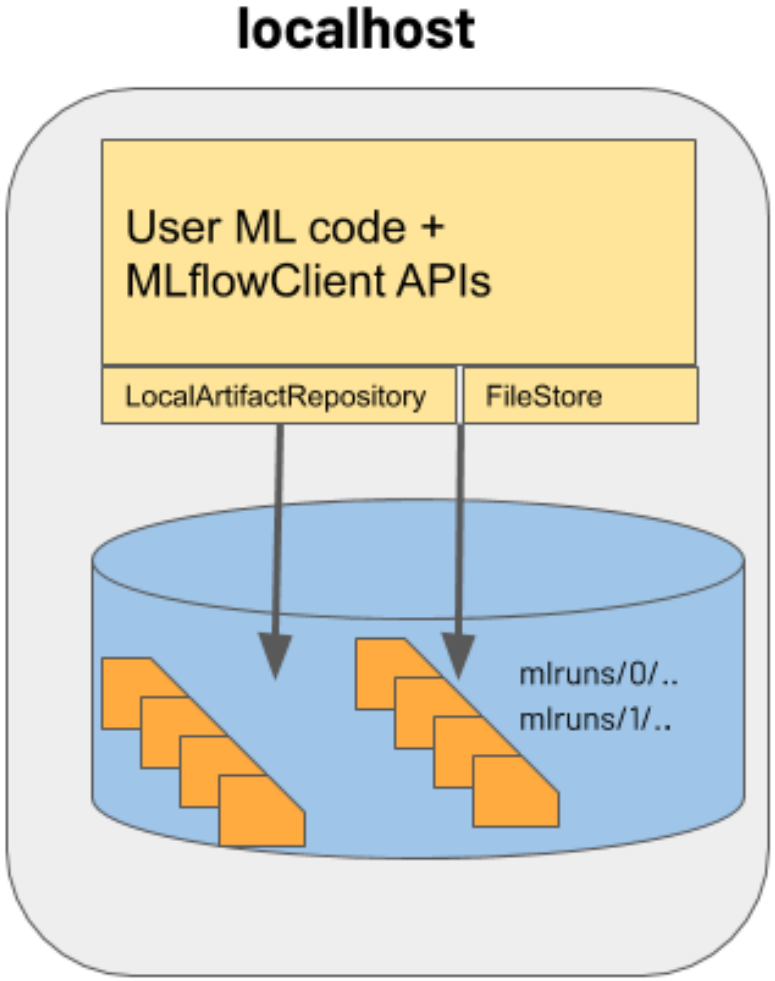



```
$ export MLFLOW_TRACKING_URI <URI>
mlflow.set_tracking_uri(URI)
```

MLflow Tracking Local Storage



MLflow user client code

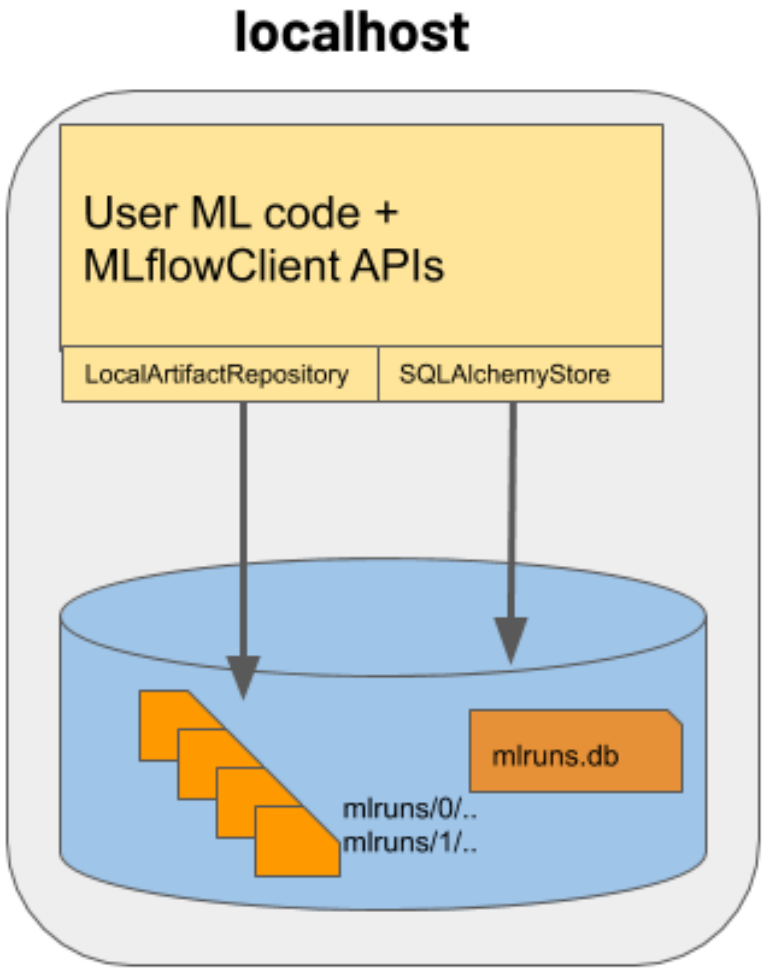


Scenario 1: MLflow on the localhost

No tracking server is involved here, as we run everything locally on our our localhost or laptop, hence no REST APIs. The **MLflowClient** directly interfaces with the **FileStore**. This store is used both as an artifact store and backend store under the **./mlruns** directory.

