mlrun

Release UNKNOWN

Iguazio

Apr 22, 2022

MLRUN BASICS

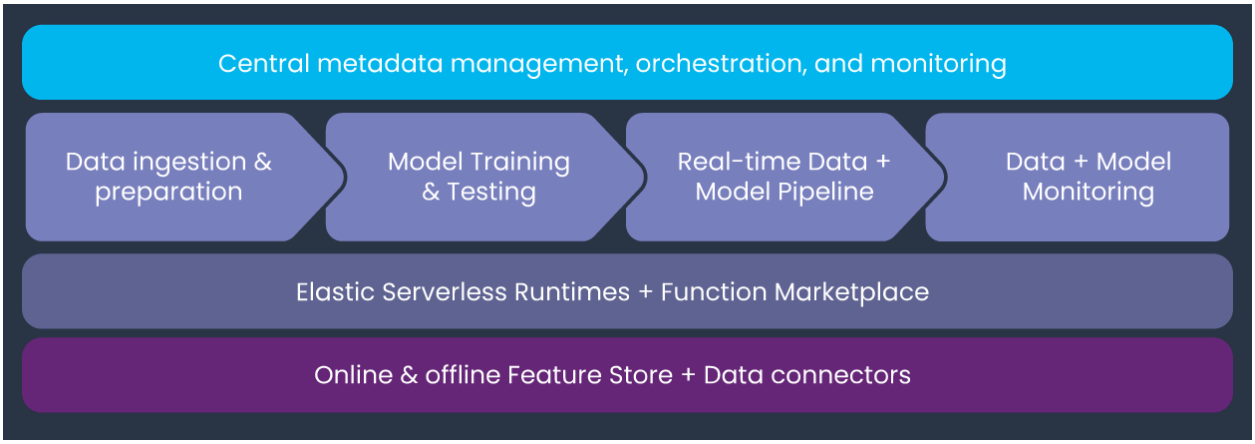
The Open-Source MLOps Orchestration Framework

MLRun is an open-source MLOps framework that offers an integrative approach to managing your machine-learning pipelines from early development through model development to full pipeline deployment in production. MLRun offers a convenient abstraction layer to a wide variety of technology stacks while empowering data engineers and data scientists to define the feature and models.

CHAPTER

ONE

THE MLRUN ARCHITECTURE



MLRun is composed of the following layers:

- **Feature and Artifact Store** Handles the ingestion, processing, metadata, and storage of data and features across multiple repositories and technologies.
- **Elastic Serverless Runtimes** Converts simple code to scalable and managed microservices with workload-specific runtime engines (such as Kubernetes jobs, Nuclio, Dask, Spark, and Horovod).
- **ML Pipeline Automation** Automates data preparation, model training and testing, deployment of real-time production pipelines, and end-to-end monitoring.
- **Central Management** Provides a unified portal for managing the entire MLOps workflow. The portal includes a UI, a CLI, and an SDK, which are accessible from anywhere.

CHAPTER

TWO

KEY BENEFITS

MLRun provides the following key benefits:

- **Rapid deployment** of code to production pipelines
- **Elastic scaling** of batch and real-time workloads
- **Feature management** – ingestion, preparation, and monitoring
- **Works anywhere** – your local IDE, multi-cloud, or on-prem

2.1 Table Of Contents

2.1.1 What is MLRun?

The challenge

As an ML developer or data scientist, you typically want to write code in your preferred local development environment (IDE) or web notebook, and then run the same code on a larger cluster using scale-out containers or functions. When you determine that the code is ready, you or someone else needs to transfer the code to an automated ML workflow (for example, using Kubeflow Pipelines). This pipeline should be secure and include capabilities such as logging and monitoring, as well as allow adjustments to relevant components and easy redeployment.

After you develop your model and feature engineering logic you need to deploy those into production pipelines with real-time feature engineering, online model serving, API and data integrations, model and data quality monitoring, no intrusive upgrades and so on.

However, the implementation is challenging: various environments ("runtimes") use different configurations, parameters, and data sources. In addition, multiple frameworks and platforms are used to focus on different stages of the development life cycle. This leads to constant development and DevOps/MLOps work.

Furthermore, as your project scales, you need greater computation power or GPUs, and you need to access large-scale data sets. This cannot succeed on laptops. You need a way to seamlessly run your code on a remote cluster and automatically scale it out.

Why MLRun?

When running ML experiments, ideally you can record and version your code, configuration, outputs, and associated inputs (lineage), so you can easily reproduce and explain your results. The fact that you probably need to use different types of storage (such as files and AWS S3 buckets) and various databases, further complicates the implementation.

Once the development is complete, it's time to serve models online or build real-time pipelines without having to refactor or re-implement the logic, or call in an army of developers to help.

Wouldn't it be great if you could write the code once, using your preferred development environment and simple "local" semantics, and then run it as-is on different platforms? Imagine having a layer that automates the build process, execution, data movement, scaling, versioning, parameterization, outputs tracking, deployment to production, monitoring, and more. A world of easily developed, published, or consumed data or ML "functions" that can be used to form complex and large-scale offline or real-time ML pipelines.

In addition, imagine a marketplace of ML functions that includes both open-source templates and your internally developed functions, to support code for reuse across projects and companies and thus further accelerate your work.

That is what MLRun does — simplifies & accelerates the time to production.

Architecture

MLRun is composed of the following layers:

- **Feature Store** — collects, prepares, catalogs, and serves data features for development (offline) and real-time (online) usage for real-time and batch data.
- **ML CI/CD pipeline** — automatically trains, tests, optimizes, and deploys or updates models using a snapshot of the production data (generated by the feature store) and code from the source control (Git).
- **Real-Time Serving Pipeline** — Rapid deployment of scalable data and ML pipelines using real-time serverless technology, including the API handling, data preparation/enrichment, model serving, ensembles, driving and measuring actions, etc.
- **Real-Time monitoring and retraining** — monitors data, models, and production components and provides a feedback loop for exploring production data, identifying drift, alerting on anomalies or data quality issues, triggering re-training jobs, measuring business impact, etc.

While each of those steps may be independent, they still require tight integration. For example:

- The training jobs need to obtain features from the feature store and update the feature store with metadata, which will be used in the serving or monitoring.
- The real-time pipeline needs to enrich incoming events with features stored in the feature store, and may use feature metadata (policies, statistics, schema, etc.) to impute missing data or validate data quality.
- The monitoring layer must collect real-time inputs and outputs from the real-time pipeline and compare it with features data/metadata from the feature store or model metadata generated by the training layer, and it needs to write all the fresh production data back to the feature store so it can be used for various tasks such as data analysis, model re-training (on fresh data), model improvements.

When one of the components detailed above is updated, it immediately impacts the feature generation, the model serving pipeline, and the monitoring. MLRun applies versioning to each component, as well as versioning and rolling upgrades across components.

Basic components

MLRun has the following main components that are used throughout the system:

- **Project** — a container for organizing all of your work on a particular activity. Projects consist of metadata, source code, workflows, data and artifacts, models, triggers, and member management for user collaboration. Read more in *Projects*.
- **Function** — a software package with one or more methods and runtime-specific attributes (such as image, command, arguments, and environment). Read more in *MLRun serverless functions* and *Creating and using functions*.
- **Run** — an object that contains information about an executed function. The run object is created as a result of running a function, and contains the function attributes (such as arguments, inputs, and outputs), as well the execution status and results (including links to output artifacts). Read more in *Runs, functions, and workflows*.
- **Artifact** — versioned data artifacts (such as data sets, files and models) are produced or consumed by functions, runs, and workflows. Read more in *Artifacts and models*.
- **Workflow** — defines a functions pipeline or a directed acyclic graph (DAG) to execute using Kubeflow Pipelines or MLRun *Real-time serving pipelines*. Read more in *Project workflows and automation*
- **UI** — a graphical user interface (dashboard) for displaying and managing projects and their contained experiments, artifacts, and code.

2.1.2 MLOps development flow

With MLrun, you can build an automated end to end ML pipeline comprising the following steps:

1. *Data collection and preparation*
2. *Training*
3. *Building online ML services (online serving)*
4. *Continuous monitoring, governance, and retraining*

Data collection and preparation

There is no ML without data. Before everything else, ML teams need access to historical and/or online data from multiple sources, and they must catalog and organize the data in a way that allows for simple and fast analysis (for example, by storing data in columnar data structures, such as Parquet).

In most cases, the raw data cannot be used as-is for machine learning algorithms for various reasons such as:

- The data is low quality (missing fields, null values, etc.) and requires cleaning and imputing.
- The data needs to be converted to numerical or categorical values which can be processed by algorithms.
- The data is unstructured in text, json, image, or audio formats, and needs to be converted to tabular or vector formats.
- The data needs to be grouped or aggregated to make it meaningful.
- The data is encoded or requires joins with reference information.
- The ML process starts with manual exploratory data analysis and feature engineering on small data extractions. In order to bring accurate models into production, ML teams must work on larger datasets and automate the process of collecting and preparing the data.

Furthermore, batch collection and preparation methodologies such as ETL, SQL queries, and batch analytics don't work well for operational or real-time pipelines. As a result, ML teams often build separate data pipelines which use stream processing, NoSQL, and containerized micro- services. 80% of data today is unstructured, so an essential part of building operational data pipelines is to convert unstructured textual, audio, and visual data into machine learning- or deep learning-friendly data organization.

MLOps solutions should incorporate a *feature store* that defines the data collection and transformations just once for both batch and real-time scenarios, processes features automatically without manual involvement, and serves the features from a shared catalog to training, serving, and data governance applications. Feature stores must also extend beyond traditional analytics and enable advanced transformations on unstructured data and complex layouts.

Training

Whether it's deep learning or machine learning, MLRun allows you to train your models at scale and capture all the relevant metadata for experiments tracking and lineage.

With MLOps, ML teams build machine learning pipelines that automatically collect and prepare data, select optimal features, run training using different parameter sets or algorithms, evaluate models, and run various model and system tests. All the executions, along with their data, metadata, code and results must be versioned and logged, providing quick results visualization, compare them with past results and understand which data was used to produce each model.

Pipelines can be more complex—for example, when ML teams need to develop a combination of models, or use Deep Learning or NLP.

ML pipelines can be triggered manually, or preferably triggered automatically when:

1. The code, packages or parameters change
2. The input data or feature engineering logic change
3. Concept drift is detected, and the model needs to be re-trained with fresh data

ML pipelines:

- Are built using micro-services (containers or serverless functions), usually over Kubernetes.
- Have all their inputs (code, package dependencies, data, parameters) and the outputs (logs, metrics, data/features, artifacts, models) tracked for every step in the pipeline, in order to reproduce and/or explain our experiment results.
- Use versioning for all the data and artifacts used throughout the pipeline.
- Store code and configuration in versioned Git repositories.
- Use Continuous Integration (CI) techniques to automate the pipeline initiation, test automation, review and approval process.

Pipelines should be executed over scalable services or functions, which can span elastically over multiple servers or containers. This way, jobs complete faster, and computation resources are freed up once they complete, saving significant costs.

The resulting models are stored in a versioned model repository along with metadata, performance metrics, required parameters, statistical information, etc. Models can be loaded later into batch or real-time serving micro-services or functions.

Building online ML services (online serving)

With MLRun, in addition to a batch inference, you can deploy a robust and scalable real-time pipeline for more complex and online scenarios. MLRun uses Nuclio, an open source serverless framework for creating real-time pipelines for *model deployment*.

Once an ML model has been built, it needs to be integrated with real-world data and the business application or front-end services. The whole application or parts thereof need to be deployed without disrupting the service. Deployment can be extremely challenging if the ML components aren't treated as an integral part of the application or production pipeline.

Production pipelines usually consist of:

- Real-time data collection, validation, and feature engineering logic
- One or more model serving services
- API services and/or application integration logic
- Data and model monitoring services
- Resource monitoring and alerting services
- Event, telemetry, and data/features logging services

The different services are interdependent. For example, if the inputs to a model change, the feature engineering logic must be upgraded along with the model serving and model monitoring services. These dependencies require online production pipelines (graphs) to reflect these changes.

Production pipelines can be more complex when using unstructured data, deep learning, NLP or model ensembles, so having flexible mechanisms to build and wire up our pipeline graphs is critical.

Production pipelines are usually interconnected with fast streaming or messaging protocols, so they should be elastic to address traffic and demand fluctuations, and they should allow non-disruptive upgrades to one or more elements of the pipeline. These requirements are best addressed with fast serverless technologies.

Production pipeline development and deployment flow:

1. Develop production components:
    - API services and application integration logic
    - Feature collection, validation, and transformation
    - Model serving graphs
2. Test online pipelines with simulated data
3. Deploy online pipelines to production
4. Monitor models and data and detect drift
5. Retrain models and re-engineer data when needed
6. Upgrade pipeline components (non-disruptively) when needed

Continuous monitoring, governance, and retraining

Once the model is deployed, use MLRun to track the operational statistics as well as identify drift. When drift is identified, MLRun can trigger the training pipeline to train a new model.

AI services and applications are becoming an essential part of any business. This trend brings with it liabilities, which drive further complexity. ML teams need to add data, code and experiment tracking, monitor data to detect quality problems, and *monitor models* to detect concept drift and improve model accuracy through the use of AutoML techniques and ensembles, and so on.

Nothing lasts forever, not even carefully constructed models that have been trained using mountains of well-labeled data. ML teams need to react quickly to adapt to constantly changing patterns in real-world data. Monitoring machine learning models is a core component of MLOps to keep deployed models current and predicting with the utmost accuracy, and to ensure they deliver value long-term.

2.1.3 Quick-Start Guide

MLRun is an end-to-end open-source MLOps solution to manage and automate your analytics and machine learning lifecycle, from data ingestion, through model development and full pipeline/model deployment, to model monitoring. Its primary goal is to ease the development of machine learning pipeline at scale and help organizations build a robust process for moving from the research phase to fully operational production deployments.

MLRun is automating the process of moving code to production by implementing a **serverless** approach, where different tasks or services are executed over elastic **serverless functions** (read more about *MLRun functions*), in this quick start guide we will use existing (marketplace) functions, see the ***tutorial*** with more detailed example of how to create and use functions.

Table of Contents

Working with MLRun

If you need to install MLRun, refer to the *Installation Guide*.

> **Note**: If you are using the Iguazio MLOps Platform, MLRun already comes preinstalled and integrated in your system.

If you are not viewing this quick-start guide from a Jupyter Lab instance, open it on your cluster, create a new notebook, and copy the sections below to the notebook to run them.

Set Environment

Before you begin, initialize MLRun by calling `set_environment` and provide it with the project name. All the work will be saved and tracked under that project.

```
import mlrun

project = mlrun.new_project('quick-start', user_project=True)
```

Train a Model

MLRun introduces the concept of *functions*. You can run your own code in functions, or use functions from the function marketplace. Functions can run locally or over elastic **"serverless"** engines (as containers over kubernetes).

In the example below, you'll use the `sklearn_classifier` from MLRun function marketplace to train a model and use a sample dataset (CSV file) as the input. You can read more on how to ***use data items*** from different data sources or from the ***Feature Store***.

Note: When training a model in an air-gapped site

If you are working in MLRun:

1. Download your data file and save it locally.
2. Run: `import os os.environ["env_name"] = 1`
3. Use the same command for the sample data set path, for example: `source_url = mlrun.get_sample_path("data/iris/iris_dataset.csv")`

If your system is integrated with an MLOps Platform:

1. Download your data file and save it locally.
2. In the UI, click the settings icon () in the top-right of the header in any page to open the **Settings** dialog.
3. Click **Environment variables | Create a new environment variable**, and set an environmental variable: SAMPLE_DATA_SOURCE_URL_PREFIX = the relative path to locally-stored data. For example: `/v3io/bigdata/path/...`
4. Use the same command for the sample data set path, for example: `source_url = mlrun.get_sample_path("data/iris/iris_dataset.csv")`

```
# import the training function from the marketplace (hub://)
train = mlrun.import_function('hub://sklearn_classifier')

# Get a sample dataset path (points to MLRun data samples repository)
source_url = mlrun.get_sample_path("data/iris/iris_dataset.csv")

# run the function and specify input dataset path and some parameters (algorithm and␣
↪label column name)
train_run = train.run(name='train',
                      inputs={'dataset': source_url},
                      params={'model_pkg_class': 'sklearn.linear_model.
↪LogisticRegression',
                              'label_column': 'label'})
```

```
> 2022-02-06 21:42:08,132 [info] starting run train␣
↪uid=d0dd1b8a252e435b8a84c1ae8206918a DB=http://mlrun-api:8080
> 2022-02-06 21:42:08,302 [info] Job is running in the background, pod: train-wmwxv
> 2022-02-06 21:42:11,353 [warning] Server or client version is unstable. Assuming␣
↪compatible: {'server_version': '0.0.0+unstable', 'client_version': '0.0.0+unstable'}
> 2022-02-06 21:42:14,612 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-06 21:42:17,636 [info] run executed, status=completed
```

The run output above contains a link to the MLRun UI. Click it to inspect the various aspects of the jobs you run:

As well as their artifacts:

When running the function in a Jupyter notebook, the output cell for your function execution contains a table with run information — including the state of the execution, all inputs and parameters, and the execution results and artifacts.

Test the Model

Now that you have a trained model, you can test it: run a task that uses the test_classifier function from the function marketplace to run the selected trained model against the test dataset. The test dataset was returned from the training task (`train_run`) in the previous step.

```
test = mlrun.import_function('hub://test_classifier')
```

You can then run the function as part of your project, just as any other function that you have written yourself. To view the function documentation, call the `doc` method:

```
test.doc()
```

```
function: test-classifier
test a classifier using held-out or new data
default handler: test_classifier
entry points:
  test_classifier: Test one or more classifier models against held-out dataset
```

(continues on next page)

(continued from previous page)

```
Using held-out test features, evaluates the peformance of the estimated model

Can be part of a kubeflow pipeline as a test step that is run post EDA and
training/validation cycles
    context  - the function context, default=
    models_path(DataItem)  - artifact models representing a file or a folder, default=
    test_set(DataItem)  - test features and labels, default=
    label_column(str)  - column name for ground truth labels, default=
    score_method(str)  - for multiclass classification, default=micro
    plots_dest(str)  - dir for test plots, default=
    model_evaluator  - NOT IMPLEMENTED: specific method to generate eval, passed in␣
↪as string or available in this folder, default=None
    default_model(str)  - , default=model.pkl
    predictions_column(str)  - column name for the predictions column on the resulted␣
↪artifact, default=yscore
    model_update  - (True) update model, when running as stand alone no need in␣
↪update, default=True
```

Configure parameters for the test function (`params`), and provide the selected trained model from the train task as an input artifact (`inputs`)

```
test_run = test.run(name="test",
                    params={"label_column": "label"},
                    inputs={"models_path": train_run.outputs['model'],
                            "test_set": train_run.outputs['test_set']})
```

```
> 2022-02-06 21:42:30,469 [info] starting run test␣
↪uid=de5736813f13448789526b8793b34700 DB=http://mlrun-api:8080
> 2022-02-06 21:42:30,612 [info] Job is running in the background, pod: test-l4rjd
> 2022-02-06 21:42:33,761 [warning] Server or client version is unstable. Assuming␣
↪compatible: {'server_version': '0.0.0+unstable', 'client_version': '0.0.0+unstable'}
> 2022-02-06 21:42:35,972 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-06 21:42:36,852 [info] run executed, status=completed
```

Serve the Model

MLRun serving can take MLRun models or standard model files and produce managed, real-time, serverless functions using the Nuclio real-time serverless framework. Nuclio is built around data, I/O, and compute-intensive workloads, and is focused on performance and flexibility. Nuclio is also deeply integrated into the MLRun framework. See *MLRun Serving documentation* to learn more about the rich serving capabilities MLRun has to offer.

To deploy your model using the v2_model_server function, run the following code:

```
serve = mlrun.import_function('hub://v2_model_server')
model_name='iris'
serve.add_model(model_name, model_path=train_run.outputs['model'])
addr = serve.deploy()
```

```
> 2022-02-06 21:42:47,287 [info] Starting remote function deploy
2022-02-06 21:42:47  (info) Deploying function
2022-02-06 21:42:47  (info) Building
2022-02-06 21:42:48  (info) Staging files and preparing base images
2022-02-06 21:42:48  (info) Building processor image
2022-02-06 21:42:50  (info) Build complete
2022-02-06 21:42:55  (info) Function deploy complete
> 2022-02-06 21:42:56,081 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-quick-start-admin-v2-model-server.default-tenant.svc.
↪cluster.local:8080'], 'external_invocation_urls': ['quick-start-admin-v2-model-
↪server-quick-start-admin.default-tenant.app.yh41.iguazio-cd1.com/']}
```

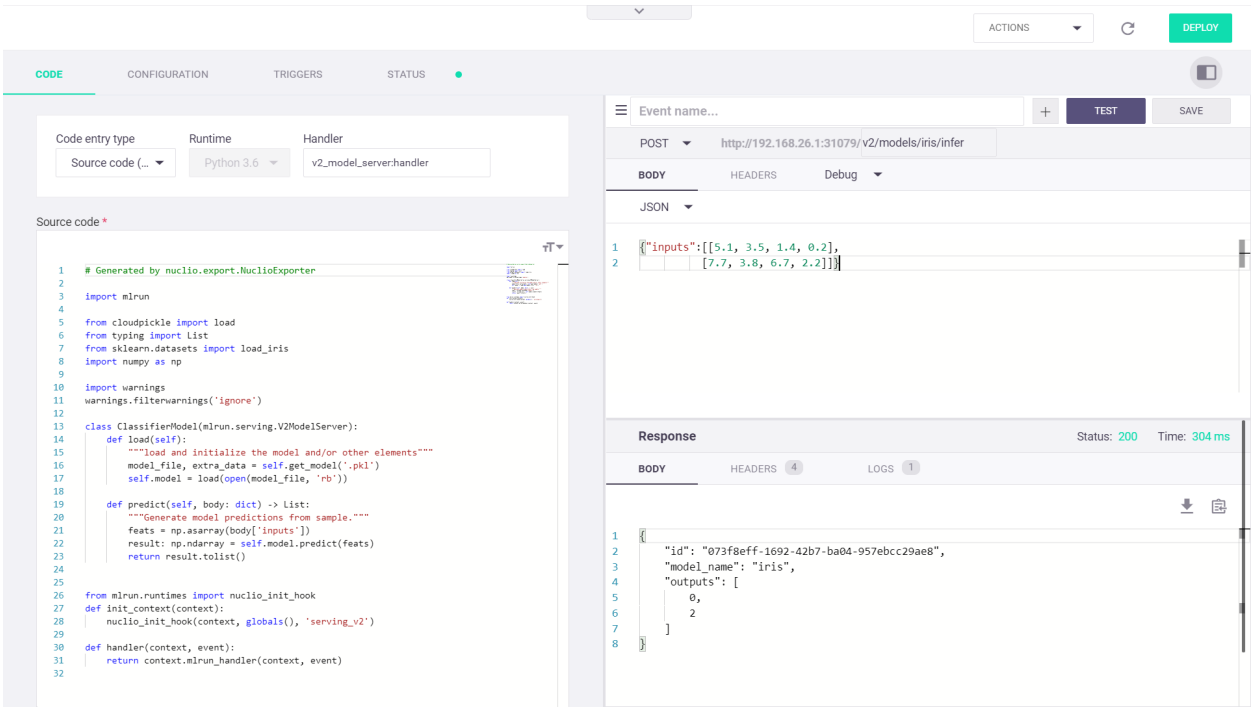
The `invoke` method enables to programmatically test the function.

```
import json

inputs = [[5.1, 3.5, 1.4, 0.2],
          [7.7, 3.8, 6.7, 2.2]]
my_data = json.dumps({'inputs': inputs})
serve.invoke(f'v2/models/{model_name}/infer', my_data)
```

```
> 2022-02-06 21:42:58,441 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-quick-start-admin-v2-model-server.default-tenant.svc.cluster.local:8080/v2/
↪models/iris/infer'}
```

```
{'id': 'e0bd75f2-bca4-4dd1-8237-3a101756c95a',
 'model_name': 'iris',
 'outputs': [0, 2]}
```

Open the Nuclio UI to view the function and test it.



For a more detailed walk-through, refer to the ***getting-started tutorial***.

2.1.4 Tutorials

Getting-Started Tutorial

This tutorial provides a hands-on introduction to using MLRun to implement a data science workflow and automate machine-learning operations (MLOps).

The tutorial covers MLRun fundamentals such as creation of projects and data ingestion and preparation, and demonstrates how to create an end-to-end machine-learning (ML) pipeline. MLRun is integrated as a default (pre-deployed) shared service in the Iguazio MLOps Platform.

You can see another example which demonstrate how to ***Convert Research Notebook (NYC Taxi) to Operational Pipeline***. For additional demos refer to MLRun demos repository at github.com/mlrun/demos.

In this tutorial you'll learn how to:

- Collect (ingest), prepare, and analyze data
- Train, deploy, and monitor an ML model
- Create and run an automated ML pipeline

You'll also learn about the basic concepts, components, and APIs that allow you to perform these tasks, including

- Setting up MLRun
- Creating and working with projects
- Creating, deploying and running MLRun functions
- Using MLRun to run functions, jobs, and full workflows
- Deploying a model to a serving layer using serverless functions

The tutorial is divided into four parts, each with a dedicated Jupyter notebook. The notebooks are designed to be run sequentially, as each notebook relies on the execution of the previous notebook:

Part 1: MLRun Basics

Part 1 of the getting-started tutorial introduces you to the basics of working with functions by using the MLRun open-source MLOps orchestration framework.

The tutorial takes you through the following steps:

1. *Installation and Setup*
2. *Creating a basic function and running it locally*
3. *Running the function on the cluster*
4. *Viewing jobs on the dashboard (UI)*
5. *Scheduling jobs*

By the end of this tutorial you'll learn how to

- Create a basic data-preparation MLRun function.
- Store data artifacts to be used and managed in a central database.
- Run your code on a distributed Kubernetes cluster without any DevOps overhead.
- Schedule jobs to run on the cluster.

Using MLRun Remotely

This tutorial is aimed at running your project from a local Jupyter Notebook service in the same environment in which MLRun is installed and running. However, as a developer you might want to develop your project from a remote location using your own IDE (such as a local Jupyter Notebook or PyCharm), and connect to the MLRun environment remotely. To learn how to use MLRun from a remote IDE, see *Setting a Remote Environment*.

Introduction to MLRun

MLRun is an open-source MLOps framework that enables faster development of production ready ML applications from data ingestion/preparation, model development to deployment of operational and real-time production pipelines.

MLRun offers a convenient abstraction layer and automation on top of powerful serverless technologies for feature engineering, machine learning, and deep learning. MLRun provides the following key benefits:

- **Rapid deployment** of code to production pipelines
- **Elastic scaling** of batch and real-time workloads
- **Feature management** — ingestion, preparation, and monitoring
- **Works anywhere** — your local IDE, multi-cloud, or on-prem

MLRun can be installed over Kubernetes or is preinstalled as a managed service in the Iguazio MLOps Platform.

For more detailed information, see the MLRun *Serverless Functions & Job Submission* documentation.

Step 1: Installation and Setup

For information on how to install and configure MLRun over Kubernetes, see the MLRun *installation guide*. To install the MLRun package, run `pip install mlrun` with the MLRun version that matches your MLRun service.

For Iguazio MLOps Platform Users

If your are using the Iguazio MLOps Platform, MLRun is available as a default (pre-deployed) shared service. You can run `!/User/align_mlrun.sh` to install the MLRun package or upgrade the version of an installed package. By default, the script attempts to download the latest version of the MLRun Python package that matches the version of the running MLRun service.

Kernel Restart

After installing or updating the MLRun package, restart the notebook kernel in your environment!

Initializing Your MLRun Environment

MLRun objects (runs, functions, workflows, features, artifacts, models, etc.) are associated with a `project` namespace. Projects can be stored or loaded from `git` repositories and can be created via API/UI.

For more information about MLRun projects, see the MLRun projects documentation.

Use the `get_or_create_project` MLRun method to create a new project, or fetch it from the DB/repository if it already exists. This method returns an MLRun project object.

Set the method's `name` parameter to your selected project name and set the `context` to the directory path hosting the project files/code (example: `./`). You can also optionally set the `user_project` parameter to `True` to automatically append the username of the running user to the project name specified in the `project` parameter, resulting in a `<project>-<username>` project name; this is useful for avoiding project-name conflicts among different users.

You can optionally pass additional parameters to `get_or_create_project`, as detailed in the *MLRun API reference*. For example, you can set the `url` parameter to specify a remote `git` url (hosting the project code). In *Part 4* of this tutorial you'll learn how project objects enable packaging, pipeline composition, and simple integration with `git` and CI/CD frameworks.

Run the following code to initialize your MLRun environment to use a "getting-started-tutorial-<username>" project and store the project artifacts in the default artifacts path:

```
from os import path
import mlrun

# Set the base project name
project_name_base = 'getting-started'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_
↪project=True)

# Display the current project name
project_name = project.metadata.name
print(f'Full project name: {project_name}')
```

```
> 2022-02-06 21:58:06,320 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2022-02-06 21:58:13,183 [warning] Server or client version is unstable. Assuming␣
↪compatible: {'server_version': '0.0.0+unstable', 'client_version': '0.0.0+unstable'}
> 2022-02-06 21:58:13,217 [info] loaded project getting-started from MLRun DB
Full project name: getting-started-admin
```

Step 2: Creating a Basic Function

This step introduces you to MLRun functions and artifacts and walks you through the process of converting a local function to an MLRun function.

Defining a Local Function

The following example code defines a data-preparation function (`prep_data`) that reads (ingests) a CSV file from the provided source URL into a pandas DataFrame. It prepares ("cleans") the data by changing the type of the categorical data in the specified label column, and returns the DataFrame and its length.

```
import pandas as pd

# Fetch and clean a dataset through ingestion
def prep_data(source_url, label_column):
    df = pd.read_csv(source_url)
    df[label_column] = df[label_column].astype('category').cat.codes
    return df, df.shape[0]
```

Creating and Running Your First MLRun Function

MLRun Functions

The MLRun jobs and pipelines run over serverless functions. These functions can include the function code and specification ("spec"). The spec contains metadata for configuring related operational aspects, such as the image, required packages, CPU/memory/GPU resources, storage, and the environment. The different serverless runtime engines automatically transform the function code and spec into fully managed and elastic services that run over Kubernetes. Functions are versioned and can be generated from code or notebooks, or loaded from a marketplace.

MLRun supports batch functions (for data processing, training, etc.) or real-time functions (for serving, APIs, and stream processing).

To work with functions you need to be familiar with the following function components:

- **Context** — Functions can be built from standard code files or function handlers. To gain the maximum value we use the job `context` object inside our code. This provides access to parameters, data, secrets, etc., as well as log text, files, artifacts, and labels.
  - If `context` is specified as the first parameter in the function signature, MLRun injects the current job context into it.

  - Alternatively, you can obtain the context object using the MLRun `get_or_create_ctx()` method, without changing the function.

- **Parameters and inputs** — You can pass `parameters` (arguments) or data `inputs` (such as datasets, feature-vectors, models, or files) to the functions through the `run` method.
  - Inside the function you can access the parameters/inputs by simply adding them as parameters to the function or you can get them from the context object (using `get_param()` and `get_input()`).
  - You can use MLRun *Iterative Hyper-Param jobs* to run functions with different parameter and input combinations.
  - Various data objects (files, tables, models, etc.) are passed to the function as *`DataItem` objects*. Data items objects abstract away the data backend implementation, provide a set of convenience methods (`.as_df`, `.get`, `.show`, ..), and enable auto logging/versioning of data and metadata.

For more information see the following MLRun documentation:

- *Functions & Job Submission*
- *Data Items*
- *Artifacts*
- *Hyper-Param and Iterative jobs*

MLRun Function Code

The step redefines this function and converts it to an MLRun function that leverages MLRun:

- Read the data
- Log the data to the MLRun database

The MLRun function has the following parameter changes compared to the original local function:

- To effectively run your code in MLRun, you need to add a `context` parameter to your function (or alternatively, get the context by using *`get_or_create_ctx()`*). This allows you to log and retrieve information related to the function's execution.
- The tutorial example sets the `source_url` parameter to `mlrun.DataItem` to send a data item as input when the function is called (using the `inputs` parameter).

The following code demonstrates how to redefine your local data-preparation function to make it compatible with MLRun, and then convert the local notebook code into an MLRun function.

This example tutorial function code works as follows:

- Prepare (clean) the data in the same way as in the local-function implementation in the previous step.
- Obtain a pandas DataFrame from the `source_url` data item by calling the `as_df` method.
- Record the data length (number of rows) using the `log_result` method. This method records (logs) the values of standard function variables (for example, int, float, string, and list).
- Log the data-set artifact using the `log_dataset` method. This method saves and logs function data items and related metadata (logs function artifacts).

```
# mlrun: start-code
```

```
import mlrun
def prep_data(context, source_url: mlrun.DataItem, label_column='label'):

    # Convert the DataItem to a pandas DataFrame
    df = source_url.as_df()
    df[label_column] = df[label_column].astype('category').cat.codes

    # Log the DataFrame size after the run
    context.log_result('num_rows', df.shape[0])

    # Store the dataset in your artifacts database
    context.log_dataset('cleaned_data', df=df, index=False, format='csv')
```

```
# mlrun: end-code
```

Converting the Notebook Code to a Function

MLRun annotations are used to identify the code that needs to be converted into an MLRun function. They provide non-intrusive hints that indicate which parts of your notebook are the code of the function.

- The `# mlrun: ignore` annotation identifies code that is not included in the MLRun function (such as prints, plots, tests, and debug code).
- The `# mlrun: start-code` and `# mlrun: end-code` annotations identify code to be converted to an MLRun function: everything before the `start-code` annotation and after the `end-code` annotation is ignored, and only code between these two annotations is converted. These annotations are used in the tutorial notebook instead of adding the `ignore` annotation to all cells that shouldn't be converted.

For more information on how to use MLRun code annotations, see *converting notebook code to a function*.

The following code uses the `code_to_function` MLRun method to convert your local `prep_data` function code to a `data_prep_func` MLRun function. If the `filename` parameter is not specified it searches the code in the current notebook. Alternatively you can specify a path to a `.py` or `.ipynb` file by setting the `filename` parameter.

The `kind` parameter of the `code_to_function` method determines the engine for running the code. MLRun allows running function code using different engines, such as Python, Spark, MPI, Nuclio, and Dask. The following example sets the `kind` parameter to `job` in order to run the code as a Python process ("job").

```
# Convert the local prep_data function to an MLRun project function
data_prep_func = mlrun.code_to_function(name='prep_data', kind='job', image='mlrun/
↪mlrun')
```

Using custom packages or code from a source repository

This tutorial uses the built-in mlrun images (`mlrun/mlrun`). You can also build custom function containers dynamically and include user specified packages or code from a repository. You can set advanced build attributes using the `function.build_config()` method or simply specify package requirements in the `code_to_function()` call.

To build the function container from the specified requirements, run `function.deploy()`, which creates a custom container for the function. Once the function is built you can run it.

Running the MLRun Function Locally

Now you're ready to run your MLRun function (`data_prep_func`). This example uses the `run` MLRun method and sets its `local` parameter to `True` to run the function code locally within your Jupyter pod. This means that the function uses the environment variables, volumes, and image that are running in this pod.

> **Note:** When running a function locally, the function code is saved only in a temporary local directory and not in your project's ML functions code repository. In the next step of this tutorial you'll run the function on a cluster, which automatically saves the function object in the project.

The execution results are stored in the MLRun database. The tutorial example sets the following function parameters:

- `name`—the job name
- `handler`—the name of the function handler
- `inputs`—the data-set URL

The example uses a CSV file from MLRun demo data repository The default location is in the Wasabi object-store service).

You can also use the function to ingest data in formats other than CSV, such as Parquet, without modifying the code.

---

training a model in an air-gapped site

If you are working in MLRun:

1. Download your data file and save it locally.
2. Run: `import os os.environ["env_name"] = 1`
3. Use the same command for the sample data set path, for example: `source_url = mlrun.get_sample_path("data/iris/iris_dataset.csv")`

If your system is integrated with an MLOps Platform:

1. Download your data file and save it locally.
2. In the UI, click the settings icon () in the top-right of the header in any page to open the **Settings** dialog.
3. Click **Environment variables | Create a new environment variable**, and set an environmental variable: SAMPLE_DATA_SOURCE_URL_PREFIX = the relative path to locally-stored data. For example: `/v3io/bigdata/path/...`
4. Use the same command for the sample data set path, for example: `source_url = mlrun.get_sample_path("data/iris/iris_dataset.csv")`

```
# Set the source-data URL
source_url = mlrun.get_sample_path("data/iris/iris.data.raw.csv")
```

```
# Run the `data_prep_func` MLRun function locally
prep_data_run = data_prep_func.run(name='prep_data',
                                   handler=prep_data,
                                   inputs={'source_url': source_url},
                                   local=True)
```

```
> 2022-02-06 22:01:18,436 [info] starting run prep_data␣
↪uid=0872125626964b3f99089df8bf928ff3 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-06 22:01:18,912 [info] run executed, status=completed
```

Getting Information About the Run Object

Every run object that's returned by the MLRun `run` method has the following methods:

- `uid` — returns the unique ID.
- `state` — returns the last known state.
- `show` — shows the latest job state and data in a visual widget (with hyperlinks and hints).
- `outputs` — returns a dictionary of the run results and artifact paths.
- `logs` — returns the latest logs. Use `Watch=False` to disable the interactive mode in running jobs.
- `artifact` — returns full artifact details for the provided key.
- `output` — returns a specific result or an artifact path for the provided key.
- `to_dict`, `to_yaml`, `to_json` — converts the run object to a dictionary, YAML, or JSON format (respectively).

```
# example
prep_data_run.state()
```

```
'completed'
```

```
# example
prep_data_run.outputs['cleaned_data']
```

```
'store://artifacts/getting-started-admin/prep_data_cleaned_
↪data:0872125626964b3f99089df8bf928ff3'
```

Reading the Output

The data-set location is returned in the `outputs` field. Therefore, you can get the location by calling `prep_data_run.outputs['cleaned_data']` and using `run.get_dataitem` to get the data set itself.

```
dataset = prep_data_run.artifact('cleaned_data').as_df()
```

```
dataset.head()
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   label
0      0
1      0
2      0
3      0
4      0
```

Saving the Artifacts in Run-Specific Paths

In the previous steps, each time the function was executed its artifacts were saved to the same directory, overwriting the existing artifacts in this directory. However, you can also select to save the run results (source-data file) to a different directory for each job execution. This is done by setting the artifact path and using the unique run-ID parameter (`{{run.uid}}`) in the path. Under the artifact path you should be able to see the source-data file in a new directory whose name is derived from the unique run ID.

```
prep_data_run = data_prep_func.run(name='prep_data',
                        handler=prep_data,
                        inputs={'source_url': source_url},
                        local=True,
                        artifact_path=path.join(mlrun.mlconf.artifact_path, '{{run.
↪uid}}'))
```

```
> 2022-02-06 22:02:55,268 [info] starting run prep_data␣
↪uid=0480c2b438da4bf7a84f3acee45c5546 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-06 22:02:55,610 [info] run executed, status=completed
```

Step 3: Running the Function on a Cluster

You can also run MLRun functions on the cluster itself, as opposed to running them locally, like in the previous steps. Running a function on the cluster allows you to leverage the cluster's resources and run more resource-intensive workloads. MLRun helps you to easily run your code without the hassle of creating configuration files and build images. To run an MLRun function on a cluster, change the `run` method's `local` flag to `False`.

```
prep_data_run = data_prep_func.run(name='prep_data',
                                   handler='prep_data',
                                   inputs={'source_url': source_url},
                                   local=False)
```

```
> 2022-02-06 22:04:07,284 [info] starting run prep_data␣
↪uid=24ecd12a877545eea35acd90910ae2b8 DB=http://mlrun-api:8080
> 2022-02-06 22:04:07,445 [info] Job is running in the background, pod: prep-data-
↪5cdzz
> 2022-02-06 22:04:10,690 [warning] Server or client version is unstable. Assuming␣
↪compatible: {'server_version': '0.0.0+unstable', 'client_version': '0.0.0+unstable'}
> 2022-02-06 22:04:10,987 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-06 22:04:13,663 [info] run executed, status=completed
```

Step 4: Viewing Jobs on the Dashboard (UI)

On the **Projects** dashboard page, select your project and then navigate to the project's jobs and workflow page by selecting the relevant link. For this tutorial, after running the `prep_data` method twice, you should see three records with types local (**<>**) and job. In this view you can track all jobs running in your project and view detailed job information. Select a job name to display tabs with additional information such as an input data set, artifacts that were generated by the job, and execution results and logs.

Step 5: Scheduling Jobs

To schedule a job, you can set the `schedule` parameter of the `run` method. The scheduling is done by using a crontab format.

You can also schedule jobs from the dashboard. On the Projects > Jobs and Workflows page, you can create a new job using the **New Job** wizard. At the end of the wizard you can set the job scheduling. In the following example, the job is set to run every 30 minutes.

```
data_prep_func
prep_data_run = data_prep_func.run(name='prep_data',
```

(continues on next page)

(continued from previous page)

```
                                    handler='prep_data',
                                    inputs={'source_url': source_url},
                                    local=False,
                                    schedule='*/30 * * * *')
```

```
> 2022-02-06 22:04:26,151 [info] starting run prep_data␣
↪uid=1371ba389bbf4a4dad91a072762c95b6 DB=http://mlrun-api:8080
> 2022-02-06 22:04:26,358 [info] task scheduled, {'schedule': '*/30 * * * *', 'project
↪': 'getting-started-admin', 'name': 'prep_data'}
```

View Scheduled Jobs on the Dashboard (UI)

You can also see your scheduled jobs on your project's **Jobs | Schedule** dashboard page.