Interaction and Flow:

1. **MLflowClient** --> creates an instance of **LocalArtifactRepository** (to store artifacts)
2. **MLflowClient** --> creates an instance of **FileStore** (to save MLflow entities)
3. Both artifacts and MLflow entities are stored on the local disk under **./mlruns/….**



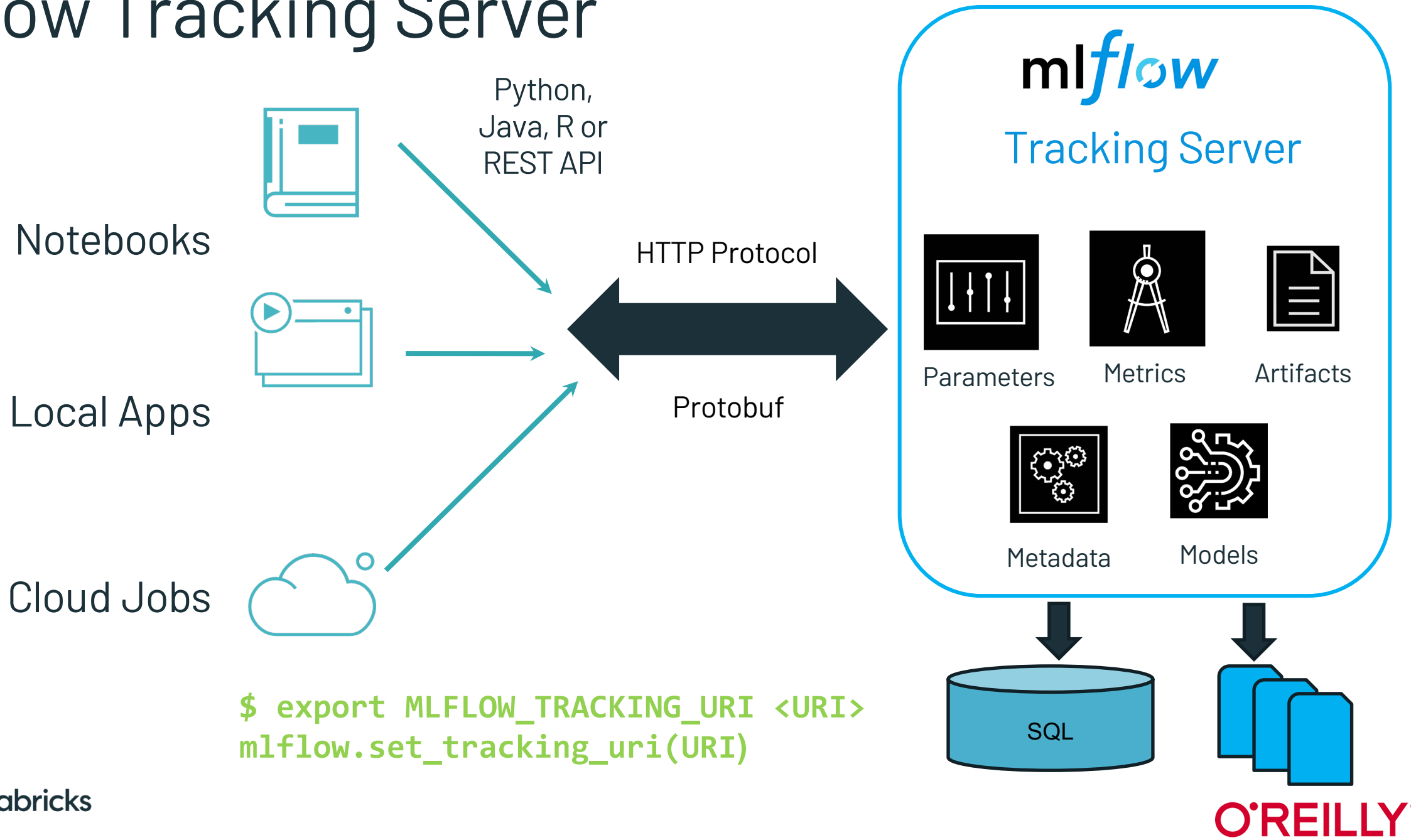
Scenario 2: MLflow on the localhost with backend store as an SQLAlchemy compatible database type: sqlite

No tracking server, but use an SQLAlchemy compatible backend store for MLflow entities. Artifacts are stored locally in the local **./mlruns** directory, and MLflow entities are stored in an sqlite database file **mlruns.db**

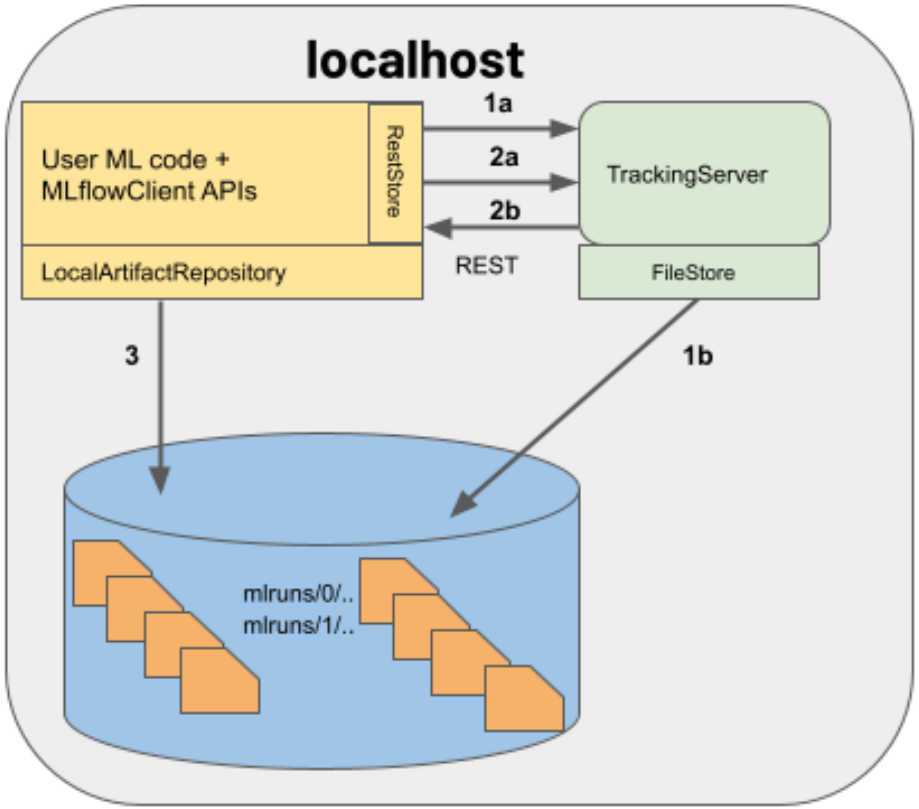
Interaction and Flow:

1. **MLflowClient** --> creates an instance of **LocalArtifactRepository** (to save artifacts)
2. **MLflowClient** --> creates an instance of **SQLAlchemyStore** (to save MLflow entities) and writes to sqlite file **mlruns.db**)

MLflow Tracking Server




```
$ export MLFLOW_TRACKING_URI <URI>
mlflow.set_tracking_uri(URI)
```



Scenario 3: Tracking server launched at localhost (default port 5000) : `mlflow server --backend-store-uri file://path/mlruns`

1. Backend store and Artifact Store both use local `file://path/mlruns`. That is, they use the **LocalArtifactRepository** to save artifacts and **FileStore** to save MLflow entities (runs, params, metrics, tags, etc).
2. Client will use an instance of **RestStore** and make REST APIs calls to the tracking server at default port 5000

Interaction and Flow:

MLflow Entities:

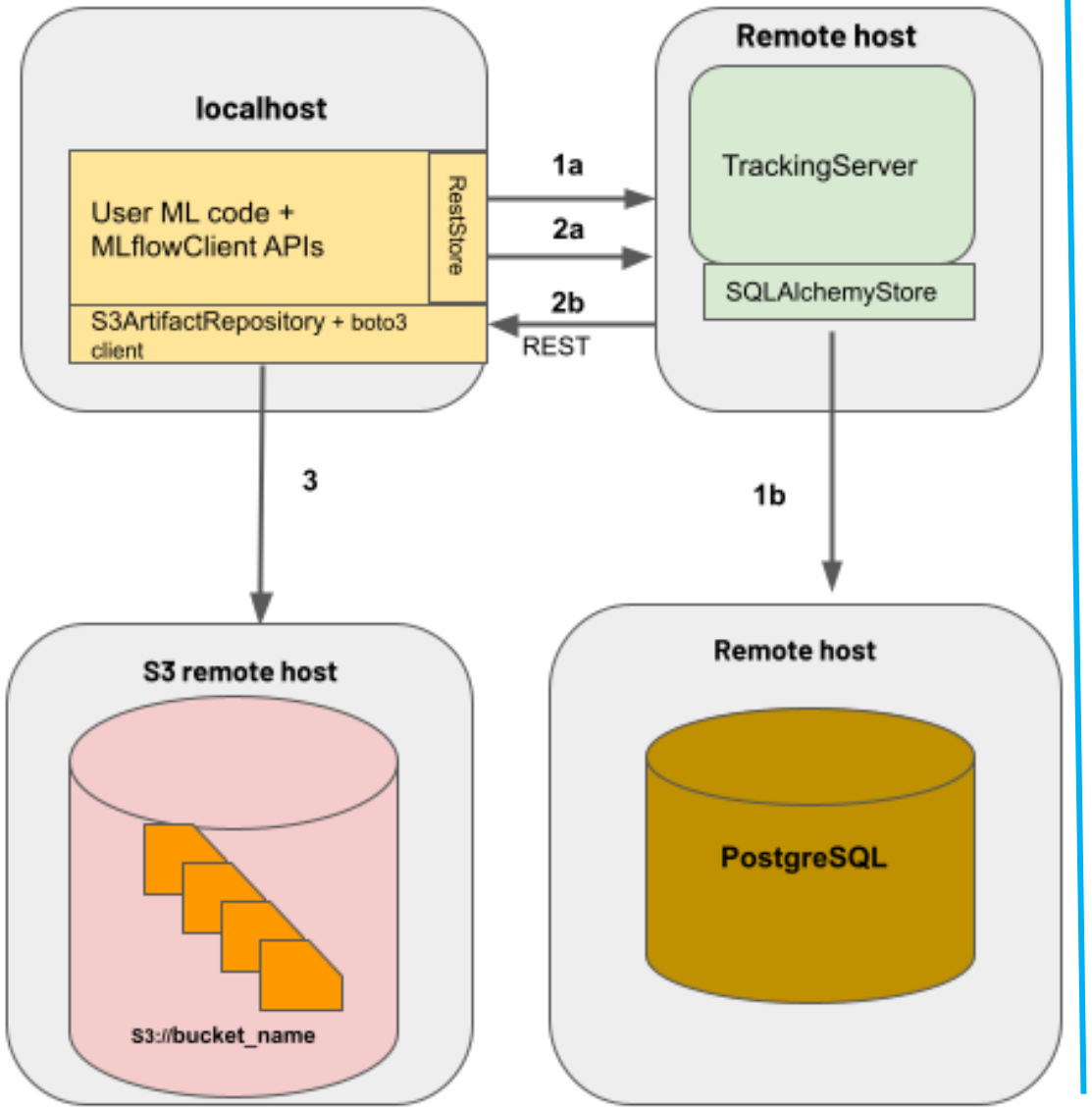
part 1a & b: MLflowClient --> creates an instance of **RestStore** → REST Log MLflow entities Request API Call --> **Tracking Server** → instance of **FileStore** (to save MLflow entities–params, runs, metrics, etc)

Artifacts:

part 2a: MLflowClient → **RestStore** --> REST Request API Call --> **Tracking Server** (fetch artifact store URI)

part 2b: Tracking Server --> REST Response with artifact store URI --> **MLflowClient**

part 3: **MLflowClient** --> creates an instance of **LocalArtifactRepository** (to stores artifacts)



Scenario 4: `mlflow server --backend-store-uri postgresql://URI --default-artifact-root S3:/bucket_name --host hostname`