Deleting Scheduled Jobs

When you no longer need to run the scheduled jobs, remove them in the UI or by using the `get_run_db().delete_schedule` MLRun method to delete the job-schedule objects that you created.

```
mlrun.get_run_db().delete_schedule(project_name, 'prep_data')
```

Done!

Congratulation! You've completed Part 1 of the MLRun getting-started tutorial. Proceed to ***Part 2: Model Training*** to learn how to train an ML model.

Part 2: Training an ML Model

This part of the MLRun getting-started tutorial walks you through the steps for training a machine-learning (ML) model, including data exploration and model testing.

The tutorial consists of the following steps:

1. *Setup and Configuration*
2. *Creating a training function*
3. *Exploring the data with an MLRun marketplace function*
4. *Testing your model*

By the end of this tutorial you'll learn how to

- Create a training function, store models, and track experiments while running them.
- Use artifacts as inputs to functions.
- Leverage the MLRun functions marketplace.

- View plot artifacts.

Prerequisites

The following steps are a continuation of the previous part of this getting-started tutorial and rely on the generated outputs. Therefore, make sure to first run *part 1* of the tutorial.

Step 1: Setup and Configuration

Initializing Your MLRun Environment

Use the `get_or_create_project` MLRun method to create a new project or fetch it from the DB/repository if it already exists. Set the `project` and `user_project` parameters to the same values that you used in the call to this method in the Part 1: MLRun Basics tutorial notebook.

```
import mlrun

# Set the base project name
project_name_base = 'getting-started'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_
↪project=True)
```

```
> 2022-02-08 19:46:23,537 [info] loaded project getting-started from MLRun DB
```

Marking The Beginning of Your Function Code

The following code uses the `# mlrun: start-code` marker comment annotation to instruct MLRun to start processing code only from this location.

> **Note:** You can add code to define function dependencies and perform additional configuration after the `# mlrun: start-code` marker and before *the `# mlrun: end-code` marker*.

```
# mlrun: start-code
```

Step 2: Creating a Training Function

Training functions generate models and various model statistics, we want to store the models along will all the relevant data, metadata and measurements. This can be achieved automatically using MLRun `auto logging` capabilities.

MLRun can apply all the MLOps functionality by simply using the framework specific `apply_mlrun()` method which manages the training process and automatically logs all the framework specific model details, data, metadata and metrics.

To log the training results and store a model named `my_model`, we simply need to add the following lines:

```
from mlrun.frameworks.sklearn import apply_mlrun
apply_mlrun(model, context, model_name='my_model', x_test=X_test, y_test=y_test)
```

The training job will automatically generate a set of results and versioned artifacts (run `train_run.outputs` to view the job outputs):

```
{'accuracy': 1.0,
 'f1_score': 1.0,
 'precision_score': 1.0,
 'recall_score': 1.0,
 'auc-micro': 1.0,
 'auc-macro': 1.0,
 'auc-weighted': 1.0,
 'feature-importance': 'v3io:///projects/getting-started-admin/artifacts/feature-
↪importance.html',
 'test_set': 'store://artifacts/getting-started-admin/train-iris-train_iris_test_
↪set:86fd0a3754c34f75b8afc5c2464959fc',
 'confusion-matrix': 'v3io:///projects/getting-started-admin/artifacts/confusion-
↪matrix.html',
 'roc-curves': 'v3io:///projects/getting-started-admin/artifacts/roc-curves.html',
 'my_model': 'store://artifacts/getting-started-admin/my_
↪model:86fd0a3754c34f75b8afc5c2464959fc'}
```

```
from sklearn import ensemble
from sklearn.model_selection import train_test_split
from mlrun.frameworks.sklearn import apply_mlrun
import mlrun
```

```
def train_iris(dataset: mlrun.DataItem, label_column: str):

    # Initialize our dataframes
    df = dataset.as_df()
    X = df.drop(label_column, axis=1)
    y = df[label_column]

    # Train/Test split Iris data-set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

    # Pick an ideal ML model
    model = ensemble.RandomForestClassifier()

    # Wrap our model with Mlrun features, specify the test dataset for analysis and␣
↪accuracy measurements
    apply_mlrun(model=model, model_name='my_model', x_test=X_test, y_test=y_test)
```

(continues on next page)

(continued from previous page)

```
    # Train our model
    model.fit(X_train, y_train)
```

Marking The End of Your Function Code

The following code uses the `# mlrun: end-code` marker code annotation to mark the end of the code section that should be converted to your MLRun function (which began with *the `# mlrun: start-code` annotation*) and instruct MLRun to stop parsing the notebook at this point.

Important: Don't remove the start-code and end-code annotation cells.

```
# mlrun: end-code
```

Converting the Code to an MLRun Function

Use the MLRun `code_to_function` method to convert the selected portions of your notebook code into an MLRun function in your project — a function object with embedded code, which can run on the cluster.

The following code converts the code of your local `train_iris` function, which is defined within `# mlrun: start-code` and `# mlrun: end-code` annotations that mark the notebook code to convert (see the previous code cells), into into a `train_iris_func` MLRun function. The function will be stored and run under the current project (which was specified in the `get_or_create_project` method above).

The code sets the following `code_to_function` parameters:

- `name` — the name of the new MLRun function (`train_iris`).
- `handler` — the name of the function-handler method (`train_iris`; the default is `main`).
- `kind` — the function's runtime type (`job` for a Python process).
- `image` — the name of the container image to use for running the job — "mlrun/mlrun". This image contains the basic machine-learning Python packages (such as scikit-learn).

```
train_iris_func = mlrun.code_to_function(name='train_iris',
                                         handler='train_iris',
                                         kind='job',
                                         image='mlrun/mlrun')
```

Running the Function on a Cluster

```
# Our dataset location (uri)
dataset = project.get_artifact_uri('prep_data_cleaned_data')
```

```
#train_iris_func.spec.image_pull_policy = "Always"
```

```
train_run = train_iris_func.run(inputs={'dataset': dataset},
                                params={'label_column': 'label'},local=True)
```

```
> 2022-02-08 19:58:17,705 [info] starting run train-iris-train_iris␣
↪uid=9d6806aec3134110bd5358479152aa5d DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-08 19:58:19,385 [info] run executed, status=completed
```

Reviewing the Run Output

You can view extensive run information and artifacts from Jupyter Notebook and the MLRun dashboard, as well as browse the project artifacts from the dashboard.

The following code extracts and displays the model from the training-job outputs.

```
train_run.outputs['model']
```

```
'store://artifacts/getting-started-admin/my_model:86fd0a3754c34f75b8afc5c2464959fc'
```

Your project's artifacts directory contains the results for the executed training job. The **plots** subdirectory has HTML output artifacts for the selected run iteration; (the **data** subdirectory contains the artifacts for the test data set).

Use the following code to extract and display information from the run object — the accuracy that was achieved with the model, and the `confusion` and `roc` HTML output artifacts for the optimal run iteration.

```
print(f'Accuracy: {train_run.outputs["accuracy"]}')
```

```
Accuracy: 1.0
```

```
# Display HTML output artifacts
train_run.artifact('confusion-matrix').show()
```

```
<IPython.core.display.HTML object>
```

```
train_run.artifact('feature-importance').show()
```

```
<IPython.core.display.HTML object>
```

Exploring the Data with pandas DataFrames

Run the following code to use pandas DataFrames to read your data set, extract some basic statistics, and display them.

```
# Read your data set
df = train_run.artifact('test_set').as_df()
df.head()
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                6.1               2.8                4.7               1.2
1                5.7               3.8                1.7               0.3
2                7.7               2.6                6.9               2.3
3                6.0               2.9                4.5               1.5
4                6.8               2.8                4.8               1.4

   label
0      1
1      0
2      2
3      1
4      1
```

```
# Display statistics grouped by label
df.groupby(['label']).describe()
```

```
      sepal length (cm)                                                \
                  count      mean       std  min   25%   50%    75%  max
label
0                  10.0  5.070000  0.346570  4.7  4.80  4.95  5.325  5.7
1                   9.0  6.011111  0.391933  5.6  5.70  6.00  6.200  6.8
2                  11.0  6.781818  0.551032  6.1  6.45  6.70  6.850  7.9

      sepal width (cm)            ... petal length (cm)      petal width (cm)  \
                 count      mean  ...                75%  max            count
label                             ...
0                 10.0  3.330000  ...               1.60  1.7             10.0
1                  9.0  2.766667  ...               4.70  4.8              9.0
2                 11.0  3.000000  ...               5.85  6.9             11.0


           mean       std  min  25%   50%  75%  max
label
0      0.250000  0.108012  0.1  0.2  0.25  0.3  0.4
1      1.344444  0.166667  1.1  1.2  1.30  1.5  1.6
2      2.118182  0.194001  1.8  2.0  2.20  2.3  2.3

[3 rows x 32 columns]
```

Step 3: Exploring the Data with an MLRun Marketplace Function

You can perform further data exploration by leveraging **the MLRun functions marketplace** (a.k.a. "the MLRun functions hub"). This marketplace is a centralized location for open-source contributions of function components that are commonly used in machine-learning development. The location of the marketplace is centrally configured . By default, it points to the mlrun/functions GitHub repository.

This step uses the `describe` marketplace function, which performs data exploration on a provided data set. The function is used to extract information from your data set, analyze it, and visualize relevant information in different ways.

Adding an Exploration Function

Use the `import_function` MLRun method, which adds or updates a function object in a project, to load the `describe` MLRun marketplace function into a new `describe` project function. The tutorial code sets the first `import_function` parameter — `url` — which identities the function to load.

> **Note:** MLRun supports multiple types of URL formats. The example uses the `hub://<function name>` format to point to the `describe` function-code directory in the MLRun functions marketplace (`'hub://describe'`). You can add `:<tag>` to this syntax to load a specific function tag — `hub://<function_name>:<tag>`; replace the `<function name>` and `<tag>` placeholders with the desired function name and tag.

```
describe = mlrun.import_function('hub://describe')
```

Viewing the Function Documentation

Use the `doc` method to view the embedded documentation of the `describe` function.

```
describe.doc()
```

```
function: describe
describe and visualizes dataset stats
default handler: summarize
entry points:
  summarize: Summarize a table
    context(MLClientCtx)  - the function context, default=
    table(DataItem)  - MLRun input pointing to pandas dataframe (csv/parquet file␣
↪path), default=
    label_column(str)  - ground truth column label, default=None
    class_labels(List[str])  - label for each class in tables and plots, default=[]
    plot_hist(bool)  - (True) set this to False for large tables, default=True
    plots_dest(str)  - destination folder of summary plots (relative to artifact_
↪path), default=plots
    update_dataset  - when the table is a registered dataset update the charts in-
↪place, default=False
```

Running the Exploration Function

Run the following code to execute the `describe` project function as a Kubernetes job by using the MLRun `run` method. The returned run object is stored in a `describe_run` variable.

The location of the data set is the only input that you need to provide. This information is provided as a `table` input artifact that points to the `table_set` output artifact of the `train_run` job that you ran in the previous step.

```
describe_run = describe.run(params={'label_column': 'label'},
                            inputs={"table": train_run.outputs['test_set']})
```

```
> 2022-02-08 19:51:32,555 [info] starting run describe-summarize␣
↪uid=d99feda4fcaa4d70b32e5032c7a7d41b DB=http://mlrun-api:8080
> 2022-02-08 19:51:32,719 [info] Job is running in the background, pod: describe-
↪summarize-6mxsz
> 2022-02-08 19:51:41,994 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-08 19:51:52,084 [info] run executed, status=completed
```

Reviewing the Run Output

The output cell for your code execution contains a run-information table. You can also view run information in the MLRun dashboard; see the output-review information in Step 2, only this time look for the `describe-summarize` job and related artifacts.

The `describe` function generates three HTML output artifacts, which provide visual insights for your data set — `histograms`, `imbalance`, and `correlation`. The artifacts are stored as HTML files in your project's artifacts directory, under **<project artifacts path>/jobs/plots/**. The following code displays the artifact files in the notebook.

```
# Display the `histograms` artifact
describe_run.artifact('histograms').show()
```

```
<IPython.core.display.HTML object>
```

```
# Display the `imbalance` artifact
describe_run.artifact('imbalance').show()
```

```
<IPython.core.display.HTML object>
```

```
# Display the `correlation` artifact
describe_run.artifact('correlation').show()
```

```
<IPython.core.display.HTML object>
```

Step 4: Testing Your Model

Now that you have a trained model, you can test it: run a task that uses the `test_classifier` marketplace function to run the selected trained model against the test data set, as returned for the training task (`train`) in the previous step.

Adding a Test Function

Run the following code to add to your project a `test` function that uses the `test_classifier` marketplace function code, and create a related `test` function object.

```
test = mlrun.import_function('hub://test_classifier')
```

Running a Model-Testing Job

Configure parameters for the test function (`params`), and provide the selected trained model from the `train_run` job as an input artifact (`inputs`).

```
test_run = test.run(name="test",
                    params={"label_column": "label"},
                    inputs={"models_path": train_run.outputs['model'],
                            "test_set": train_run.outputs['test_set']})
```

```
> 2022-02-08 19:51:52,306 [info] starting run test␣
↪uid=529d6b7a24f34f6b8071005365c254b0 DB=http://mlrun-api:8080
> 2022-02-08 19:51:52,460 [info] Job is running in the background, pod: test-b29jj
> 2022-02-08 19:51:58,625 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-02-08 19:52:01,766 [info] run executed, status=completed
```

Reviewing the Run Output

Check the output information for your run in Jupyter Notebook and on the MLRun dashboard.

Use the following code to display information from the run object — the accuracy of the model, and the `confusion` and `roc` HTML output artifacts.

```
print(f'Test Accuracy: {test_run.outputs["accuracy"]}')
```

```
Test Accuracy: 1.0
```

```
test_run.artifact('confusion-matrix').show()
```

```
<IPython.core.display.HTML object>
```

```
test_run.artifact('roc-multiclass').show()
```

```
<IPython.core.display.HTML object>
```

Done!

Congratulation! You've completed Part 2 of the MLRun getting-started tutorial. Proceed to ***Part 3: Model Serving*** to learn how to deploy and server your model using a serverless function.

Part 3: Serving an ML Model

This part of the MLRun getting-started tutorial walks you through the steps for implementing ML model serving using MLRun serving and Nuclio runtimes. The tutorial walks you through the steps for creating, deploying, and testing a model-serving function ("a serving function" a.k.a. "a model server").

MLRun serving can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. Nuclio is a high-performance open-source "serverless" framework that's focused on data, I/O, and compute-intensive workloads.

Simple model serving classes can be written in Python or be taken from a set of pre-developed ML/DL classes. The code can handle complex data, feature preparation, and binary data (such as images and video files). The Nuclio serving engine supports the full model-serving life cycle; this includes auto generation of microservices, APIs, load balancing, model logging and monitoring, and configuration management.

MLRun serving supports more advanced real-time data processing and model serving pipelines. For more details and examples, see the *MLRun Serving Graphs* documentation.

The tutorial consists of the following steps:

1. *Setup and Configuration* — load your project
2. *Writing A Simple Serving Class*
3. *Deploying the Model-Serving Function (Service)*
4. *Using the Live Model-Serving Function*
5. *Viewing the Nuclio Serving Function on the Dashboard*

By the end of this tutorial you'll learn how to

- Create model-serving functions.
- Deploy models at scale.
- Test your deployed models.

Prerequisites

The following steps are a continuation of the previous parts of this getting-started tutorial and rely on the generated outputs. Therefore, make sure to first run parts 1—2 of the tutorial.

Step 1: Setup and Configuration

Importing Libraries

Run the following code to import required libraries:

```
import mlrun
```

Initializing Your MLRun Environment

Use the `get_or_create_project` MLRun method to create a new project or fetch it from the DB/repository if it already exists. Set the `project` and `user_project` parameters to the same values that you used in the call to this method in the Part 1: MLRun Basics tutorial notebook.

```
# Set the base project name
project_name_base = 'getting-started'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_
↪project=True)
```

```
> 2022-02-08 19:57:17,874 [info] loaded project getting-started from MLRun DB
```

Step 2: Writing A Simple Serving Class

The serving class is initialized automatically by the model server. All you need is to implement two mandatory methods:

- `load` — downloads the model files and loads the model into memory. This can be done either synchronously or asynchronously.
- `predict` — accepts a request payload and returns prediction (inference) results.

For more detailed information on serving classes, see the MLRun documentation.

The following code demonstrates a minimal scikit-learn (a.k.a. sklearn) serving-class implementation:

```
# mlrun: start-code
```

```
from cloudpickle import load
import numpy as np
from typing import List
import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

```
# mlrun: end-code
```

Step 3: Deploying the Model-Serving Function (Service)

To provision (deploy) a function for serving the model ("a serving function") you need to create an MLRun function of type `serving`. You can do this by using the `code_to_function` MLRun method from a web notebook, or by importing an existing serving function or template from the MLRun functions marketplace.

Converting a Serving Class to a Serving Function

The following code converts the `ClassifierModel` class that you defined in the previous step to a serving function. The name of the class to be used by the serving function is set in `spec.default_class`.

```
serving_fn = mlrun.code_to_function('serving', kind='serving',image='mlrun/mlrun')
serving_fn.spec.default_class = 'ClassifierModel'
```

Add the model created in previous notebook by the training function

```
model_file = project.get_artifact_uri('my_model')
serving_fn.add_model('my_model',model_path=model_file)
```

```
<mlrun.serving.states.TaskStep at 0x7fec77d70390>
```

Testing Your Function Locally

To test your function locally, create a test server (mock server) and test it with sample data.

```
my_data = '''{"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}'''
```

```
server = serving_fn.to_mock_server()
server.test("/v2/models/my_model/infer", body=my_data)
```

```
> 2022-02-08 19:58:44,716 [info] model my_model was loaded
> 2022-02-08 19:58:44,716 [info] Loaded ['my_model']
```

```
{'id': '97ed827394c24011bc2d95a001f7c372',
 'model_name': 'my_model',
 'outputs': [0, 2]}
```

Building and Deploying the Serving Function

Use the `deploy` method of the MLRun serving function to build and deploy a Nuclio serving function from your serving-function code.

```
function_address = serving_fn.deploy()
```

```
> 2022-02-08 19:58:50,645 [info] Starting remote function deploy
2022-02-08 19:58:51  (info) Deploying function
2022-02-08 19:58:51  (info) Building
2022-02-08 19:58:52  (info) Staging files and preparing base images
2022-02-08 19:58:52  (info) Building processor image
2022-02-08 19:59:47  (info) Build complete
> 2022-02-08 19:59:52,828 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-getting-started-admin-serving.default-tenant.svc.cluster.
↪local:8080'], 'external_invocation_urls': ['getting-started-admin-serving-getting-
↪started-admin.default-tenant.app.yh41.iguazio-cd1.com/']}
```

Step 4: Using the Live Model-Serving Function

After the function is deployed successfully, the serving function has a new HTTP endpoint for handling serving requests. The example tutorial serving function receives HTTP prediction (inference) requests on this endpoint; calls the `infer` method to get the requested predictions; and returns the results on the same endpoint.

```
print (f'The address for the function is {function_address} \n')

!curl $function_address
```

```
The address for the function is http://getting-started-admin-serving-getting-started-
↪admin.default-tenant.app.yh41.iguazio-cd1.com/

{"name": "ModelRouter", "version": "v2", "extensions": []}
```

Testing the Model Server

Test your model server by sending data for inference. The `invoke` serving-function method enables programmatic testing of the serving function. For model inference (predictions), specify the model name followed by `infer`:

```
/v2/models/{model_name}/infer
```

For complete model-service API commands — such as for list models (`models`), get model health (`ready`), and model explanation (`explain`) — see the MLRun documentation.

```
serving_fn.invoke('/v2/models/my_model/infer', my_data)
```

```
> 2022-02-08 19:59:53,584 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-getting-started-admin-serving.default-tenant.svc.cluster.local:8080/v2/
↪models/my_model/infer'}
```

```
{'id': '3fe451d7-20d8-4813-a7d4-292ceeaca98c',
 'model_name': 'my_model',
 'outputs': [0, 2]}
```

Step 5: Viewing the Nuclio Serving Function on the Dashboard

On the **Projects** dashboard page, select the project and then select "Real-time functions (Nuclio)".

Done!

Congratulation! You've completed Part 3 of the MLRun getting-started tutorial. Proceed to ***Part 4: ML Pipeline*** to learn how to create an automated pipeline for your project.

Part 4: Projects and Automated ML Pipeline

This part of the MLRun getting-started tutorial walks you through the steps for working with projects, source control (git), and automating the ML pipeline.

MLRun Project is a container for all your work on a particular activity: all the associated code, functions, jobs/workflows and artifacts. Projects can be mapped to `git` repositories to enable versioning, collaboration, and CI/CD.

You can create project definitions using the SDK or a yaml file and store those in MLRun DB, file, or archive. Once the project is loaded you can run jobs/workflows which refer to any project element by name, allowing separation between configuration and code. See *Create and load projects* for details.

Projects contain `workflows` that execute the registered functions in a sequence/graph (DAG), and which can reference project parameters, secrets and artifacts by name. MLRun currently supports two workflow engines, `local` (for simple tasks) and Kubeflow Pipelines (for more complex/advanced tasks). MLRun also supports a real-time workflow engine (see *MLRun serving graphs*).

> **Note**: The Iguazio MLOps Platform has a default (pre-deployed) shared Kubeflow Pipelines service (`pipelines`).

An ML Engineer can gather the different functions created by the Data Engineer and Data Scientist and create this automated pipeline.

The tutorial consists of the following steps:

1. *Setting up Your Project*
2. *Updating Project and Function Definitions*
3. *Defining and Saving a Pipeline Workflow*
4. *Registering the Workflow*
5. *Running A Pipeline*
6. *Viewing the Pipeline on the Dashboard (UI)*
7. *Invoking the Model*

By the end of this tutorial you'll learn how to:

- Create an operational pipeline using previously defined functions.
- Run the pipeline and track the pipeline results.

Prerequisites

The following steps are a continuation of the previous parts of this getting-started tutorial and rely on the generated outputs. Therefore, make sure to first run parts *1*—*3* of the tutorial.

Step 1: Setting Up Your Project

To run a pipeline, you first need to create a Python project object and import the required functions for its execution.

Create a project by using one of:

- the `new_project` MLRun method
- the `get_or_create_project` method: loads a project from the MLRun DB or the archive/context if it exists, or creates a new project if it doesn't exist.

Both methods have the following parameters:

- **`name`** (required) — the project name.
- **`context`** — the path to a local project directory (the project's context directory). The project directory contains a project-configuration file (default: **project.yaml**) that defines the project, and additional generated Python code. The project file is created when you save your project (using the `save` MLRun project method or when saving your first function within the project).
- **`init_git`** — set to `True` to perform Git initialization of the project directory (`context`) in case its not initialized.

  Note: It's customary to store project code and definitions in a Git repository.

The following code gets or creates a user project named "getting-started-<username>".

Note: Platform projects are currently shared among all users of the parent tenant, to facilitate collaboration. Therefore:

- Set `user_project` to `True` if you want to create a project unique to your user. You can easily change the default project name for this tutorial by changing the definition of the `project_name_base` variable in the following code.
- Don't include in your project proprietary information that you don't want to expose to other users. Note that while projects are a useful tool, you can easily develop and run code in the platform without using projects.

```
import mlrun

# Set the base project name
project_name_base = 'getting-started'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_
↪project=True, init_git=True)

print(f'Full project name: {project.metadata.name}')
```

```
> 2022-02-08 22:48:12,960 [info] loaded project getting-started from MLRun DB
Full project name: getting-started-admin
```

Step 2: Updating Project and Function Definitions

You must save the definitions for the functions used in the project so that you can automatically convert code to functions, import external functions when you load new versions of MLRun code, or run automated CI/CD workflows. In addition, you might want to set other project attributes such as global parameters, secrets, and data.

The code can be stored in Python files, notebooks, external repositories, packaged containers, etc. Use the `project.set_function()` method to register the code in the project. The definitions are saved to the project object as well as in a YAML file in the root of the project. Functions can also be imported from MLRun marketplace (using the `hub://` schema).

This tutorial uses the functions:

- `prep-data` — the first function, which ingests the Iris data set (in Notebook 01)
- `describe` — generates statistics on the data set (from the marketplace)
- `train-iris` — the model-training function (in Notebook 02)
- `test-classifier` — the model-testing function (from the marketplace)
- `mlrun-model` — the model-serving function (in Notebook 03)

  Note: `set_function` uses the `code_to_function` and `import_function` methods under the hood (used in the previous notebooks), but in addition it saves the function configurations in the project spec for use in automated workflows and CI/CD.

Add the function definitions to the project along with parameters and data artifacts, and save the project.

```
project.set_function('01-mlrun-basics.ipynb', 'prep-data', kind='job', image='mlrun/
↪mlrun')
project.set_function('02-model-training.ipynb', 'train', kind='job', image='mlrun/
↪mlrun', handler='train_iris')

# set project level parameters and save
project.spec.params = {'label_column': 'label'}
project.save()
```

When you save the project it stores the project definitions in the `project.yaml`. This means that you can load the project from the source control (GIT) and run it with a single command or API call.

The project YAML for this project can be printed using:

```
print(project.to_yaml())
```

```
kind: project
metadata:
  name: getting-started-admin
  created: '2022-02-06T17:00:53.057000+00:00'
spec:
  params:
    label_column: label
  functions:
  - url: 01-mlrun-basics.ipynb
    name: prep-data
    kind: job
    image: mlrun/mlrun
  - url: 02-model-training.ipynb
```

(continues on next page)

(continued from previous page)

```
    name: train
    kind: job
    image: mlrun/mlrun
    handler: train_iris
  workflows:
  - name: main
    path: workflow.py
    engine: null
  artifacts: []
  source: ''
  subpath: ''
  origin_url: ''
  desired_state: online
  owner: admin
  disable_auto_mount: false
status:
  state: online
```

Saving and Loading Projects from GIT

After you save the project and its elements (functions, workflows, artifacts, etc.) you can commit all the changes to a GIT repository. Use the standard GIT tools or use the MLRun `project` methods such as `pull`, `push`, `remote`, which call the Git API for you.

Projects can then be loaded from GIT using the MLRun `load_project` method, for example:

```
project = mlrun.load_project("./myproj", "git://github.com/mlrun/project-demo.git",␣
↪name=project_name)
```

or using MLRun CLI:

```
mlrun project -n myproj -u "git://github.com/mlrun/project-demo.git" ./myproj
```

Read *Create and load projects* for more details.

Using Kubeflow Pipelines

You're now ready to create a full ML pipeline. This is done by using Kubeflow Pipelines — an open-source framework for building and deploying portable, scalable machine-learning workflows based on Docker containers. MLRun leverages this framework to take your existing code and deploy it as steps in the pipeline.

> **Note:** When using the Iguazio MLOps Platform, Kubeflow Pipelines is available as a default (pre-deployed) shared platform service.

Step 3: Defining and Saving a Pipeline Workflow

A pipeline is created by running an MLRun **"workflow"**. The following code defines a workflow and writes it to a file in your local directory, with the file name **workflow.py**. The workflow describes a directed acyclic graph (DAG) for execution using Kubeflow Pipelines, and depicts the connections between the functions and the data as part of an end-to-end pipeline. The workflow file has two parts: initialization of the function objects, and definition of a pipeline DSL (domain-specific language) for connecting the function inputs and outputs. Examine the code to see how function objects are initialized and used (by name) within the workflow.

The defined pipeline includes the following steps:

- Ingest the Iris flower data set (`ingest`).
- Train and the model (`train`).
- Test the model with its test data set.
- Deploy the model as a real-time serverless function (`deploy`).

  Note: A pipeline can also include continuous build integration and deployment (CI/CD) steps, such as building container images and deploying models.

```
%%writefile './workflow.py'

from kfp import dsl
import mlrun

DATASET = 'cleaned_data'

# Create a Kubeflow Pipelines pipeline
@dsl.pipeline(
    name="Getting-started-tutorial",
    description="Demonstrate some of the main capabilities of MLRun"
)
def kfpipeline(source_url, model_name="iris"):
    project = mlrun.get_current_project()

    # Ingest the data set
    ingest = mlrun.run_function(
        'prep-data',
        handler='prep_data',
        inputs={'source_url': source_url},
        params={'label_column': project.get_param("label_column")},
        outputs=[DATASET])

    # Train a model
    train = mlrun.run_function(
        "train",
        params={"label_column": project.get_param("label_column")},
        inputs={"dataset": ingest.outputs[DATASET]},
        outputs=['model', 'test_set'])

    # Test and visualize the model
    test = mlrun.run_function(
        "hub://test_classifier",
        params={"label_column": project.get_param("label_column")},
        inputs={"models_path": train.outputs['model'],
                "test_set": train.outputs['test_set']})
```

(continues on next page)

(continued from previous page)

```
    # Deploy the model as a serverless function
    serving = mlrun.import_function("hub://v2_model_server", new_name="serving")
    deploy = mlrun.deploy_function(serving,
                                   models=[{"key":model_name, "model_path": train.
↪outputs['model']}])
```

```
Overwriting ./workflow.py
```

Step 4: Registering the Workflow

Use the `set_workflow` MLRun project method to register your workflow with MLRun. The following code sets the `name` parameter to the selected workflow name ("main") and the `code` parameter to the name of the workflow file that is found in your project directory (**workflow.py**).

```
# Register the workflow file as "main"
project.set_workflow('main', 'workflow.py')
```

Step 5: Running A Pipeline

First run the following code to save your project:

```
project.save()
```

Use the `run` MLRun project method to execute your workflow pipeline with Kubeflow Pipelines. The tutorial code sets the following method parameters; (for the full parameters list, see the MLRun documentation or embedded help):

- **`name`** — the workflow name (in this case, "main" — see the previous step).
- **`arguments`** — A dictionary of Kubeflow Pipelines arguments (parameters). The tutorial code sets this parameter to an empty arguments list (`{}`), but you can edit the code to add arguments.
- **`artifact_path`** — a path or URL that identifies a location for storing the workflow artifacts. You can use `{{workflow.uid}}` in the path to signify the ID of the current workflow run iteration. The tutorial code sets the artifacts path to a **<worker ID>** directory (`{{workflow.uid}}`) in a **pipeline** directory under the projects container (**/v3io/projects/getting-started-tutorial-project name/pipeline/<worker ID>**).
- **`dirty`** — set to `True` to allow running the workflow also when the project's Git repository is dirty (i.e., contains uncommitted changes). (When the notebook that contains the execution code is in the same Git directory as the executed workflow, the directory is always dirty during the execution.)
- **`watch`** — set to `True` to wait for the pipeline to complete and output the execution graph as it updates.

The `run` method returns the ID of the executed workflow, which the code stores in a `run_id` variable. You can use this ID to track the progress or your workflow, as demonstrated in the following sections.

> **Note**: You can also run the workflow from a command-line shell by using the `mlrun` CLI. The following CLI command defines a similar execution logic as that of the `run` call in the tutorial:
>
> ```
> mlrun project /User/getting-started-tutorial/conf -r main -p "$V3IO_HOME_URL/
> ↪getting-started-tutorial/pipeline/{{workflow.uid}}/"
> ```

```
source_url = mlrun.get_sample_path("data/iris/iris.data.raw.csv")
model_name="iris"

# when not using a managed cluster run with "local" engine (assume no Kubeflow)
engine = "kfp" if mlrun.mlconf.remote_host else "local"
```

```
model_name="iris"
run_id = project.run(
    'main',
    arguments={'source_url' : source_url, "model_name": model_name},
    #dirty=True,
    engine=engine,
    watch=True)
```

```
<IPython.core.display.HTML object>
```

```
<graphviz.dot.Digraph at 0x7fe3d1241450>
```

```
<IPython.core.display.HTML object>
```

Step 6: Viewing the Pipeline on the Dashboard (UI)

In the **Projects > Jobs and Workflows > Monitor Workflows** tab, press the workflow name to view a graph of the workflow. Press any step to open another pane with full details of the step: either the job's overview, inputs, artifacts, etc.; or the deploy / build function's overview, code, and log.

After the pipelines execution completes, you should be able to view the pipeline and see its functions:

- `prep-data`
- `train`
- `test`
- `deploy-serving`

The graph is refreshed while the pipeline is running.

Step 7: Invoking the Model

Now that your model is deployed using the pipeline, you can invoke it as usual:

```
serving_func = project.func('serving')  # get the latest serving function

my_data = {'inputs': [[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}
serving_func.invoke(f'/v2/models/{model_name}/predict', my_data)
```

```
> 2022-02-08 22:33:07,930 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-getting-started-admin-serving.default-tenant.svc.cluster.local:8080/v2/
↪models/iris/infer'}
```

```
{'id': '1a938c32-0be5-4f6b-a468-0cfc0cb60a6a',
 'model_name': 'iris',
 'outputs': [0, 2]}
```

You can also make an HTTP call directly:

```
import requests
import json
predict_url = f'http://{serving_func.status.address}/v2/models/{model_name}/predict'
resp = requests.put(predict_url, json=json.dumps(my_data))
print(resp.json())
```

```
{'id': '296c0de3-0d3d-4286-84ef-ed297d15727f', 'model_name': 'iris', 'outputs': [0,␣
↪2]}
```

Done!

Congratulations! You've completed the getting started tutorial.

You might also want to explore the following demos:

- For an example of distributed training of an image-classification pipeline using TensorFlow (versions 1 or 2), Keras, and Horovod, see the **image-classification with distributed training demo**.
- To learn more about deploying live endpoints and concept drift, see the **network-operations (NetOps) demo**.
- To learn how to deploy your model with streaming information, see the **model-deployment pipeline demo**.

For additional information and guidelines, see the MLRun ***How-To Guides and Demos***.

Converting Research Notebook to Operational Pipeline With MLRun

Overview | *Running the Demo* | *Demo Flow* | *Pipeline Output* | *Notebooks and Code*

Overview

This demo demonstrates how to convert existing machine-learning (ML) code to an MLRun project. The demo implements an MLRun project for taxi ride-fare prediction based on a Kaggle notebook with an ML Python script that uses data from the New York City Taxi Fare Prediction competition.

Running the Demo

To run the demo, simply open the ***mlrun-code.ipynb*** notebook from an environment with a running MLRun service and run the code cells.

Demo Flow

The code includes the following components:

1. **Data ingestion** — ingest NYC taxi-rides data.
2. **Data cleaning and preparation** — process the data to prepare it for the model training.
3. **Model training** — train an ML model that predicts taxi-ride fares.
4. **Model serving** — deploy a function for serving the trained model.

Pipeline Output

Notebooks and Code

Original NYC Taxi ML Notebook

```
%pip install lightgbm shapely
```

```
Requirement already satisfied: lightgbm in /conda/lib/python3.7/site-packages (2.3.0)
Requirement already satisfied: shapely in /User/.pythonlibs/jupyter2/lib/python3.7/
↪site-packages (1.7.1)
Requirement already satisfied: scipy in /conda/lib/python3.7/site-packages (from␣
↪lightgbm) (1.4.1)
Requirement already satisfied: numpy in /conda/lib/python3.7/site-packages (from␣
↪lightgbm) (1.19.2)
Requirement already satisfied: scikit-learn in /conda/lib/python3.7/site-packages␣
↪(from lightgbm) (0.23.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /conda/lib/python3.7/site-
↪packages (from scikit-learn->lightgbm) (2.1.0)
Requirement already satisfied: joblib>=0.11 in /conda/lib/python3.7/site-packages␣
↪(from scikit-learn->lightgbm) (0.17.0)
Note: you may need to restart the kernel to use updated packages.
```

```
import numpy as np
import pandas as pd
import scipy as scipy
import datetime as dt
from sklearn.model_selection import train_test_split
import lightgbm as lgbm
import os
import gc
import shapely.wkt
from io import StringIO
```

```
def clean_df(df):
    return df[(df.fare_amount > 0)  & (df.fare_amount <= 500) &
              (df.PULocationID > 0) & (df.PULocationID <= 263) &
              (df.DOLocationID > 0) & (df.DOLocationID <= 263)]
```

```
def radian_conv(degree):
    """
    Return radian.
    """
    return  np.radians(degree)
```

```
# To Compute Haversine distance
def sphere_dist(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon):
    """
    Return distance along great radius between pickup and dropoff coordinates.
    """
    #Define earth radius (km)
    R_earth = 6371
    #Convert degrees to radians
    pickup_lat, pickup_lon, dropoff_lat, dropoff_lon = map(np.radians,
                                                           [pickup_lat, pickup_lon,
                                                            dropoff_lat, dropoff_
↪lon])
    #Compute distances along lat, lon dimensions
    dlat = dropoff_lat - pickup_lat
    dlon = dropoff_lon - pickup_lon

    #Compute haversine distance
    a = np.sin(dlat/2.0)**2 + np.cos(pickup_lat) * np.cos(dropoff_lat) * np.sin(dlon/
↪2.0)**2
    return 2 * R_earth * np.arcsin(np.sqrt(a))
```

```
def add_airport_dist(dataset):
    """
    Return minimum distance from pickup or dropoff coordinates to each airport.
    JFK: John F. Kennedy International Airport
    EWR: Newark Liberty International Airport
    LGA: LaGuardia Airport
    SOL: Statue of Liberty
    NYC: Newyork Central
    """
    jfk_coord = (40.639722, -73.778889)
    ewr_coord = (40.6925, -74.168611)
    lga_coord = (40.77725, -73.872611)
    sol_coord = (40.6892,-74.0445) # Statue of Liberty
    nyc_coord = (40.7141667,-74.0063889)


    pickup_lat = dataset['pickup_latitude']
    dropoff_lat = dataset['dropoff_latitude']
    pickup_lon = dataset['pickup_longitude']
    dropoff_lon = dataset['dropoff_longitude']

    pickup_jfk = sphere_dist(pickup_lat, pickup_lon, jfk_coord[0], jfk_coord[1])
    dropoff_jfk = sphere_dist(jfk_coord[0], jfk_coord[1], dropoff_lat, dropoff_lon)
    pickup_ewr = sphere_dist(pickup_lat, pickup_lon, ewr_coord[0], ewr_coord[1])
    dropoff_ewr = sphere_dist(ewr_coord[0], ewr_coord[1], dropoff_lat, dropoff_lon)
    pickup_lga = sphere_dist(pickup_lat, pickup_lon, lga_coord[0], lga_coord[1])
    dropoff_lga = sphere_dist(lga_coord[0], lga_coord[1], dropoff_lat, dropoff_lon)
    pickup_sol = sphere_dist(pickup_lat, pickup_lon, sol_coord[0], sol_coord[1])
    dropoff_sol = sphere_dist(sol_coord[0], sol_coord[1], dropoff_lat, dropoff_lon)
    pickup_nyc = sphere_dist(pickup_lat, pickup_lon, nyc_coord[0], nyc_coord[1])
```

(continues on next page)

(continued from previous page)

```
    dropoff_nyc = sphere_dist(nyc_coord[0], nyc_coord[1], dropoff_lat, dropoff_lon)



    dataset['jfk_dist'] = pickup_jfk + dropoff_jfk
    dataset['ewr_dist'] = pickup_ewr + dropoff_ewr
    dataset['lga_dist'] = pickup_lga + dropoff_lga
    dataset['sol_dist'] = pickup_sol + dropoff_sol
    dataset['nyc_dist'] = pickup_nyc + dropoff_nyc

    return dataset
```

```
def add_datetime_info(dataset):
    #Convert to datetime format
    dataset['pickup_datetime'] = pd.to_datetime(dataset['tpep_pickup_datetime'],
↪format="%Y-%m-%d %H:%M:%S")

    dataset['hour'] = dataset.pickup_datetime.dt.hour
    dataset['day'] = dataset.pickup_datetime.dt.day
    dataset['month'] = dataset.pickup_datetime.dt.month
    dataset['weekday'] = dataset.pickup_datetime.dt.weekday
    dataset['year'] = dataset.pickup_datetime.dt.year

    return dataset
```

```
def get_zones_dict(zones_url):
    zones_df = pd.read_csv(zones_url)

    # Remove unnecessary fields
    zones_df.drop(['Shape_Leng', 'Shape_Area', 'zone', 'LocationID', 'borough'],␣
↪axis=1, inplace=True)

    # Convert DF to dictionary
    zones_dict = zones_df.set_index('OBJECTID').to_dict('index')

    # Add lat/long to each zone
    for zone in zones_dict:
        shape = shapely.wkt.loads(zones_dict[zone]['the_geom'])
        zones_dict[zone]['long'] = shape.centroid.x
        zones_dict[zone]['lat'] = shape.centroid.y

    return zones_dict
```

```
def get_zone_lat(zones_dict, zone_id):
    return zones_dict[zone_id]['lat']
```

```
def get_zone_long(zones_dict, zone_id):
    return zones_dict[zone_id]['long']
```

```
zones_dict = get_zones_dict('https://s3.wasabisys.com/iguazio/data/Taxi/taxi_zones.csv
↪')
```

```
new_train_df = pd.read_csv('https://s3.wasabisys.com/iguazio/data/Taxi/yellow_
↪tripdata_2019-01_subset.csv')
new_train_df = clean_df(new_train_df)
```

```
# This can take a minute or two
new_train_df['pickup_latitude'] = new_train_df.apply(lambda x: get_zone_lat(zones_
↪dict, x['PULocationID']), axis=1 )
new_train_df['pickup_longitude'] = new_train_df.apply(lambda x: get_zone_long(zones_
↪dict, x['PULocationID']), axis=1 )
new_train_df['dropoff_latitude'] = new_train_df.apply(lambda x: get_zone_lat(zones_
↪dict, x['DOLocationID']), axis=1 )
new_train_df['dropoff_longitude'] = new_train_df.apply(lambda x: get_zone_long(zones_
↪dict, x['DOLocationID']), axis=1 )
```