1. Backend store is a PostgreSQL://URI at a remote host
2. URI scheme-based concrete class of **S3ArtifactRepository** for URI `S3:/bucket_name`

MLflow Entities:

part 1a & b: MLflowClient --> **RestStore** --> REST Request API Call --> **Tracking Server** --> creates an instance of **SQLAlchemyStore** (to store MLflow entities, params, runs, metrics, etc) connects to remote host Postgres DB

Artifacts:

part 2a: MLflowClient --> **RestStore** --> REST Request API Call --> **Tracking Server** (fetch artifact store URI)

part 2b: **Tracking Server** --> REST Response Response with artifact store URI --> **MLflowClient**

part 3: **MLflowClient** --> instance of **S3ArtifactRepository** --> S3 remote host bucket (to store artifacts)

The **MLflowClient** will use the **S3ArtifactRepository** (`s3:/bucket_name/`) to save artifacts on the remote S3 bucket, and the **Tracking Server** will use an existing instance of **SQLAlchemyStore** (`postresql://URI`) to save MLflow entities (runs, params, metrics, tags, etc), after each REST request to log MLflow entities.

MLflow Tracking Server Storage

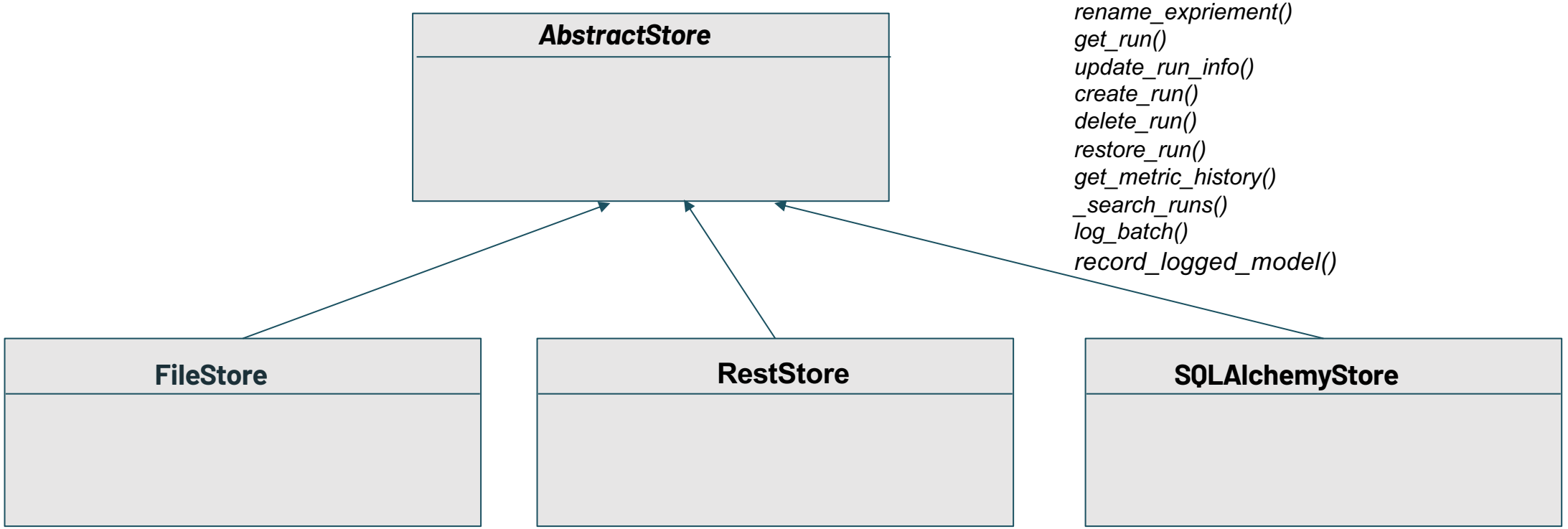
Entity (Metadata) Backend Store

- FileStore (local filesystem)
  - ***mlruns*** directory by default
- SQLStore (via SQLAlchemy)
  - PostgreSQL, MySQL, SQLite
- MLflow Plugins Scheme
  - Customized Entity Metastore
- Managed MLflow on Databricks
  - MySQL on AWS and Azure

Artifact Store

- Local Filesystem
  - ***mlruns*** directory
- S3 backed store
- Azure Blob storage
- Google Cloud Storage
- DBFS artifact repo

Backend Store Class Hierarchy



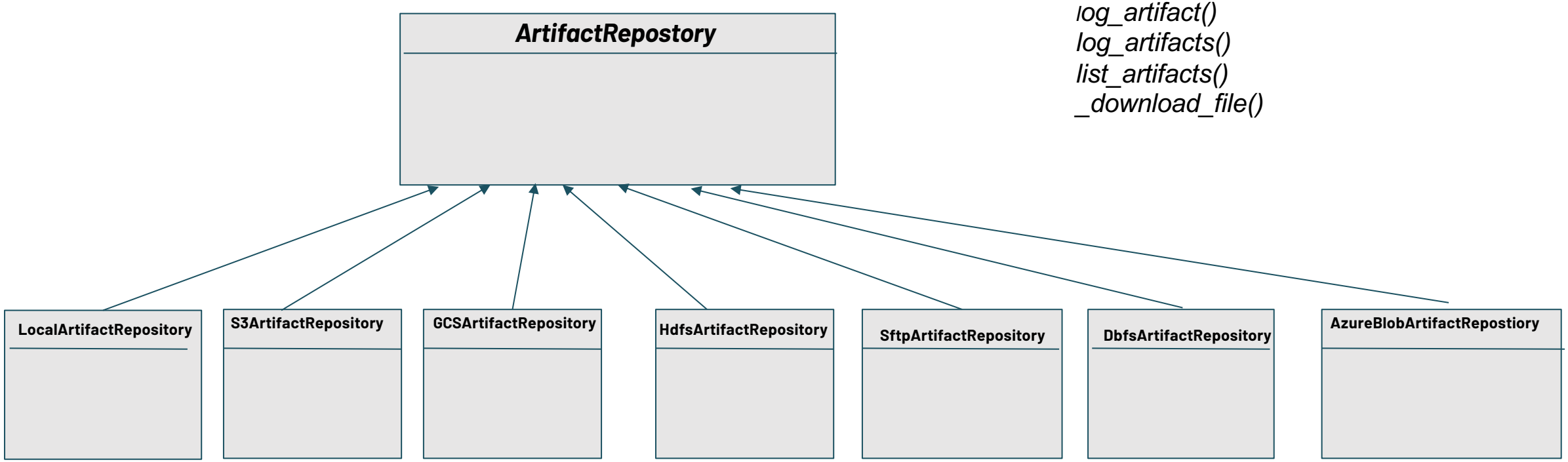
Abstract methods:

list_experiments()
create_experiment()
delete_experiment()
restore_experiment()
rename_expriement()
get_run()
update_run_info()
create_run()
delete_run()
restore_run()
get_metric_history()
_search_runs()
log_batch()
record_logged_model()

ArtifactRepository Class Hierarchy

Abstract methods:

log_artifact()
log_artifacts()
list_artifacts()
_download_file()



Other Concrete classes include:

- DatabricksArtifactRepository
- DatabricksModelsArtifactRepository
- FtpArtifactRepository
- ModelsArtifactRepository
- RunsArtifactRepository

Python API

The MLflow Python API is organized into the following modules. The most common functions are exposed in the `mlflow` module, so we recommend starting there.

- mlflow
- mlflow.azureml
- mlflow.deployments
- mlflow.entities
- mlflow.fastai
- mlflow.gluon
- mlflow.h2o
- mlflow.keras
- mlflow.lightgbm
- mlflow.mleap
- mlflow.models
- mlflow.onnx
- mlflow.projects
- mlflow.pyfunc
  - Inference API
  - Filesystem format
    - MLModel configuration
  - Creating custom Pyfunc models
    - Workflows
    - Which workflow is right for my use case?
- mlflow.pytorch
- mlflow.sagemaker
- mlflow.sklearn
- mlflow.spacy
- mlflow.spark
- mlflow.tensorflow
- mlflow.tracking

Built-In Model Flavors

MLflow provides several standard flavors that might be useful in your applications. Specifically, many of its deployment tools support these flavors, so you can export your own model in one of these flavors to benefit from all these tools:

- Python Function (`python_function`)
- R Function (`crate`)
- H_2O (`h2o`)
- Keras (`keras`)
- MLeap (`mleap`)
- PyTorch (`pytorch`)
- Scikit-learn (`sklearn`)
- Spark MLlib (`spark`)
- TensorFlow (`tensorflow`)
- ONNX (`onnx`)
- MXNet Gluon (`gluon`)
- XGBoost (`xgboost`)
- LightGBM (`lightgbm`)

mlflow.sklearn

The `mlflow.sklearn` module provides an API for logging and loading scikit-learn models. This module exports scikit-learn models with the following flavors:

Python (native) pickle format

This is the main flavor that can be loaded back into scikit-learn.

`mlflow.pyfunc`

Produced for use by generic pyfunc-based deployment tools and batch inference.

`mlflow.sklearn.get_default_conda_env`(*include_cloudpickle=False*) [source]

Returns

The default Conda environment for MLflow Models produced by calls to `save_model()` and `log_model()`.

`mlflow.sklearn.load_model`(*model_uri*) [source]

Load a scikit-learn model from a local file or a run.

Parameters

model_uri –

The location, in URI format, of the MLflow model, for example:

- `/Users/me/path/to/local/model`
- `relative/path/to/local/model`

MLflow API Data Flow: Localhost defaults

- **No** mlflow server --backend-store-uri --default-root-artifact-uri
- **Everything** stored under local directory **mlruns**

```
with mlflow.start_run(run_name="DEFAULTS") as run:
    params = {"n_estimators": 3, "random_state": 42}
    sk_learn_rfr = RandomForestRegressor(params)

    # Log params using the MLflow sklearn APIs
     mlflow.log_params(params)
    mlflow.log_param("param_1", randint(0, 100))
    mlflow.log_metric("metric_1", random())
    mlflow.log_metric("metric_2", random() + 1)
    mlflow.sklearn.log_model(sk_learn_rfr,
                             artifact_path="sklearn-model")
```

```
mlruns
└── 0
    ├── e7d505e713f14d64a74279f6c1a5d8e7
    │   ├── artifacts
    │   │   └── sklearn-model
    │   │       ├── MLmodel
    │   │       ├── conda.yaml
    │   │       └── model.pkl
    │   ├── meta.yaml
    │   ├── metrics
    │   │   ├── metric_1
    │   │   └── metric_2
    │   ├── params
    │   │   ├── n_estimators
    │   │   ├── param_1
    │   │   └── random_state
    │   └── tags
    │       ├── mlflow.log-model.history
    │       ├── mlflow.runName
    │       ├── mlflow.source.git.commit
    │       ├── mlflow.source.name
    │       ├── mlflow.source.type
    │       └── mlflow.user
    └── meta.yaml

7 directories, 16 files
```