```
new_train_df = add_datetime_info(new_train_df)
new_train_df = add_airport_dist(new_train_df)

new_train_df['pickup_latitude'] = radian_conv(new_train_df['pickup_latitude'])
new_train_df['pickup_longitude'] = radian_conv(new_train_df['pickup_longitude'])
new_train_df['dropoff_latitude'] = radian_conv(new_train_df['dropoff_latitude'])
new_train_df['dropoff_longitude'] = radian_conv(new_train_df['dropoff_longitude'])

y = new_train_df['fare_amount']
```

```
# Remove unnecessary fields
new_train_df.drop(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
↪'congestion_surcharge', 'improvement_surcharge', 'pickup_datetime',
                   'extra', 'mta_tax', 'tip_amount', 'tolls_amount', 'total_amount',
↪'RatecodeID', 'store_and_fwd_flag',
                   'PULocationID', 'DOLocationID', 'payment_type', 'fare_amount'],
                   axis=1, inplace=True, errors='ignore')
```

```
x_train,x_test,y_train,y_test = train_test_split(new_train_df,y,random_state=123,test_
↪size=0.10)
```

```
del new_train_df
del y
gc.collect()
```

```
79
```

```
params = {
        'boosting_type':'gbdt',
        'objective': 'regression',
        'nthread': 4,
        'num_leaves': 31,
        'learning_rate': 0.05,
        'max_depth': -1,
        'subsample': 0.8,
        'bagging_fraction' : 1,
        'max_bin' : 5000 ,
        'bagging_freq': 20,
        'colsample_bytree': 0.6,
        'metric': 'rmse',
        'min_split_gain': 0.5,
        'min_child_weight': 1,
        'min_child_samples': 10,
        'scale_pos_weight':1,
        'zero_as_missing': True,
```

(continues on next page)

(continued from previous page)

```
        'seed':0,
        'num_rounds':50000
    }
```

```
train_set = lgbm.Dataset(x_train, y_train, silent=False,categorical_feature=['year',
↪'month','day','weekday'])
valid_set = lgbm.Dataset(x_test, y_test, silent=False,categorical_feature=['year',
↪'month','day','weekday'])
model = lgbm.train(params, train_set = train_set, num_boost_round=10000,early_
↪stopping_rounds=500,verbose_eval=500, valid_sets=valid_set)
del x_train
del y_train
del x_test
del y_test
gc.collect()
```

```
/conda/lib/python3.7/site-packages/lightgbm/engine.py:148: UserWarning: Found `num_
↪rounds` in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
/conda/lib/python3.7/site-packages/lightgbm/basic.py:1243: UserWarning: Using␣
↪categorical_feature in Dataset.
  warnings.warn('Using categorical_feature in Dataset.')
```

```
Training until validation scores don't improve for 500 rounds
[500]        valid_0's rmse: 3.01588
[1000]        valid_0's rmse: 2.9836
[1500]        valid_0's rmse: 2.97776
[2000]        valid_0's rmse: 2.98137
Early stopping, best iteration is:
[1615]        valid_0's rmse: 2.97663
```

```
732
```

```
test_data = StringIO("""
passenger_count,trip_distance,pickup_latitude,pickup_longitude,dropoff_latitude,
↪dropoff_longitude,hour,day,month,weekday,year,jfk_dist,ewr_dist,lga_dist,sol_dist,
↪nyc_dist
1,0.80,0.711950,-1.291073,0.712059,1.290988,13,1,1,1,2019,47.274013,40.386065,16.
↪975747,26.587155,18.925788
""")
```

```
test_df = pd.read_csv(test_data, sep=",")
```

```
test_df
```

```
   passenger_count  trip_distance  pickup_latitude  pickup_longitude  \
0                1            0.8          0.71195         -1.291073

   dropoff_latitude  dropoff_longitude  hour  day  month  weekday  year  \
0          0.712059           1.290988    13    1      1        1  2019

    jfk_dist   ewr_dist   lga_dist   sol_dist   nyc_dist
0  47.274013  40.386065  16.975747  26.587155  18.925788
```

```
#Predict from test set
prediction = model.predict(test_df, num_iteration = model.best_iteration)
print(prediction)
```

```
[9.53629134]
```

Refactored As Operational Pipeline (with MLRun)

```
# Install prerequisites
%pip install mlrun lightgbm shapely
```

Create an MLRun project and configuration

```
from os import path
import mlrun

project_name_base = 'nyc-taxi'

project_name, artifact_path = mlrun.set_environment(project=project_name_base, user_
↪project=True)

print(f'Project name: {project_name}')
print(f'Artifact path: {artifact_path}')
```

```
Project name: nyc-taxi-iguazio
Artifact path: /v3io/projects/{{run.project}}/artifacts
```

Define Nuclio and MLRun Functions

```
# mlrun: start-code
```

```
from os import path
import numpy as np
import pandas as pd
import datetime as dt
from sklearn.model_selection import train_test_split
import lightgbm as lgbm
from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from pickle import dumps
import shapely.wkt
```

```
def get_zones_dict(zones_url):
    zones_df = pd.read_csv(zones_url)

    # Remove unnecessary fields
    zones_df.drop(['Shape_Leng', 'Shape_Area', 'zone', 'LocationID', 'borough'],␣
↪axis=1, inplace=True)
```

(continues on next page)

(continued from previous page)

```
    # Convert DF to dictionary
    zones_dict = zones_df.set_index('OBJECTID').to_dict('index')

    # Add lat/long to each zone
    for zone in zones_dict:
        shape = shapely.wkt.loads(zones_dict[zone]['the_geom'])
        zones_dict[zone]['long'] = shape.centroid.x
        zones_dict[zone]['lat'] = shape.centroid.y

    return zones_dict
```

```
def get_zone_lat(zones_dict, zone_id):
    return zones_dict[zone_id]['lat']
```

```
def get_zone_long(zones_dict, zone_id):
    return zones_dict[zone_id]['long']
```

```
def clean_df(df):
    return df[(df.fare_amount > 0)  & (df.fare_amount <= 500) &
              (df.PULocationID > 0) & (df.PULocationID <= 263) &
              (df.DOLocationID > 0) & (df.DOLocationID <= 263)]
```

```
# To Compute Haversine distance
def sphere_dist(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon):
    """
    Return distance along great radius between pickup and dropoff coordinates.
    """
    #Define earth radius (km)
    R_earth = 6371
    #Convert degrees to radians
    pickup_lat, pickup_lon, dropoff_lat, dropoff_lon = map(np.radians,
                                                           [pickup_lat, pickup_lon,
                                                            dropoff_lat, dropoff_
↪lon])
    #Compute distances along lat, lon dimensions
    dlat = dropoff_lat - pickup_lat
    dlon = dropoff_lon - pickup_lon

    #Compute haversine distance
    a = np.sin(dlat/2.0)**2 + np.cos(pickup_lat) * np.cos(dropoff_lat) * np.sin(dlon/
↪2.0)**2
    return 2 * R_earth * np.arcsin(np.sqrt(a))
```

```
def radian_conv(degree):
    """
    Return radian.
    """
    return  np.radians(degree)
```

```
def add_airport_dist(dataset):
    """
    Return minimum distance from pickup or dropoff coordinates to each airport.
    JFK: John F. Kennedy International Airport
    EWR: Newark Liberty International Airport
```

(continues on next page)

(continued from previous page)

```
    LGA: LaGuardia Airport
    SOL: Statue of Liberty
    NYC: Newyork Central
    """
    jfk_coord = (40.639722, -73.778889)
    ewr_coord = (40.6925, -74.168611)
    lga_coord = (40.77725, -73.872611)
    sol_coord = (40.6892,-74.0445) # Statue of Liberty
    nyc_coord = (40.7141667,-74.0063889)


    pickup_lat = dataset['pickup_latitude']
    dropoff_lat = dataset['dropoff_latitude']
    pickup_lon = dataset['pickup_longitude']
    dropoff_lon = dataset['dropoff_longitude']

    pickup_jfk = sphere_dist(pickup_lat, pickup_lon, jfk_coord[0], jfk_coord[1])
    dropoff_jfk = sphere_dist(jfk_coord[0], jfk_coord[1], dropoff_lat, dropoff_lon)
    pickup_ewr = sphere_dist(pickup_lat, pickup_lon, ewr_coord[0], ewr_coord[1])
    dropoff_ewr = sphere_dist(ewr_coord[0], ewr_coord[1], dropoff_lat, dropoff_lon)
    pickup_lga = sphere_dist(pickup_lat, pickup_lon, lga_coord[0], lga_coord[1])
    dropoff_lga = sphere_dist(lga_coord[0], lga_coord[1], dropoff_lat, dropoff_lon)
    pickup_sol = sphere_dist(pickup_lat, pickup_lon, sol_coord[0], sol_coord[1])
    dropoff_sol = sphere_dist(sol_coord[0], sol_coord[1], dropoff_lat, dropoff_lon)
    pickup_nyc = sphere_dist(pickup_lat, pickup_lon, nyc_coord[0], nyc_coord[1])
    dropoff_nyc = sphere_dist(nyc_coord[0], nyc_coord[1], dropoff_lat, dropoff_lon)



    dataset['jfk_dist'] = pickup_jfk + dropoff_jfk
    dataset['ewr_dist'] = pickup_ewr + dropoff_ewr
    dataset['lga_dist'] = pickup_lga + dropoff_lga
    dataset['sol_dist'] = pickup_sol + dropoff_sol
    dataset['nyc_dist'] = pickup_nyc + dropoff_nyc

    return dataset
```

```
def add_datetime_info(dataset):
    #Convert to datetime format
    dataset['pickup_datetime'] = pd.to_datetime(dataset['tpep_pickup_datetime'],
↪format="%Y-%m-%d %H:%M:%S")

    dataset['hour'] = dataset.pickup_datetime.dt.hour
    dataset['day'] = dataset.pickup_datetime.dt.day
    dataset['month'] = dataset.pickup_datetime.dt.month
    dataset['weekday'] = dataset.pickup_datetime.dt.weekday
    dataset['year'] = dataset.pickup_datetime.dt.year

    return dataset
```

```
def fetch_data(context : MLClientCtx, taxi_records_csv_path: DataItem, zones_csv_
↪path: DataItem):

    context.logger.info('Reading taxi records data from {}'.format(taxi_records_csv_
↪path))
    taxi_records_dataset = taxi_records_csv_path.as_df()
```

(continues on next page)

(continued from previous page)

```
    context.logger.info('Reading zones data from {}'.format(zones_csv_path))
    zones_dataset = zones_csv_path.as_df()

    target_path = path.join(context.artifact_path, 'data')
    context.logger.info('Saving datasets to {} ...'.format(target_path))

    # Store the data sets in your artifacts database
    context.log_dataset('nyc-taxi-dataset', df=taxi_records_dataset, format='csv',
                        index=False, artifact_path=target_path)
    context.log_dataset('zones-dataset', df=zones_dataset, format='csv',
                        index=False, artifact_path=target_path)
```

```
def get_zones_dict(zones_df):

    # Remove unnecessary fields
    zones_df.drop(['Shape_Leng', 'Shape_Area', 'zone', 'LocationID', 'borough'],␣
↪axis=1, inplace=True)

    # Convert DF to dictionary
    zones_dict = zones_df.set_index('OBJECTID').to_dict('index')

    # Add lat/long to each zone
    for zone in zones_dict:
        shape = shapely.wkt.loads(zones_dict[zone]['the_geom'])
        zones_dict[zone]['long'] = shape.centroid.x
        zones_dict[zone]['lat'] = shape.centroid.y

    return zones_dict
```

```
def get_zone_lat(zones_dict, zone_id):
    return zones_dict[zone_id]['lat']
```

```
def get_zone_long(zones_dict, zone_id):
    return zones_dict[zone_id]['long']
```

```
def transform_dataset(context : MLClientCtx, taxi_records_csv_path: DataItem, zones_
↪csv_path: DataItem):

    context.logger.info('Begin datasets transform')

    context.logger.info('zones_csv_path: ' + str(zones_csv_path))

    zones_df = zones_csv_path.as_df()

    # Get zones dictionary
    zones_dict = get_zones_dict(zones_df)

    train_df = taxi_records_csv_path.as_df()

    # Clean DF
    train_df = clean_df(train_df)

    # Enrich DF
    train_df['pickup_latitude'] = train_df.apply(lambda x: get_zone_lat(zones_dict, x[
↪'PULocationID']), axis=1 )
```

(continues on next page)

(continued from previous page)

```
    train_df['pickup_longitude'] = train_df.apply(lambda x: get_zone_long(zones_dict,␣
↪x['PULocationID']), axis=1 )
    train_df['dropoff_latitude'] = train_df.apply(lambda x: get_zone_lat(zones_dict,␣
↪x['DOLocationID']), axis=1 )
    train_df['dropoff_longitude'] = train_df.apply(lambda x: get_zone_long(zones_dict,
↪ x['DOLocationID']), axis=1 )

    train_df = add_datetime_info(train_df)
    train_df = add_airport_dist(train_df)

    train_df['pickup_latitude'] = radian_conv(train_df['pickup_latitude'])
    train_df['pickup_longitude'] = radian_conv(train_df['pickup_longitude'])
    train_df['dropoff_latitude'] = radian_conv(train_df['dropoff_latitude'])
    train_df['dropoff_longitude'] = radian_conv(train_df['dropoff_longitude'])

    train_df.drop(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
↪'congestion_surcharge', 'improvement_surcharge', 'pickup_datetime',
                   'extra', 'mta_tax', 'tip_amount', 'tolls_amount', 'total_amount',
↪'RatecodeID', 'store_and_fwd_flag',
                   'PULocationID', 'DOLocationID', 'payment_type'],
                   axis=1, inplace=True, errors='ignore')

    # Save dataset to artifact
    target_path = path.join(context.artifact_path, 'data')
    context.log_dataset('nyc-taxi-dataset-transformed', df=train_df, artifact_
↪path=target_path, format='csv')

    context.logger.info('End dataset transform')
```

```
params = {
        'boosting_type':'gbdt',
        'objective': 'regression',
        'nthread': 4,
        'num_leaves': 31,
        'learning_rate': 0.05,
        'max_depth': -1,
        'subsample': 0.8,
        'bagging_fraction' : 1,
        'max_bin' : 5000 ,
        'bagging_freq': 20,
        'colsample_bytree': 0.6,
        'metric': 'rmse',
        'min_split_gain': 0.5,
        'min_child_weight': 1,
        'min_child_samples': 10,
        'scale_pos_weight':1,
        'zero_as_missing': True,
        'seed':0,
        'num_rounds':50000
    }
```

```
def train_model(context: MLClientCtx, input_ds: DataItem):

    context.logger.info('Begin training')
    context.logger.info('LGBM version is ' + str(lgbm.__version__))
```

(continues on next page)

(continued from previous page)

```
    train_df = input_ds.as_df()

    y = train_df['fare_amount']

    train_df = train_df.drop(columns=['fare_amount'])
    train_df = train_df.drop(train_df.columns[[0]], axis=1)
    x_train,x_test,y_train,y_test = train_test_split(train_df,y,random_state=123,test_
↪size=0.10)

    train_set = lgbm.Dataset(x_train, y_train, silent=False,categorical_feature=['year
↪','month','day','weekday'])
    valid_set = lgbm.Dataset(x_test, y_test, silent=False,categorical_feature=['year',
↪'month','day','weekday'])
    model = lgbm.train(params, train_set = train_set, num_boost_round=10000,early_
↪stopping_rounds=500,verbose_eval=500, valid_sets=valid_set)

    context.log_model('FareModel',
                      body=dumps(model),
                      artifact_path=context.artifact_subpath("models"),
                      model_file="FareModel.pkl")

    context.logger.info('End training')
```

```
# mlrun: end-code
```

Set Input Paths

```
taxi_records_csv_path = 'https://s3.wasabisys.com/iguazio/data/Taxi/yellow_tripdata_
↪2019-01_subset.csv'
zones_csv_path = 'https://s3.wasabisys.com/iguazio/data/Taxi/taxi_zones.csv'
```

Convert Code to a Function

```
taxi_func = mlrun.code_to_function(name='taxi',
                                   kind='job',
                                   image='mlrun/mlrun',
                                   requirements=['lightgbm', 'shapely'])
```

Run `fetch_data` Locally

We can test out code locally, by calling the function with `local` parameter set to `True`

```
fetch_data_run = taxi_func.run(handler='fetch_data',
                               inputs={'taxi_records_csv_path': taxi_records_csv_path,
                                       'zones_csv_path': zones_csv_path},
                               local=True)
```

```
> 2021-01-28 10:50:17,131 [info] starting run taxi-fetch_data␣
↪uid=6a4667dc66684e96aa4792c109afa2e4 DB=http://mlrun-api:8080
> 2021-01-28 10:50:17,272 [info] Reading taxi records data from https://s3.wasabisys.
↪com/iguazio/data/Taxi/yellow_tripdata_2019-01_subset.csv
```

(continues on next page)

(continued from previous page)

```
> 2021-01-28 10:50:20,868 [info] Reading zones data from https://s3.wasabisys.com/
↪iguazio/data/Taxi/taxi_zones.csv
> 2021-01-28 10:50:21,290 [info] Saving datasets to /v3io/projects/nyc-taxi-iguazio/
↪artifacts/data ...
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 6a4667dc66684e96aa4792c109afa2e4 --project nyc-taxi-iguazio , !mlrun␣
↪logs 6a4667dc66684e96aa4792c109afa2e4 --project nyc-taxi-iguazio
> 2021-01-28 10:50:35,407 [info] run executed, status=completed
```

```
fetch_data_run.outputs
```

```
{'nyc-taxi-dataset': 'store://artifacts/nyc-taxi-iguazio/taxi-fetch_data_nyc-taxi-
↪dataset:6a4667dc66684e96aa4792c109afa2e4',
 'zones-dataset': 'store://artifacts/nyc-taxi-iguazio/taxi-fetch_data_zones-
↪dataset:6a4667dc66684e96aa4792c109afa2e4'}
```

Run on the Cluster

Prepare Cluster Function

Create an MLRun function and create a custom image for it (that uses shapely).

```
from mlrun.platforms import auto_mount
taxi_func.apply(auto_mount())
taxi_func.deploy()
```

```
> 2021-01-28 10:50:35,424 [info] starting remote build, image: .mlrun/func-nyc-taxi-
↪iguazio-taxi-latest
INFO[0020] Retrieving image manifest mlrun/mlrun:0.6.0-rc11
INFO[0020] Retrieving image manifest mlrun/mlrun:0.6.0-rc11
INFO[0021] Built cross stage deps: map[]
INFO[0021] Retrieving image manifest mlrun/mlrun:0.6.0-rc11
INFO[0021] Retrieving image manifest mlrun/mlrun:0.6.0-rc11
INFO[0021] Executing 0 build triggers
INFO[0021] Unpacking rootfs as cmd RUN python -m pip install lightgbm shapely␣
↪requires it.
INFO[0036] RUN python -m pip install lightgbm shapely
INFO[0036] Taking snapshot of full filesystem...
INFO[0048] cmd: /bin/sh
INFO[0048] args: [-c python -m pip install lightgbm shapely]
INFO[0048] Running: [/bin/sh -c python -m pip install lightgbm shapely]
Collecting lightgbm
  Downloading lightgbm-3.1.1-py2.py3-none-manylinux1_x86_64.whl (1.8 MB)
Collecting shapely
  Downloading Shapely-1.7.1-cp37-cp37m-manylinux1_x86_64.whl (1.0 MB)
Requirement already satisfied: scikit-learn!=0.22.0 in /usr/local/lib/python3.7/site-
↪packages (from lightgbm) (0.23.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/site-packages (from␣
↪lightgbm) (1.19.5)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: scipy in /usr/local/lib/python3.7/site-packages (from␣
↪lightgbm) (1.6.0)
Requirement already satisfied: wheel in /usr/local/lib/python3.7/site-packages (from␣
↪lightgbm) (0.36.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/site-
↪packages (from scikit-learn!=0.22.0->lightgbm) (2.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/site-packages␣
↪(from scikit-learn!=0.22.0->lightgbm) (1.0.0)
Installing collected packages: lightgbm, shapely
Successfully installed lightgbm-3.1.1 shapely-1.7.1
WARNING: You are using pip version 20.2.4; however, version 21.0 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade␣
↪pip' command.
INFO[0050] Taking snapshot of full filesystem...
```

```
True
```

```
fetch_data_run = taxi_func.run(name='fetch_data',
                               handler='fetch_data',
                               inputs={'taxi_records_csv_path': taxi_records_csv_path,
                                       'zones_csv_path': zones_csv_path})
```

```
> 2021-01-28 10:51:29,816 [info] starting run fetch_data␣
↪uid=40a122c3066a479d9c09c865487e23ab DB=http://mlrun-api:8080
> 2021-01-28 10:51:30,197 [info] Job is running in the background, pod: fetch-data-
↪f8p7f
> 2021-01-28 10:51:34,657 [info] Reading taxi records data from https://s3.wasabisys.
↪com/iguazio/data/Taxi/yellow_tripdata_2019-01_subset.csv
> 2021-01-28 10:51:38,368 [info] Reading zones data from https://s3.wasabisys.com/
↪iguazio/data/Taxi/taxi_zones.csv
> 2021-01-28 10:51:38,795 [info] Saving datasets to /v3io/projects/nyc-taxi-iguazio/
↪artifacts/data ...
> 2021-01-28 10:51:51,082 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 40a122c3066a479d9c09c865487e23ab --project nyc-taxi-iguazio , !mlrun␣
↪logs 40a122c3066a479d9c09c865487e23ab --project nyc-taxi-iguazio
> 2021-01-28 10:51:59,497 [info] run executed, status=completed
```