MLflow API Data Flow: Localhost w/ SQLAlchemy backend store

- **No** mlflow server --backend-store-uri --default-root-artifact-uri
- **Artifacts** stored under local directory **mlruns**
- **Mlflow** entities stored in **mlruns.db**

```
local_store_uri = "sqlite:///mlruns.db"
mlflow.set_tracking_uri(local_store_uri)
```

```
with mlflow.start_run(run_name="LOCAL_REGISTRY") as run:
    params = {"n_estimators": 3, "random_state": 42}
    sk_learn_rfr = RandomForestRegressor(params)

    # Log params using the MLflow sklearn APIs
    mlflow.log_params(params)
    mlflow.log_param("param_1", randint(0, 100))
    mlflow.log_metric("metric_1", random())
    mlflow.log_metric("metric_2", random() + 1)
    mlflow.sklearn.log_model(sk_learn_rfr,
           artifact_path="sklearn-model")
```

```
mlruns
└── 0
    └── 1f896ddc61d34d089d4126eddceb5b52
        └── artifacts
            └── sklearn-model
                ├── MLmodel
                ├── conda.yaml
                └── model.pkl

4 directories, 3 files
```

```
(tutorials) ➜  mlflow_fluent git:(master) ✗ sqlite3 mlruns.db
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
sqlite> .tables
alembic_version        metrics                registered_model_tags
experiment_tags        model_version_tags     registered_models
experiments            model_versions         runs
latest_metrics         params                 tags
sqlite> select * from params;
n_estimators|3|29c1e723cfad48bbbc20c4ca52414780
random_state|42|29c1e723cfad48bbbc20c4ca52414780
param_1|10|29c1e723cfad48bbbc20c4ca52414780
sqlite> select * from metrics;
metric_1|0.216184107170305|1597259612673|29c1e723cfad48bbbc20c4ca52414780|0|0
metric_2|1.17106399018329|1597259612697|29c1e723cfad48bbbc20c4ca52414780|0|0
sqlite>
```

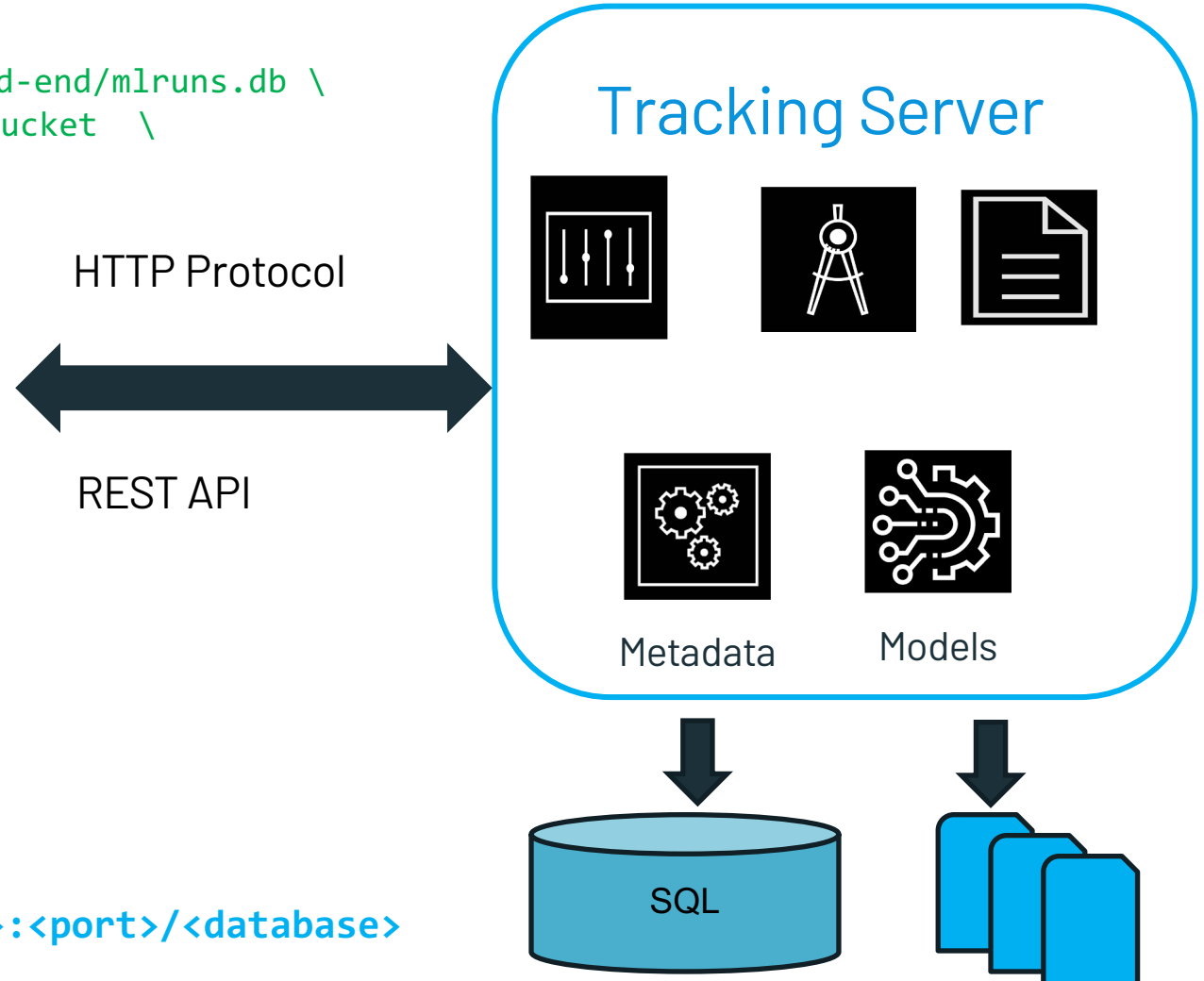
MLflow API Data Flow: Use Tracking Server SQLAlchemy backend store

```
mlflow server --backend-store-uri sqlite:///mnt/backend-end/mlruns.db \
              --default-artifact-root s3://my-mlflow-bucket  \
              --host host.mydomain.com
```

```
tracking_uri = “https://host.mydomain.com:5000”
mlflow.set_tracking_uri(tracking_uri)
```

```
with mlflow.start_run() as run:
    params = {"n_estimators": 3, "random_state": 42}
    sk_learn_rfr = RandomForestRegressor(params)

    # Log params using the MLflow sklearn APIs
    mlflow.log_params(params)
    mlflow.log_param("param_1", randint(0, 100))
    mlflow.log_metric("metric_1", random())
    mlflow.log_metric("metric_2", random() + 1)
    mlflow.sklearn.log_model(sk_learn_rfr,
           artifact_path="sklearn-model")
```