```
fetch_data_run.outputs
```

```
{'nyc-taxi-dataset': 'store://artifacts/nyc-taxi-iguazio/fetch_data_nyc-taxi-
↪dataset:40a122c3066a479d9c09c865487e23ab',
 'zones-dataset': 'store://artifacts/nyc-taxi-iguazio/fetch_data_zones-
↪dataset:40a122c3066a479d9c09c865487e23ab'}
```

Transform the Dataset

```
transform_dataset_run = taxi_func.run(name='transform_dataset',
                                      handler='transform_dataset',
                                      inputs={'taxi_records_csv_path': fetch_data_run.
↪outputs['nyc-taxi-dataset'],
                                              'zones_csv_path': fetch_data_run.
↪outputs['zones-dataset']})
```

```
> 2021-01-28 10:51:59,516 [info] starting run transform_dataset␣
↪uid=23089337d5d7483cb81c599da6e64e03 DB=http://mlrun-api:8080
> 2021-01-28 10:52:00,239 [info] Job is running in the background, pod: transform-
↪dataset-5mhwx
> 2021-01-28 10:52:04,698 [info] Begin datasets transform
> 2021-01-28 10:52:04,698 [info] zones_csv_path: /v3io/projects/nyc-taxi-iguazio/
↪artifacts/data/zones-dataset.csv
> 2021-01-28 10:53:06,205 [info] End dataset transform
> 2021-01-28 10:53:06,242 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 23089337d5d7483cb81c599da6e64e03 --project nyc-taxi-iguazio , !mlrun␣
↪logs 23089337d5d7483cb81c599da6e64e03 --project nyc-taxi-iguazio
> 2021-01-28 10:53:09,631 [info] run executed, status=completed
```

```
transform_dataset_run.outputs
```

```
{'nyc-taxi-dataset-transformed': 'store://artifacts/nyc-taxi-iguazio/transform_
↪dataset_nyc-taxi-dataset-transformed:23089337d5d7483cb81c599da6e64e03'}
```

Train Model

```
train_model_run = taxi_func.run(name='train_model',
                                handler='train_model',
                                inputs={'input_ds': transform_dataset_run.outputs[
↪'nyc-taxi-dataset-transformed']})
```

```
> 2021-01-28 10:53:09,648 [info] starting run train_model␣
↪uid=91bbaebdc91a43c9b8412a37b389bffa DB=http://mlrun-api:8080
> 2021-01-28 10:53:09,840 [info] Job is running in the background, pod: train-model-
↪pn5jb
[LightGBM] [Warning] Met categorical feature which contains sparse values. Consider␣
↪renumbering to consecutive integers started from zero
> 2021-01-28 10:53:14,273 [info] Begin training
> 2021-01-28 10:53:14,273 [info] LGBM version is 3.1.1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored.␣
↪Current value: bagging_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored.␣
↪Current value: bagging_fraction=1
[LightGBM] [Warning] Auto-choosing row-wise multi-threading, the overhead of testing␣
↪was 0.006758 seconds.
```

(continues on next page)

(continued from previous page)

```
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 23961
[LightGBM] [Info] Number of data points in the train set: 879294, number of used␣
↪features: 16
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored.␣
↪Current value: bagging_fraction=1
[LightGBM] [Info] Start training from score 12.418691
Training until validation scores don't improve for 500 rounds
[500]        valid_0's rmse: 3.01302
[1000]        valid_0's rmse: 2.98594
[1500]        valid_0's rmse: 2.98293
[2000]        valid_0's rmse: 2.98666
Early stopping, best iteration is:
[1551]        valid_0's rmse: 2.98227
> 2021-01-28 10:53:54,777 [info] End training
> 2021-01-28 10:53:54,845 [info] run executed, status=completed
/usr/local/lib/python3.7/site-packages/lightgbm/engine.py:151: UserWarning: Found␣
↪`num_rounds` in params. Will use it instead of argument
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
/usr/local/lib/python3.7/site-packages/lightgbm/basic.py:1551: UserWarning: Using␣
↪categorical_feature in Dataset.
  warnings.warn('Using categorical_feature in Dataset.')
/usr/local/lib/python3.7/site-packages/lightgbm/basic.py:1286: UserWarning:␣
↪Overriding the parameters from Reference Dataset.
  warnings.warn('Overriding the parameters from Reference Dataset.')
/usr/local/lib/python3.7/site-packages/lightgbm/basic.py:1098: UserWarning:␣
↪categorical_column in param dict is overridden.
  warnings.warn('{} in param dict is overridden.'.format(cat_alias))
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 91bbaebdc91a43c9b8412a37b389bffa --project nyc-taxi-iguazio , !mlrun␣
↪logs 91bbaebdc91a43c9b8412a37b389bffa --project nyc-taxi-iguazio
> 2021-01-28 10:53:58,264 [info] run executed, status=completed
```

```
train_model_run.outputs
```

```
{'FareModel': 'store://artifacts/nyc-taxi-iguazio/train_model_
↪FareModel:91bbaebdc91a43c9b8412a37b389bffa'}
```

Serving

The model serving class is in model-serving.ipynb.

```
serving = mlrun.code_to_function(filename=path.abspath('model-serving.ipynb')).
↪apply(auto_mount())

serving.spec.default_class = 'LGBMModel'
serving.add_model('taxi-serving', train_model_run.outputs['FareModel'])
serving_address = serving.deploy()
```

```
> 2021-01-28 10:54:04,257 [info] Starting remote function deploy
2021-01-28 10:54:04  (info) Deploying function
2021-01-28 10:54:04  (info) Building
2021-01-28 10:54:04  (info) Staging files and preparing base images
2021-01-28 10:54:04  (info) Building processor image
2021-01-28 10:54:07  (info) Build complete
> 2021-01-28 10:54:13,316 [info] function deployed, address=default-tenant.app.
↪jinkwubtllaf.iguazio-cd1.com:30486
```

```
my_data = '''{"inputs":[[1,0.80,0.711950,-1.291073,0.712059,1.290988,13,1,1,1,2019,47.
↪274013,40.386065,16.975747,26.587155,18.925788]]}'''
serving.invoke('/v2/models/taxi-serving/predict', my_data)
```

```
{'id': '985aaf56-fb29-436e-9d96-ef834b8ec12f',
 'model_name': 'taxi-serving',
 'outputs': [9.52302976897415]}
```

Kubeflow Pipeline

Create Project Object

```
project_path = path.abspath('conf')
project = mlrun.new_project(project_name_base,
                            context=project_path,
                            init_git=True,
                            user_project=True)

project.set_function(f'db://{project.name}/taxi')
project.set_function(f'db://{project.name}/model-serving')
```

```
<mlrun.runtimes.serving.ServingRuntime at 0x7f5509b74f90>
```

Create the Workflow

```
%%writefile {path.join(project_path, 'workflow.py')}
from kfp import dsl
from mlrun.platforms import auto_mount

funcs = {}
taxi_records_csv_path = 'https://s3.wasabisys.com/iguazio/data/Taxi/yellow_tripdata_
↪2019-01_subset.csv'
zones_csv_path = 'https://s3.wasabisys.com/iguazio/data/Taxi/taxi_zones.csv'

# init functions is used to configure function resources and local settings
def init_functions(functions: dict, project=None, secrets=None):
    for f in functions.values():
        f.apply(auto_mount())

@dsl.pipeline(
    name="NYC Taxi Demo",
    description="Convert ML script to MLRun"
```

(continues on next page)

(continued from previous page)

```
)

def kfpipeline():

    # build our ingestion function (container image)
    builder = funcs['taxi'].deploy_step(skip_deployed=True)

    # run the ingestion function with the new image and params
    ingest = funcs['taxi'].as_step(
        name="fetch_data",
        handler='fetch_data',
        image=builder.outputs['image'],
        inputs={'taxi_records_csv_path': taxi_records_csv_path,
                'zones_csv_path': zones_csv_path},
        outputs=['nyc-taxi-dataset', 'zones-dataset'])

    # Join and transform the data sets
    transform = funcs["taxi"].as_step(
        name="transform_dataset",
        handler='transform_dataset',
        inputs={"taxi_records_csv_path": ingest.outputs['nyc-taxi-dataset'],
                "zones_csv_path" : ingest.outputs['zones-dataset']},
        outputs=['nyc-taxi-dataset-transformed'])

    # Train the model
    train = funcs["taxi"].as_step(
        name="train",
        handler="train_model",
        inputs={"input_ds" : transform.outputs['nyc-taxi-dataset-transformed']},
        outputs=['FareModel'])

    # Deploy the model
    deploy = funcs["model-serving"].deploy_step(models={"taxi-serving_v1": train.
↪outputs['FareModel']}, tag='v2')
```

```
Overwriting /User/demos/howto/converting-to-mlrun/conf/workflow.py
```

```
project.set_workflow('main', 'workflow.py', embed=True)
```

```
project.save()
```

Run the Workflow

```
artifact_path = path.abspath('./pipe/{{workflow.uid}}')
run_id = project.run(
    'main',
    arguments={},
    artifact_path=artifact_path,
    dirty=True,
    watch=True)
```

```
> 2021-01-28 10:54:13,695 [info] using in-cluster config.
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-01-28 10:54:14,163 [info] Pipeline run id=32e62e7f-424f-474b-b9f5-54ae1834d39f,
↪ check UI or DB for progress
> 2021-01-28 10:54:14,164 [info] waiting for pipeline run completion
```

```
<IPython.core.display.HTML object>
```

Model Serving Function

```
import mlrun
```

```
%nuclio cmd -c pip install lightgbm
```

```
%nuclio config spec.build.baseImage = "mlrun/mlrun"
%nuclio config kind = "serving"
```

```
%nuclio: setting spec.build.baseImage to 'mlrun/mlrun'
%nuclio: setting kind to 'serving'
```

```
import numpy as np
from cloudpickle import load

class LGBMModel(mlrun.serving.V2ModelServer):

    def load(self):
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body):
        try:
            feats = np.asarray(body['inputs'])
            result = self.model.predict(feats)
            return result.tolist()
        except Exception as exc:
            raise Exception(f"Failed to predict {exc}")
```

```
# mlrun: end-code
```

Deploy and Test The Function

```
models_path = 'https://s3.wasabisys.com/iguazio/models/lightgbm/SampleModel.pkl'
```

```
fn = mlrun.code_to_function('lightgbm-serving',
                            description="LightGBM Serving",
                            categories=['serving', 'ml'],
                            labels={'author': 'edmondg', 'framework': 'lightgbm'})
fn.spec.default_class = 'LGBMModel'
```

```
fn.add_model('nyc-taxi-server', model_path=models_path)
```

```
<mlrun.serving.states.TaskState at 0x7fa838f23b50>
```

```
# deploy the function
fn.apply(mlrun.platforms.auto_mount())
address = fn.deploy()
```

```
> 2021-01-28 10:56:00,391 [info] Starting remote function deploy
2021-01-28 10:56:00  (info) Deploying function
2021-01-28 10:56:00  (info) Building
2021-01-28 10:56:00  (info) Staging files and preparing base images
2021-01-28 10:56:00  (info) Building processor image
2021-01-28 10:56:01  (info) Build complete
2021-01-28 10:56:09  (info) Function deploy complete
> 2021-01-28 10:56:10,399 [info] function deployed, address=default-tenant.app.
↪jinkwubtllaf.iguazio-cd1.com:32169
```

```
# test the function
my_data = '''{"inputs":[[5.1, 3.5, 1.4, 3, 5.1, 3.5, 1.4, 0.2, 5.1, 3.5, 1.4, 0.2, 5.
↪1, 3.5, 1.4, 0.2]]}'''
fn.invoke('/v2/models/nyc-taxi-server/predict', my_data)
```

```
{'id': 'cd3aca51-1823-453b-81fb-e2cf2a148906',
 'model_name': 'nyc-taxi-server',
 'outputs': [25.374309065093435]}
```

2.1.5 Installation and setup guide

This guide outlines the steps for installing and running MLRun. Once MLRun is installed you can access it remotely from your IDE (PyCharm or VSCode), read ***how to setup your IDE environment***.

MLRun client backward compatibility

Starting from MLRun 0.10.0, the MLRun client and images are compatible with minor MLRun releases that are released during the following 6 months. When you upgrade to 0.11.0, for example, you can continue to use your 0.10-based images.

Important

- Images from 0.9.0 are not compatible with 0.10.0. Backward compatibility starts from 0.10.0.
- When you upgrade the MLRun major version, for example 0.10.x to 1.0.x, there is no backward compatibility.
- The feature store is not backward compatible.
- When you upgrade the platform, for example from 3.2 to 3.3, the clients should be upgraded. There is no guaranteed compatibility with an older MLRun client after a platform upgrade.

See also Images and their usage in MLRun.

Installation options:

Installing MLRun on a Kubernetes Cluster

In this section

Prerequisites

- Access to a Kubernetes cluster. You must have administrator permissions in order to install MLRun on your cluster. For local installation on Windows or Mac, Docker Desktop is recommended. MLRun fully supports k8s releases up to, and including, 1.21.
- The Kubernetes command-line tool (kubectl) compatible with your Kubernetes cluster is installed. Refer to the kubectl installation instructions for more information.
- Helm CLI is installed. Refer to the Helm installation instructions for more information.
- An accessible docker-registry (such as Docker Hub). The registry's URL and credentials are consumed by the applications via a pre-created secret.

Note

These instructions use `mlrun` as the namespace (`-n` parameter). You can choose a different namespace in your kubernetes cluster.

Installing on Docker Desktop

Docker Desktop is available for Mac and Windows. For download information, system requirements, and installation instructions, see:

- Install Docker Desktop on Mac
- Install Docker Desktop on Windows. Note that WSL 2 backend was tested, Hyper-V was not tested.

Configuring Docker Desktop

Docker Desktop includes a standalone Kubernetes server and client, as well as Docker CLI integration that runs on your machine. The Kubernetes server runs locally within your Docker instance. To enable Kubernetes support and install a standalone instance of Kubernetes running as a Docker container, go to **Preferences > Kubernetes** and then click **Enable Kubernetes**. Click **Apply & Restart** to save the settings and then click **Install** to confirm. This instantiates the images that are required to run the Kubernetes server as containers, and installs the `/usr/local/bin/kubectl` command on your machine. For more information, see the Kubernetes documentation.

It's recommended to limit the amount of memory allocated to Kubernetes. If you're using Windows and WSL 2, you can configure global WSL options by placing a `.wslconfig` file into the root directory of your users folder: `C:\Users\<yourUserName>\.wslconfig`. Keep in mind that you might need to run `wsl --shutdown` to shut down the WSL 2 VM and then restart your WSL instance for these changes to take effect.

```
[wsl2]
memory=8GB # Limits VM memory in WSL 2 to 8 GB
```

To learn about the various UI options and their usage, see:

- Docker Desktop for Mac user manual
- Docker Desktop for Windows user manual

Installing the chart

Create a namespace for the deployed components:

```
kubectl create namespace mlrun
```

Add the `v3io-stable` helm chart repo:

```
helm repo add v3io-stable https://v3io.github.io/helm-charts/stable
```

Update the repo to make sure you're getting the latest chart:

```
helm repo update
```

Create a secret with your docker-registry named `registry-credentials`:

```
kubectl --namespace mlrun create secret docker-registry registry-credentials \
    --docker-server <your-registry-server> \
    --docker-username <your-username> \
    --docker-password <your-password> \
    --docker-email <your-email>
```

Where:

- `<your-registry-server>` is your Private Docker Registry FQDN. (https://index.docker.io/v1/ for Docker Hub).
- `<your-username>` is your Docker username.
- `<your-password>` is your Docker password.
- `<your-email>` is your Docker email.

Note

First-time MLRun users will experience a relatively longer installation time because all required images are being pulled locally for the first time (it will take an average of 10-15 minutes mostly depends on your internet speed).

To install the chart with the release name `mlrun-kit` use the following command. Note the reference to the pre-created `registry-credentials` secret in `global.registry.secretName`:

```
helm --namespace mlrun \
    install mlrun-kit \
    --wait \
    --timeout 960 \
    --set global.registry.url=<registry-url> \
    --set global.registry.secretName=registry-credentials \
    v3io-stable/mlrun-kit
```

Where `<registry-url>` is the registry URL which can be authenticated by the `registry-credentials` secret (e.g., `index.docker.io/<your-username>` for Docker Hub).

Installing on Minikube/VM**

The Open source MLRun kit uses node ports for simplicity. If your Kubernetes cluster is running inside a VM, as is usually the case when using minikube, the Kubernetes services exposed over node ports are not available on your local host interface, but instead, on the virtual machine's interface. To accommodate for this, use the `global.externalHostAddress` value on the chart. For example, if you're using the kit inside a minikube cluster (with some non-empty `vm-driver`), pass the VM address in the chart installation command as follows:

```
helm --namespace mlrun \
    install my-mlrun \
    --wait \
    --timeout 960 \
    --set global.registry.url=<registry URL e.g. index.docker.io/iguazio > \
    --set global.registry.secretName=registry-credentials \
    --set global.externalHostAddress=$(minikube ip) \
    v3io-stable/mlrun-kit
```

Where `$(minikube ip)` shell command resolving the external node address of the k8s node VM.

Installing Kubeflow

You can run your functions while saving outputs and artifacts in a way that is visible to Kubeflow Pipelines. To use this capability you need to install Kubeflow on your cluster. Refer to the Kubeflow documentation for more information.

Usage

Your applications are now available in your local browser:

- Jupyter-notebook - http://localhost:30040
- Nuclio - http://localhost:30050
- MLRun UI - http://localhost:30060
- MLRun API (external) - http://localhost:30070 (health check via http://localhost:30070/api/healthz)

Check state

You can check current state of installation via command `kubectl -n mlrun get pods`, where the main information is in columns `Ready` and `State`. If all images have already been pulled locally, typically it will take a minute for all services to start.

Note

The above links assume your Kubernetes cluster is exposed on localhost. If that's not the case, the different components are available on the provided `externalHostAddress`

- You can change the ports by providing values to the helm install command.
- You can add and configure a k8s ingress-controller for better security and control over external access.

Start working

Open Jupyter Lab on **jupyter-lab UI** and run the code in **quick-start.ipynb** notebook.

Important

Make sure to save your changes in the `data` folder within the Jupyter Lab. The root folder and any other folder do not retain the changes when you restart the Jupyter Lab.

Configuring the remote environment

You can use your code on a local machine while running your functions on a remote cluster.

Prerequisites

Before you begin, ensure that the following prerequisites are met:

- The MLRun version installed with the MLRun Kit is the same as the MLRun version on your remote cluster. If needed, upgrade MLRun either in your local installation or on the remote cluster so that they match.
- You have remote access to your MLRun service (i.e. to the service URL on the remote Kubernetes cluster).

Setting environment variables

Define your MLRun configuration.

- As a minimum requirement: Set `MLRUN_DBPATH` to the URL of the remote MLRun database/API service; replace the `<...>` placeholders to identify your remote target:

```
MLRUN_DBPATH=<API endpoint of the MLRun APIs service endpoint; e.g., "https://
↪mlrun-api.default-tenant.app.mycluster.iguazio.com">
```

- To store the artifacts on the remote server, you need to set the `MLRUN_ARTIFACT_PATH` to the desired root folder of your artifact. You can use template values in the artifact path. The supported values are:
  - `{{project}}` to include the project name in the path.
  - `{{run.uid}}` to include the specific run uid in the artifact path.

  For example:

```
MLRUN_ARTIFACT_PATH=/User/artifacts/{{project}}
```

  or:

```
MLRUN_ARTIFACT_PATH=/User/artifacts/{{project}}/{{run.uid}}
```

- If the remote service is on an instance of the Iguazio MLOps Platform (**"the platform"**), set the following environment variables as well. Replace the `<...>` placeholders with the details for your specific platform cluster:

```
V3IO_USERNAME=<username of a platform user with access to the MLRun service>
V3IO_API=<API endpoint of the webapi service endpoint; e.g., "https://default-
↪tenant.app.mycluster.iguazio.com:8444">
V3IO_ACCESS_KEY=<platform access key>
```

  You can get the platform access key from the platform dashboard: select the user-profile picture or icon from the top right corner of any page, and select **Access Keys** from the menu. In the **Access Keys** dialog, either copy an existing access key or create a new key and copy it. Alternatively, you can get the access key by checking the value of the `V3IO_ACCESS_KEY` environment variable in a web- shell or Jupyter Notebook service.

Advanced chart configuration

Configurable values are documented in the `values.yaml`, and the `values.yaml` of all sub charts. Override those in the normal methods.

Uninstalling the chart

```
helm --namespace mlrun uninstall mlrun-kit
```

Terminating pods and hanging resources

This chart generates several persistent volume claims and also provisions an NFS provisioning server, giving persistency (via PVC) out of the box. Because of the persistency of PV/PVC resources, after installing this chart, PVs and PVCs are created. Upon uninstallation, any hanging / terminating pods hold the PVs and PVCs respectively, since those prevent their safe removal. Since pods that are stuck in terminating state seem to be a never-ending plague in k8s, note this, and remember to clean the remaining PVs and PVCs.

Handing stuck-at-terminating pods:

```
kubectl --namespace mlrun delete pod --force --grace-period=0 <pod-name>
```

Reclaim dangling persistency resources:

WARNING

This will result in data loss!