Backend Store URI Format: MySQL, PostgreSQL, SQLite, etc.

```
<dialect>+<driver>://<username>:<password>@<host>:<port>/<database>
```

databricks

O'REILLY®

MLflow set of APIs

Fluent MLflow APIs

- Python
  - High-level operations for runs and experiments
  - Model Flavor APIs
- Java
  - MLflowContext
  - Experiments, runs, search, etc
- R
  - Experiments, runs, search etc

MLflowClient

- Low Level CRUD interface to experiments, runs, Entities, Model Registry, etc
- `import mflow.tracking`
- `client = MLflowClient(**kwargs)`
- `Metric,Param,Search etc`

MLflow REST API

- Make REST calls to Tracking server with endpoints
- `https://<traciking_server>/api/..`
- `/2.0/mlfow/2.0/runs/log-metric`
- `/2.0/mlflow/runs/log-parameter`
- `/2.0/mlflow/runs/search`

mlflow Tracking for Model Schemas

Record what fields are consumed & produced by the model to prevent data mismatches

in MLflow 1.11

```
with mlflow.start_run(run_name='keras'):
  # log model and datasource
  mlflow.keras.autolog()
  mlflow.spark.autolog()

  sig = infer_signature(X_train, y_train)
```

▾ Schema

Name	Type
Inputs (7)	
user_purchases_7d	long
user_purchases_14d	long
user_location	string
user_last_login	string
item_inventory	long
Outputs (1)	
prediction	double

mlflow Tracking for Interpretability

SHAP library feature importances and visualizations

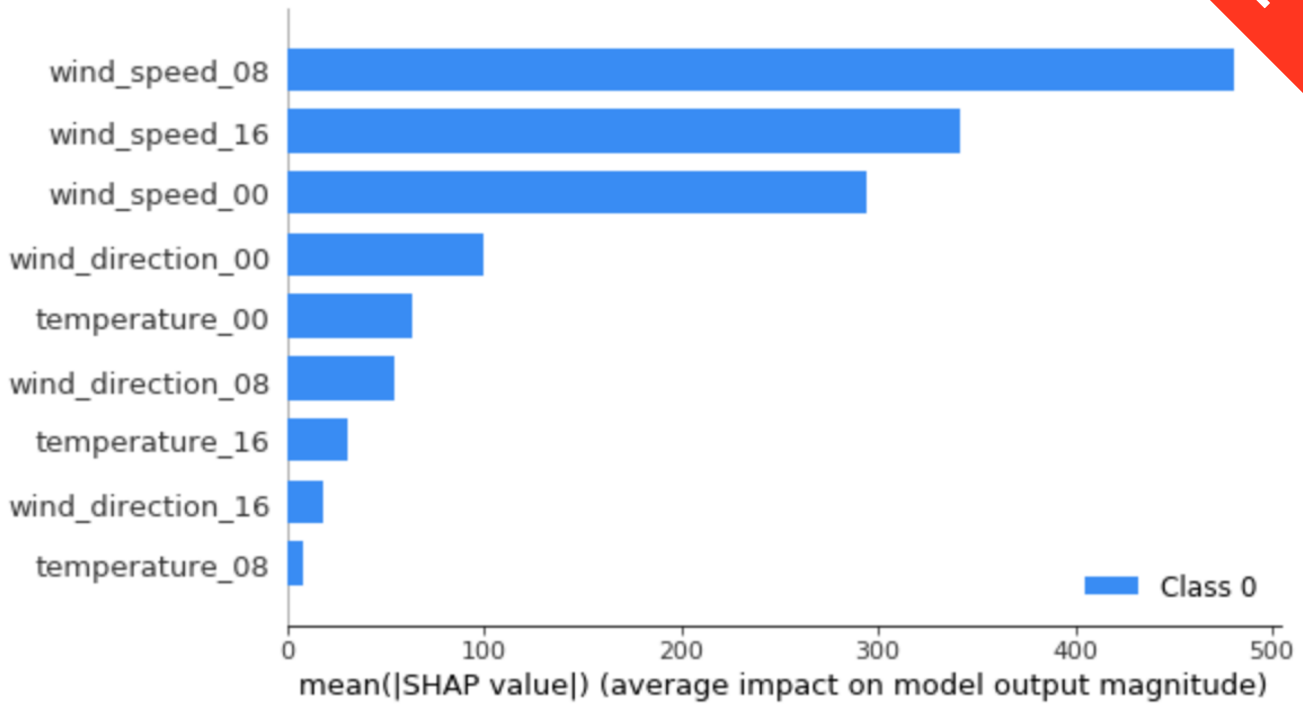
in MLflow 1.12

```
with mlflow.start_run(run_name='keras'):
  # log model and datasource
  mlflow.keras.autolog()
  mlflow.spark.autolog()

  sig = infer_signature(X_train, y_train)

  mlflow.shap.log_explanation(model, X_train[:100])
```

SHAP



Easily log and query all aspects of your ML development process

mlflow + PyTorch

in MLflow 1.12

Collaboration between Facebook and Databricks to bring ML platform features to PyTorch

- MLflow autologging for PyTorch Lightning
- TorchScript support for faster packaged models
- Model deployment to TorchServe