```
# To list PVCs
kubectl --namespace mlrun get pvc
...

# To remove a PVC
kubectl --namespace mlrun delete pvc <pvc-name>
...

# To list PVs
kubectl --namespace mlrun get pv
...

# To remove a PV
kubectl --namespace mlrun delete pv <pv-name>

# Remove hostpath(s) used for mlrun (and possibly nfs). Those will be created by␣
↪default under /tmp, and will contain
# your release name, e.g.:
rm -rf /tmp/mlrun-kit-mlrun-kit-mlrun
...
```

Install MLRun on a local Docker registry

To use MLRun with your local Docker registry, run the MLRun API service, dashboard, and example Jupyter server, by using the following script.

Note

- Using Docker is limited to local runtimes.
- By default, the MLRun API service runs inside the Jupyter server. Set the MLRUN_DBPATH env var in Jupyter to point to an alternative service address.
- The artifacts and DB are stored under **/home/jovyan/data**. Use the docker -v option to persist the content on the host (e.g. `-v $(SHARED_DIR}:/home/jovyan/data`).
- If Docker is running on Windows with WSL 2, you must create a SHARED_DIR before running these commands. Provide the full path when executing (e.g. `mkdir /mnt/c/mlrun-data SHARED_DIR=/mnt/c/mlrun-data`).

```
SHARED_DIR=~/mlrun-data

docker pull mlrun/jupyter:1.0.0
docker pull mlrun/mlrun-ui:1.0.0

docker network create mlrun-network
docker run -it -p 8080:8080 -p 30040:8888 --rm -d --network mlrun-network --name␣
↪jupyter -v ${SHARED_DIR}:/home/jovyan/data mlrun/jupyter:1.0.0
docker run -it -p 30050:80 --rm -d --network mlrun-network --name mlrun-ui -e MLRUN_
↪API_PROXY_URL=http://jupyter:8080 mlrun/mlrun-ui:1.0.0
```

When the execution completes:

- Open Jupyter Lab on port 30040 and run the code in the **mlrun_basics.ipynb** notebook.
- Use the MLRun dashboard on port 30050.

Setting a remote environment

MLRun allows you to use your code on a local machine while running your functions on a remote cluster. This tutorial explains how you can set this up.

In this section

- *Set environment variables in a terminal*

Prerequisites

Before you begin, ensure that the following prerequisites are met:

1. Install MLRun locally.

   You need to install MLRun locally and make sure the that the MLRun version you install is the same as the MLRun service version. Install a specific version using the following command; replace the `<version>` placeholder with the MLRun version number (e.g., `1.0.0`):

   ```
   pip install mlrun==<version>
   ```

   If you already installed a previous version of MLRun, you should first uninstall it by running:

   ```
   pip uninstall -y mlrun
   ```

2. Ensure that you have remote access to your MLRun service (i.e., to the service URL on the remote Kubernetes cluster).

Configure remote environment

Set environment variables

Set environment variables to define your MLRun configuration. As a minimum requirement:

1. Set `MLRUN_DBPATH` to the URL of the remote MLRun database/API service; replace the `<...>` placeholders to identify your remote target:

   ```
   MLRUN_DBPATH=<API endpoint of the MLRun APIs service endpoint; e.g., "https://
   ↪mlrun-api.default-tenant.app.mycluster.iguazio.com">
   ```

2. To store the artifacts on the remote server, you need to set the `MLRUN_ARTIFACT_PATH` to the desired root folder of your artifact. You can use template values in the artifact path. The supported values are:

   - `{{project}}` to include the project name in the path.
   - `{{run.uid}}` to include the specific run uid in the artifact path.

   For example:

   ```
   MLRUN_ARTIFACT_PATH=/User/artifacts/{{project}}
   ```

   or:

   ```
   MLRUN_ARTIFACT_PATH=/User/artifacts/{{project}}/{{run.uid}}
   ```

3. If the remote service is on an instance of the Iguazio MLOps Platform (**"the platform"**), set the following environment variables as well; replace the `<...>` placeholders with the information for your specific platform cluster:

```
V3IO_USERNAME=<username of a platform user with access to the MLRun service>
V3IO_API=<API endpoint of the webapi service endpoint; e.g., "https://default-
↪tenant.app.mycluster.iguazio.com:8444">
V3IO_ACCESS_KEY=<platform access key>
```

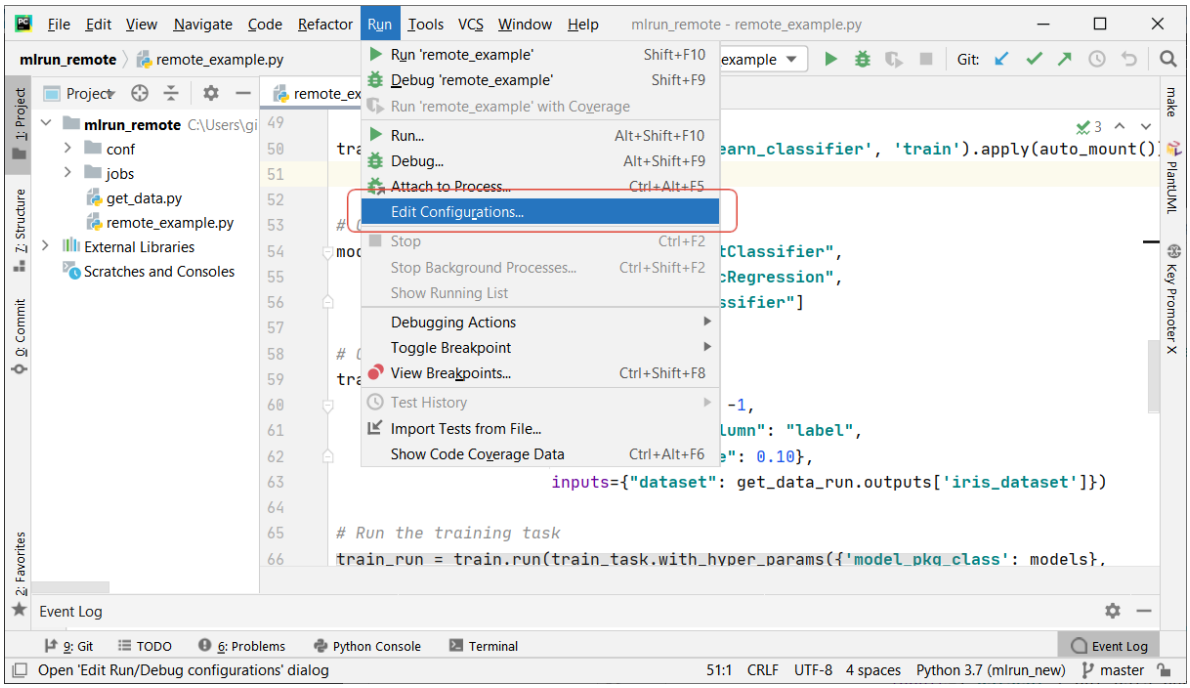
You can get the platform access key from the platform dashboard: select the user-profile picture or icon from the top right corner of any page, and select **Access Keys** from the menu. In the **Access Keys** window, either copy an existing access key or create a new key and copy it. Alternatively, you can get the access key by checking the value of the `V3IO_ACCESS_KEY` environment variable in a web-shell or Jupyter Notebook service.

IDE configuration

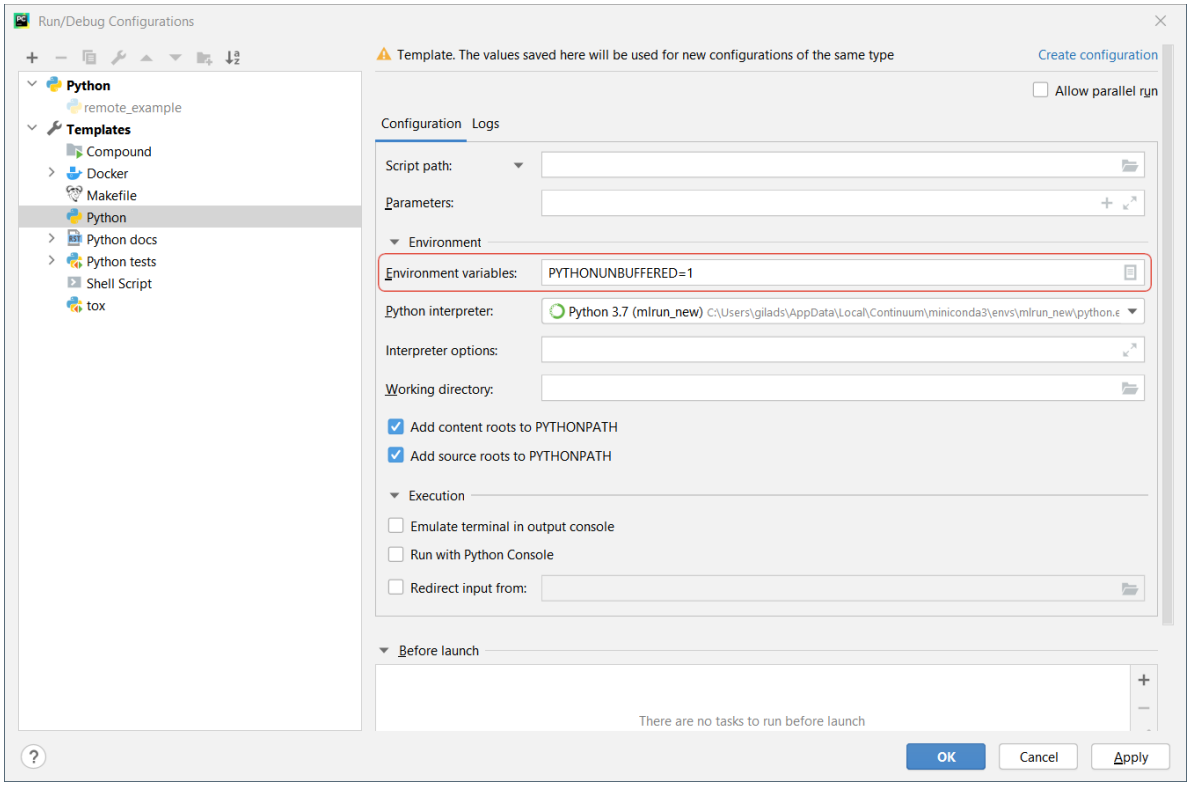
Remote environment from PyCharm

You can use PyCharm with MLRun remote by changing the environment variables configuration.

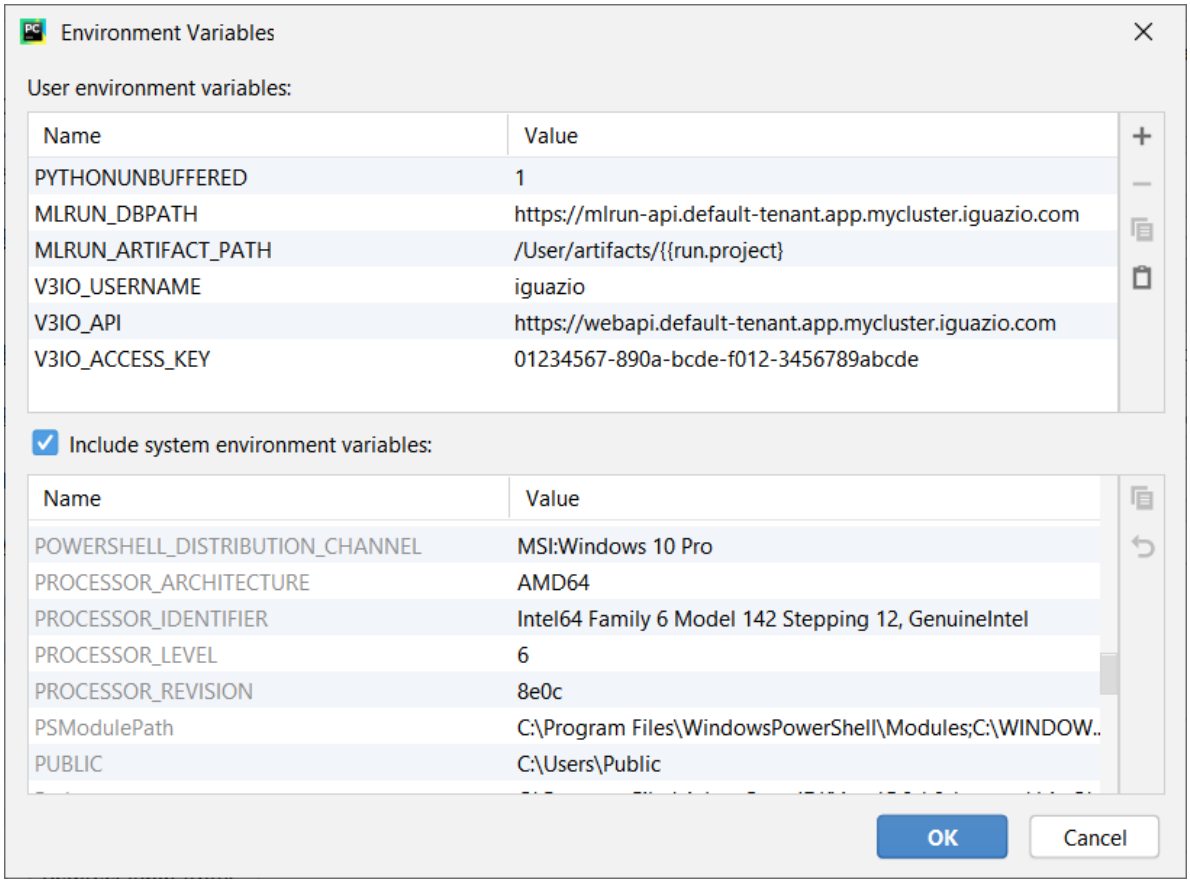
1. From the main menu, choose **Run | Edit Configurations**.



2. To set-up default values for all Python configurations, on the left-hand pane of the run/debug configuration dialog, expand the **Templates** node and select the **Python** node. The corresponding configuration template appears in the right-hand pane. Alternatively, you can edit a specific file configuration by choosing the corresponding file on the left-hand pane. Choose the **Environment Variables** edit box and expand it to edit the environment variables.



3. Add the environment variables and values of `MLRUN_DBPATH`, `MLRUN_ARTIFACT_PATH`, `V3IO_USERNAME`, `V3IO_API`, and `V3IO_ACCESS_KEY`.



Remote environment from VSCode

Create environment file

Create an environment file called `mlrun.env` in your workspace folder. Copy-paste the configuration below; replace the `<...>` placeholders to identify your remote target:

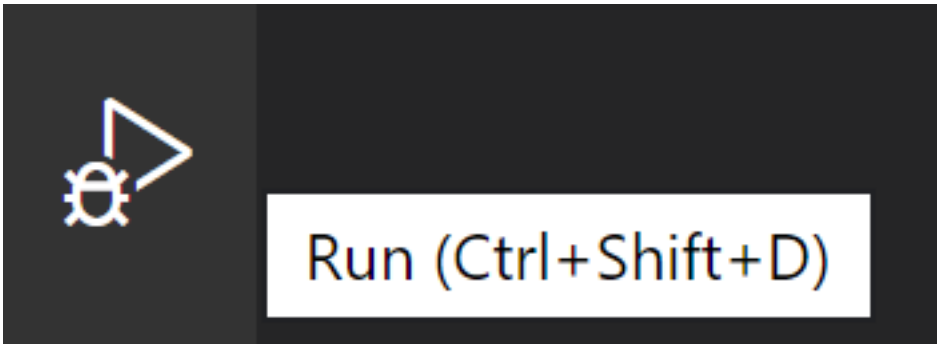
```
# Remote URL to mlrun service
MLRUN_DBPATH=<API endpoint of the MLRun APIs service endpoint; e.g., "https://mlrun-
↪api.default-tenant.app.mycluster.iguazio.com">
# Root artifact path on the remote server
MLRUN_ARTIFACT_PATH=<remote path; e.g., "/User/artifacts/{{run.project}}">
# Iguazio platform username
V3IO_USERNAME=<username of a platform user with access to the MLRun service>
# V3IO data access API URL (copy from the services screen)
V3IO_API=<API endpoint of the webapi service endpoint; e.g., "https://default-tenant.
↪app.mycluster.iguazio.com:8444">
# Iguazio V3IO data layer credentials (copy from your user settings)
V3IO_ACCESS_KEY=<platform access key>
```

Note: Make sure that you add `.env` to your `.gitignore` file. The environment file contains sensitive information that you should not store in your source control.

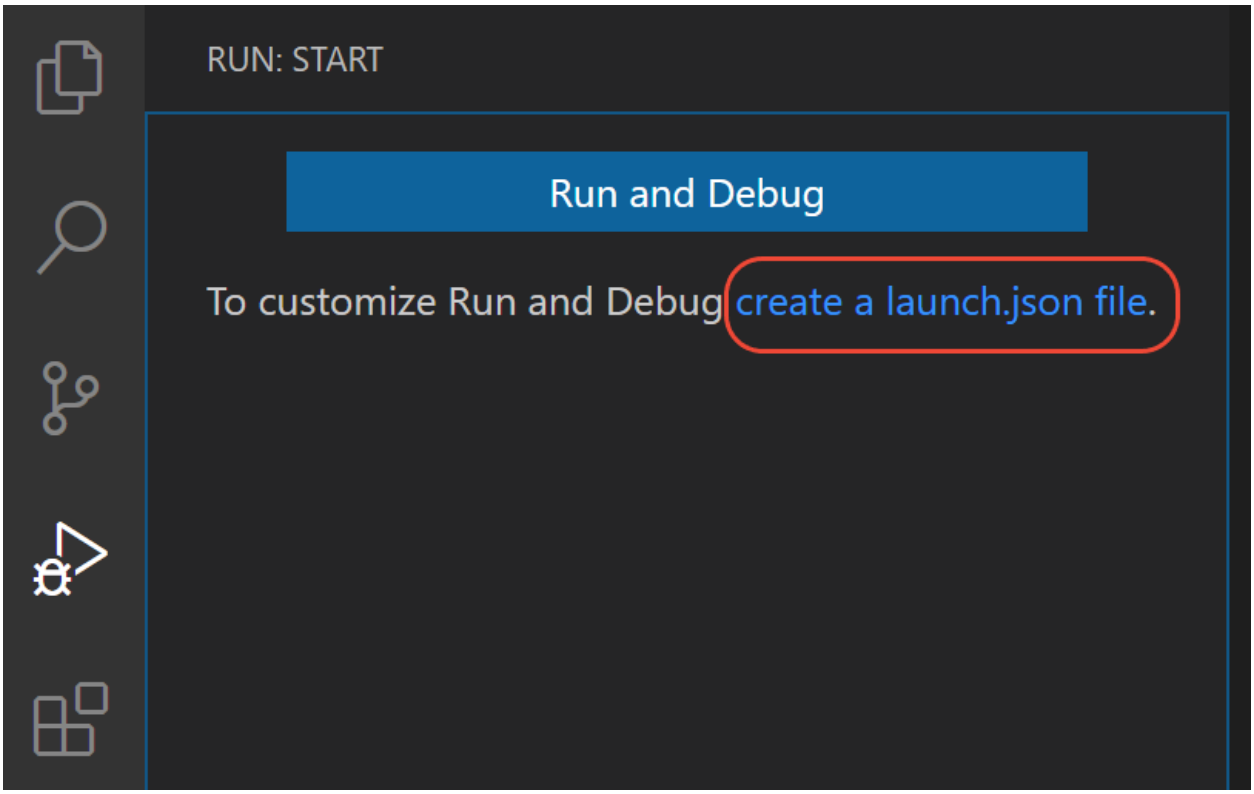
Create Python debug configuration

Create a debug configuration in VSCode. Configurations are defined in a `launch.json` file that's stored in a `.vscode` folder in your workspace.

To initialize debug configurations, first select the Run view in the sidebar:

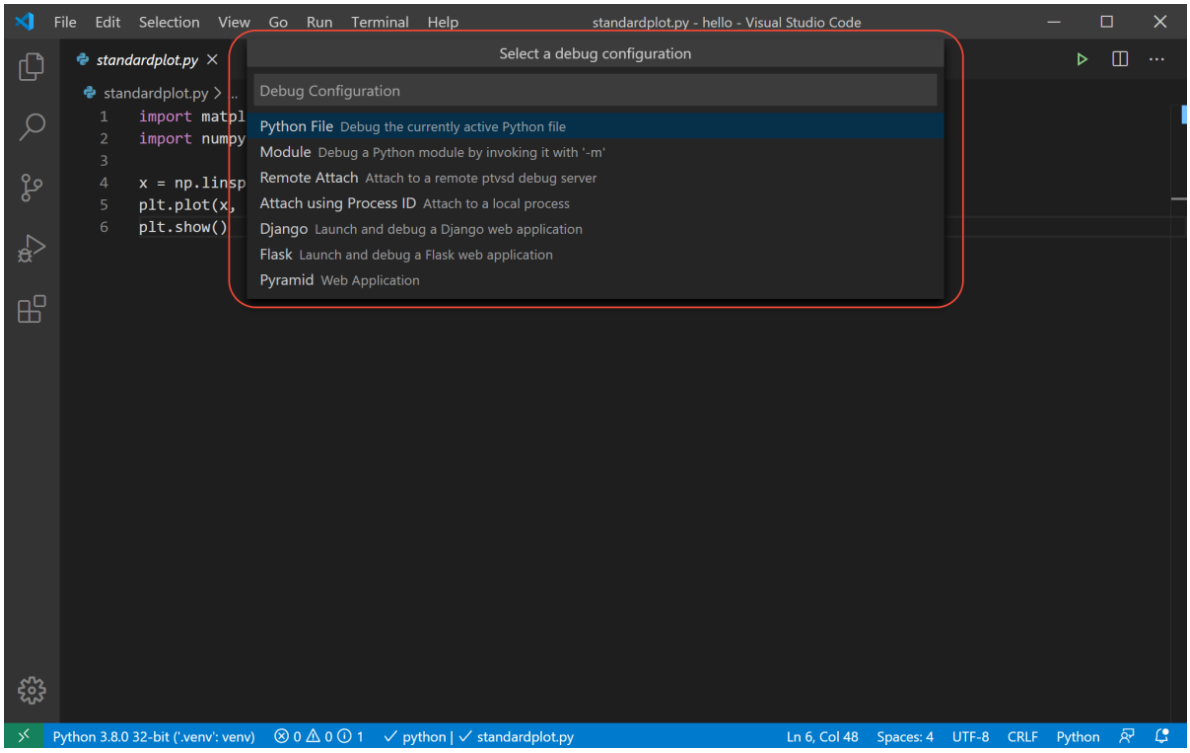


If you don't yet have any configurations defined, you'll see a button to Run and Debug, as well as a link to create a configuration (launch.json) file:



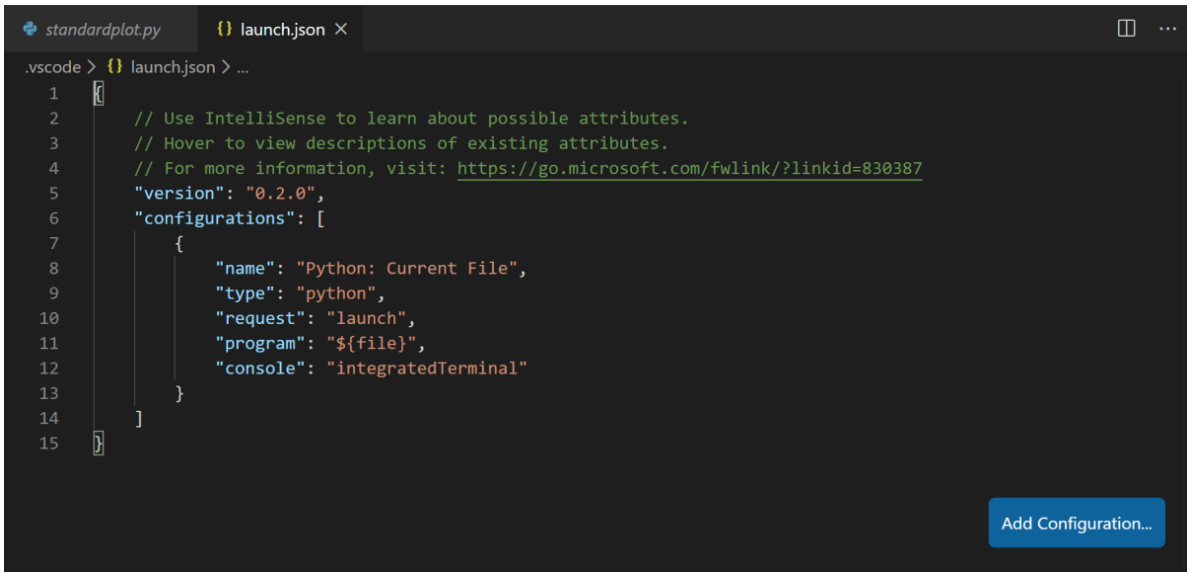
To generate a `launch.json` file with Python configurations, do the following steps:

1. Click the **create a launch.json file** link (circled in the image above) or use the **Run > Open configurations** menu command.
2. A configuration menu opens from the Command Palette. Select the type of debug configuration you want for the opened file. For now, in the **Select a debug configuration** menu that appears, select **Python File**.



Note Starting a debugging session through the Debug Panel, **F5** or **Run > Start Debugging**, when no configuration exists will also bring up the debug configuration menu, but will not create a launch.json file.

3. The Python extension then creates and opens a `launch.json` file that contains a pre-defined configuration based on what you previously selected, in this case **Python File**. You can modify configurations (to add arguments, for example), and also add custom configurations.



Set environment file in debug configuration

Add an `envFile` setting to your configuration with the value of `${workspaceFolder}/mlrun.env`

If you created a new configuration in the previous step, your `launch.json` would look as follows:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Current File",
            "type": "python",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal",
            "envFile": "${workspaceFolder}/mlrun.env"
        }
    ]
}
```