MLflow 1.12 Features Extended PyTorch Integration

And model explainability, autologging, and other enhancements

by Jules Damji, Siddharth Murching and Harutaka Kawamura
Posted in **ENGINEERING BLOG** | November 13, 2020

databricks

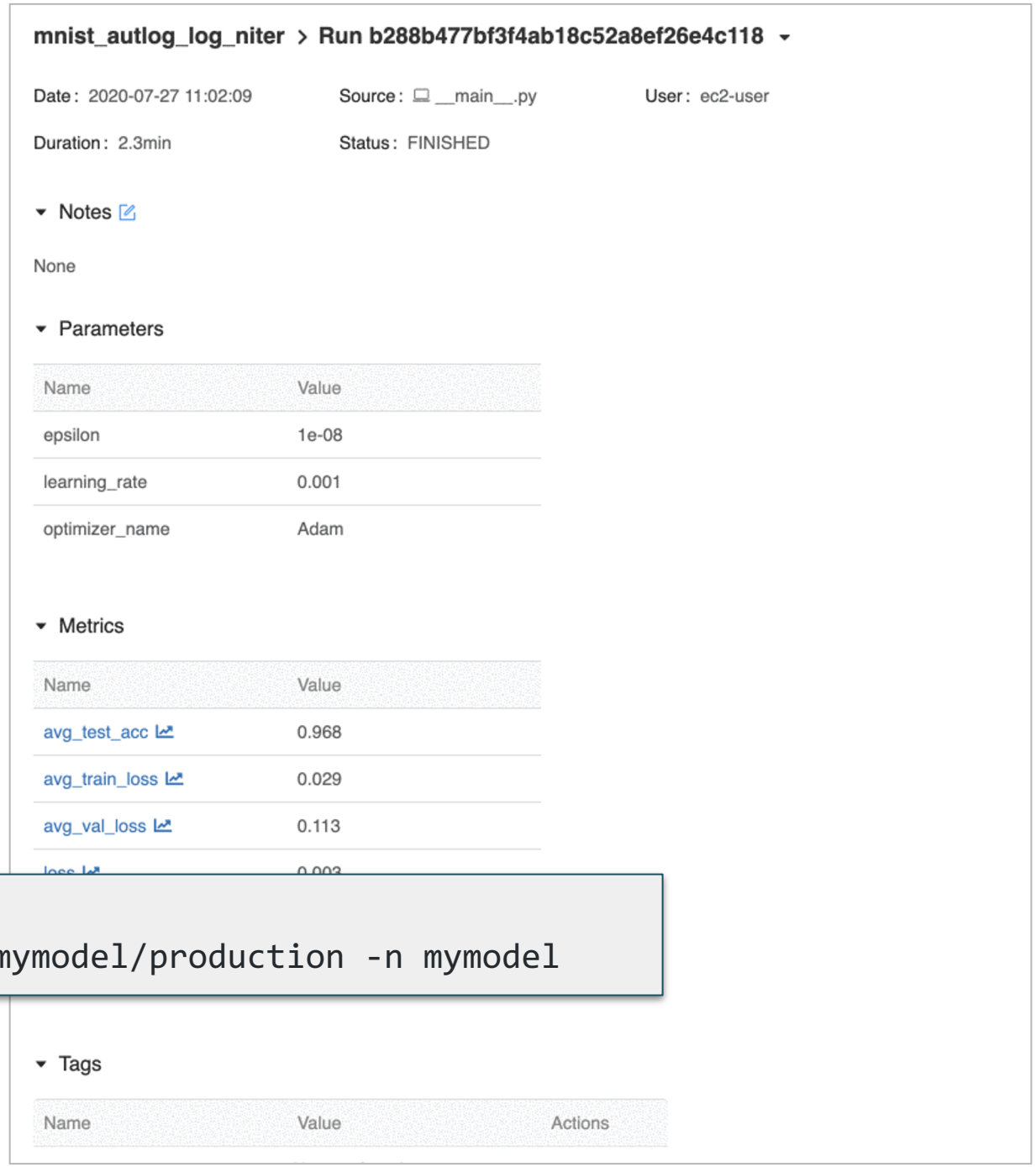
O'REILLY®

mlflow + PyTorch

```
# Enable MLflow autologging
import mlflow.pytorch_lightning
mlflow.pytorch_lightning.autolog()

# Train a model with PyTorch Lightning
model = MyModel()
trainer = pl.Trainer()
trainer.fit(model, train, val)
```

```
# Deploy to TorchServe
mlflow deployments create -t torchserve -m models:/mymodel/production -n mymodel
```



databricks

mlflow + PyTorch

MLflow and PyTorch — Where Cutting Edge AI meets MLOps

PyTorch Follow
Nov 12 · 6 min read

Authors: *Geeta Chauhan, PyTorch Partner Engineering Lead and Joe Spisak, PyTorch Product Lead at Facebook*

New Tracking APIs

in MLflow 1.13

Extend Tracking APIs for new types of Artifacts

- `mlflow.log_text("text1", "file1.txt")`
- `mlflow.log_text("<h1>header</h1>", "index.html")`
- `mlflow.log_dict(dictionary, "dict.json")`
- `mlflow.log_figure(figure, “figure.png”)`
  - matplotlib.figure.Figure
  - plotly.graph_objects.Figure
- `mlflow.log_image(image, “image.png”)`
  - PIL.Image.Image

```python
import mlflow

with mlflow.start_run():
    # Log text to a file under the run's root artifact
    # directory
    mlflow.log_text("text1", "file1.txt")

    # Log text in a subdirectory of the run's root artifact
    #directory
    mlflow.log_text("text2", "dir/file2.txt")

    # Log HTML text
    mlflow.log_text("<h1>header</h1>", "index.html")
```

What Did We Talk About? mlflow™

- Modular Components greatly simplify the ML lifecycle
- Easy to install and use & Great Developer Experience
- Develop & Deploy locally and track locally or remotely
- Available APIs: Python, Java & R (Soon Scala)
- Visualize experiments and compare runs

Learning More About MLflow & Get Involved!

- pip install mlflow – to get started
- Find docs & examples at mlflow.org
- Peruse code and contribute at MLflow Github
- Join the Slack channel
- More MLflow tutorials

MLflow Tracking Tutorials

https://github.com/dmatrix/olt-mlflow

Thank you! ☺

Q & A

jules@databricks.com
@2twitme
https://www.linkedin.com/in/dmatrix/