Set environment variables in a terminal

You can create a script that sets the desired environment variables before launching your IDE

Create a file `mlrun_env.sh`, and copy-paste the code below; replace the `<...>` placeholders to identify your remote target:

```
#!/usr/bin/env bash

# Remote URL to mlrun service
export MLRUN_DBPATH=<API endpoint of the MLRun APIs service endpoint; e.g., "https://
↪mlrun-api.default-tenant.app.mycluster.iguazio.com">
# Root artifact path on the remote server
export MLRUN_ARTIFACT_PATH=<remote path; e.g., "/User/artifacts/{{run.project}}">
# Iguazio platform username
export V3IO_USERNAME=<username of a platform user with access to the MLRun service>
# V3IO data access API URL (copy from the services screen)
export V3IO_API=<API endpoint of the webapi service endpoint; e.g., "https://default-
↪tenant.app.mycluster.iguazio.com:8444">
# Iguazio V3IO data layer credentials (copy from your user settings)
export V3IO_ACCESS_KEY=<platform access key>
```

In your terminal session execute:

```
source mlrun_env.sh
```

Then launch your IDE from the same terminal session.

2.1.6 Projects

A project is a container for all your work on a particular activity/application. It is the basic starting point for your work. Project definitions include all of the associated code, *functions*, *Submitting tasks/jobs to functions*, *artifacts*, lists of parameters, and secrets. You can create project definitions using the SDK or a yaml file and store those in the MLRun DB, a file, or an archive. Project jobs/workflows refer to all project resources by name, establishing a separation between configuration and code.

Projects can be mapped to `git` repositories or IDE project (in PyCharm, VSCode, etc.), which enables versioning, collaboration, and CI/CD. Project access can be restricted to a set of users and roles.

Projects can be loaded/cloned using a single command. Once the project is loaded you can execute the functions or workflows locally, on a cluster, or inside a CI/CD framework.

In this section

Create and load projects

Projects refer to a `context` directory that holds all the project code and configuration. The `context` dir is usually mapped to a `git` repository and/or to an IDE (PyCharm, VSCode, etc.) project.

There are three ways to create/load a `project` object:

- *`new_project()`* — Create a new MLRun project and optionally load it from a yaml/zip/git template.
- *`load_project()`* — Load a project from a context directory or remote git/zip/tar archive.
- *`get_or_create_project()`* — Load a project from the MLRun DB if it exists, or from a specified context/archive.

Projects can also be loaded and workflows/pipelines can be executed using the CLI (using the `mlrun project` command).

In this section

- *Creating a new project*
- *Load and run projects from context, git or archive*
- *Get a project from DB or create it (get_or_create_project)*
- *Working with Git*

Creating a new project

To define a new project from scratch, use *`new_project()`*. You must specify a `name`, location for the `context` directory (e.g. `./`) and other optional parameters (see below). The `context` dir holds the configuration, code, and workflow files. File paths in the project are relative to the context root.

```
    # create a project with local and marketplace functions
    project = mlrun.new_project("myproj", "./", init_git=True, description="my new␣
↪project")
    project.set_function('prep_data.py', 'prep-data', image='mlrun/mlrun', handler=
↪'prep_data')
    project.set_function('hub://sklearn_classifier', 'train')

    # register a simple named artifact in the project (to be used in workflows)
    data_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris.data.raw.csv'
    project.set_workflow('main', "./myflow.py")
```

(continues on next page)

(continued from previous page)

```
    # add a multi-stage workflow (./myflow.py) to the project with the name 'main'␣
↪and save the project
    project.set_artifact('data', Artifact(target_path=data_url))
    project.save()

    # run the "main" workflow (watch=True to wait for run completion)
    project.run("main", watch=True)
```

When projects are saved a `project.yaml` file with project definitions is written to the `context` dir. Alternatively you can manually create the `project.yaml` file and load it using `load_project()` or the `from_template` parameter. The generated `project.yaml` for the above project looks like:

```
kind: project
metadata:
  name: myproj
spec:
  description: my new project
  functions:
  - url: prep_data.py
    name: prep-data
    image: mlrun/mlrun
    handler: prep_data
  - url: hub://sklearn_classifier
    name: train
  workflows:
  - name: main
    path: ./myflow.py
    engine: kfp
  artifacts:
  - kind: ''
    target_path: https://s3.wasabisys.com/iguazio/data/iris/iris.data.raw.csv
    key: data
```

Projects can also be created from a template (yaml file, zip file, or git repo), allowing users to create reusable skeletons. The content of the zip/tar/git archive is copied into the context dir.

The `init_git` flag is used to initialize git in the context dir, the `remote` attribute is used to register the remote git repository URL, and the `user_project` flag indicates that the project name is unique to the user.

Example of creating a new project from a zip template:

```
    # create a project from zip, initialize a local git, and register the git remote␣
↪path
    project = mlrun.new_project("myproj", "./", init_git=True, user_project=True,
                                remote="git://github.com/mlrun/demo-xgb-project.git",
                                from_template="http://mysite/proj.zip")
    # add another marketplace function and save
    project.set_function('hub://test_classifier', 'test')
    project.save()
```

Note

- Projects are visible in the MLRun dashboard only after they're saved to the MLRun database (with `.save()`) or after the workflows are executed (with `.run()`).
- You can ensure the project name is unique per user by setting the the `user_project` parameter to `True`.

Load and run projects from context, git or archive

When a project is already created and stored in a git archive you can quickly load and use it with the *load_project()* method. `load_project` uses a local context directory (with initialized `git`) or clones a remote repo into the local dir and returns a project object.

You need to provide the path to the `context` dir and the git/zip/tar archive `url`. The `name` can be specified or taken from the project object, they can also specify `secrets` (repo credentials), `init_git` flag (to initialize git in the context dir), `clone` flag (indicating we must clone and ignore/remove local copy), and `user_project` flag (indicate the project name is unique to the user).

Example of loading a project from git and running the `main` workflow:

```
    project = mlrun.load_project("./", "git://github.com/mlrun/project-demo.git")
    project.run("main", arguments={'data': data_url})
```

Note

If the `url` parameter is not specified it searches for Git repo inside the context dir and uses its metadata, or uses the init_git=True flag to initialize a Git repo in the target context directory.

Load and run using the CLI

Loading a project from `git` into `./` :

```
mlrun project -n myproj -u "git://github.com/mlrun/project-demo.git" .
```

Running a specific workflow (`main`) from the project stored in `.` (current dir):

```
mlrun project -r main -w .
```

CLI usage details:

```
Usage: mlrun project [OPTIONS] [CONTEXT]

Options:
  -n, --name TEXT           project name
  -u, --url TEXT            remote git or archive url
  -r, --run TEXT            run workflow name of .py file
  -a, --arguments TEXT      pipeline arguments name and value tuples (with -r flag),
                            e.g. -a x=6

  -p, --artifact-path TEXT  output artifacts path if not default
  -x, --param TEXT          mlrun project parameter name and value tuples,
                            e.g. -p x=37 -p y='text'

  -s, --secrets TEXT        secrets file=<filename> or env=ENV_KEY1,..
  --init-git                for new projects init git context
  -c, --clone               force override/clone into the context dir
  --sync                    sync functions into db
  -w, --watch               wait for pipeline completion (with -r flag)
  -d, --dirty               allow run with uncommitted git changes
```

Get a project from DB or create it (`get_or_create_project`)

If you already have a project saved in the DB and you need to access/use it (for example from a different notebook or file), use the *`get_or_create_project()`* method. It first tries to read the project from the DB, and only if it doesn't exist in the DB it loads/creates it.

Note

If you update the project object from different files/notebooks/users, make sure you `.save()` your project after a change, and run `get_or_create_project` to load changes made by others.

Example:

```
    # load project from the DB (if exist) or the source repo
    project = mlrun.get_or_create_project("myproj", "./", "git://github.com/mlrun/
↪demo-xgb-project.git")
    project.pull("development")  # pull the latest code from git
    project.run("main", arguments={'data': data_url})  # run the workflow "main"
```

Working with Git

You can update the code using the standard Git process (commit, push). If you update/edit the project object you need to run `project.save()`, which updates the `project.yaml` file in your context directory, followed by pushing your updates.

You can use the standard `git` cli to `pull`, `commit`, `push`, etc. MLRun project syncs with the local git state. You can also use project methods with the same functionality. It simplifies the work for common task but does not expose the full git functionality.

- ***`pull()`*** — pull/update sources from git or tar into the context dir
- ***`create_remote()`*** — create remote for the project git
- ***`push()`*** — save project state and commit/push updates to remote git repo

For example: `proj.push(branch, commit_message, add=[])` saves the state to DB & yaml, commits updates, push

---

Note

You must push updates before you build functions or run workflows which use code from git, since the builder or containers pull the code from the git repo.

If you are using containerized Jupyter you might need to first set your Git parameters, e.g. using the following commands:

```
git config --global user.email "<my@email.com>"
git config --global user.name "<name>"
git config --global credential.helper store
```

After that you need to login once to git with your password, as well as restart the notebook.

```
project.push('master', 'some edits')
```

Using projects

You can add/update a project's functions, artifacts, or workflows using *set_function()*, *set_artifact()*, *set_workflow()*, and set various project attributes (parameters, secrets, etc.).

Use the project *run()* method to run a registered workflow using a pipeline engine (e.g. Kubeflow pipelines). The workflow executes its registered functions in a sequence/graph (DAG). The workflow can reference project parameters, secrets and artifacts by name.

Projects can also be loaded and workflows/pipelines can be executed using the CLI (using mlrun project command).

In this section

- *Updating and using project functions*
- *Run, Build, and Deploy functions*

Updating and using project functions

Projects host or link to functions that are used in job or workflow runs. You add functions to a project using *set_function()*. This registers them as part of the project definition (and Yaml file). Alternatively you can create functions using methods like *code_to_function()* and save them to the DB (under the same project). The preferred approach is to use set_function (which also records the functions in the project spec).

The *set_function()* method allow you to add/update many types of functions:

- **marketplace functions** - load/register a marketplace function into the project (func="hub://...")
- **notebook file** - convert a notebook file into a function (func="path/to/file.ipynb")
- **python file** - convert a python file into a function (func="path/to/file.py")
- **database function** - function stored in MLRun DB (func="db://project/func-name:version")
- **function yaml file** - read the function object from a yaml file (func="path/to/file.yaml")
- **inline function spec** - save the full function spec in the project definition file (func=func_object), not recommended

When loading a function from code file (py, ipynb) you should also specify a container image and the runtime kind (will use job kind as default). You can optionally specify the function handler (the function handler to invoke), and a name.

If the function is not a single file function, and it requires access to multiple files/libraries in the project, you should set the with_repo=True to add the entire repo code into the destination container during build or run time.

Note

When using with_repo=True the functions need to be deployed (function.deploy()) to build a container, unless you set project.spec.load_source_on_run=True which instructs MLRun to load the git/archive repo into the function container at run time and do not require a build (this is simpler when developing, for production its preferred to build the image with the code)

Examples:

```
    project.set_function('hub://sklearn_classifier', 'train')
    project.set_function('http://.../mynb.ipynb', 'test', image="mlrun/mlrun")
    project.set_function('./src/mycode.py', 'ingest',
```

(continues on next page)

(continued from previous page)

```
                        image='myrepo/ing:latest', with_repo=True)
    project.set_function('db://project/func-name:version')
    project.set_function('./func.yaml')
    project.set_function(func_object)
```

once functions are registered or saved in the project we can get their function object using `project.get_function(key)`.

example:

```
    # get the data-prep function, add volume mount and run it with data input
    project.get_function("data-prep").apply(v3io_mount())
    run = project.run_function("data-prep", inputs={"data": data_url})
```

Run, Build, and Deploy functions

there are a set of methods used to deploy and run project functions, those can be used interactively or inside a pipeline (inside a pipeline it will be automatically mapped to the relevant pipeline engine command).

- *`run_function()`* - Run a local or remote task as part of a local run or pipeline
- *`build_function()`* - deploy ML function, build container with its dependencies for use in runs
- *`deploy_function()`* - deploy real-time/online (nuclio or serving based) functions

You can use those methods as `project` methods, or as global (`mlrun.`) methods, the current project will be assumed for the later case.

```
run = myproject.run_function("train", inputs={"data": data_url})  # will run the
↪"train" function in myproject
run = mlrun.run_function("train", inputs={"data": data_url})  # will run the "train"␣
↪function in the current/active project
```

The first parameter in those three methods is the function name (in the project), or it can be a function object if we want to use functions we imported/created ad hoc, example:

```
# import a serving function from the marketplace and deploy a tarined model over it
serving = import_function("hub://v2_model_server", new_name="serving")
deploy = deploy_function(
    serving,
    models=[{"key": "mymodel", "model_path": train.outputs["model"]}],
)
```

Project workflows and automation

A workflow is a definition of execution of functions. It defines the order of execution of multiple dependent steps in a directed acyclic graph (DAG). A workflow can reference the project's params, secrets, artifacts, etc. It can also use a function execution output as a function execution input (which, of course, defines the order of execution).

MLRun supports running workflows on a `local` or `kubeflow` pipeline engine. The `local` engine runs the workflow as a local process, which is simpler for debuggimg and running simple/sequential tasks. The `kubeflow` ("kfp") engine runs as a task over the cluster and supports more advanced operations (conditions, branches, etc.). You can select the engine at runtime. Kubeflow-specific directives like conditions and branches are not supported by the `local` engine.

Workflows are saved/registered in the project using the *set_workflow()*.Workflows are executed using the *run()* method or using the CLI command `mlrun project`.

Refer to the ***tutorials section*** for complete examples.

In this section

- *Composing workflows*
- *Saving workflows*
- *Running workflows*

Composing workflows

Workflows are written as python functions that make use of function *operations (run, build, deploy)* operations and can access project parameters, secrets, and artifacts using *get_param()*, *get_secret()* and *get_artifact_uri()*.

For workflows to work in Kubeflow you need to add a decorator (`@dsl.pipeline(..)`) as shown below.

Example workflow:

```
from kfp import dsl
import mlrun
from mlrun.model import HyperParamOptions

funcs = {}
DATASET = "iris_dataset"

in_kfp = True


@dsl.pipeline(name="Demo training pipeline", description="Shows how to use mlrun.")
def newpipe():

    project = mlrun.get_current_project()

    # build our ingestion function (container image)
    builder = mlrun.build_function("gen-iris")

    # run the ingestion function with the new image and params
    ingest = mlrun.run_function(
        "gen-iris",
        name="get-data",
        params={"format": "pq"},
        outputs=[DATASET],
    ).after(builder)

    # train with hyper-paremeters
    train = mlrun.run_function(
        "train",
        name="train",
        params={"sample": -1, "label_column": project.get_param("label", "label"),
↪"test_size": 0.10},
        hyperparams={
            "model_pkg_class": [
                "sklearn.ensemble.RandomForestClassifier",
                "sklearn.linear_model.LogisticRegression",
```

(continues on next page)

(continued from previous page)

```
                "sklearn.ensemble.AdaBoostClassifier",
            ]
        },
        hyper_param_options=HyperParamOptions(selector="max.accuracy"),
        inputs={"dataset": ingest.outputs[DATASET]},
        outputs=["model", "test_set"],
    )
    print(train.outputs)

    # test and visualize our model
    mlrun.run_function(
        "test",
        name="test",
        params={"label_column": project.get_param("label", "label")},
        inputs={
            "models_path": train.outputs["model"],
            "test_set": train.outputs["test_set"],
        },
    )

    # deploy our model as a serverless function, we can pass a list of models to serve
    serving = mlrun.import_function("hub://v2_model_server", new_name="serving")
    deploy = mlrun.deploy_function(
        serving,
        models=[{"key": f"{DATASET}:v1", "model_path": train.outputs["model"]}],
    )

    # test out new model server (via REST API calls), use imported function
    tester = mlrun.import_function("hub://v2_model_tester", new_name="live_tester")
    mlrun.run_function(
        tester,
        name="model-tester",
        params={"addr": deploy.outputs["endpoint"], "model": f"{DATASET}:v1"},
        inputs={"table": train.outputs["test_set"]},
    )
```

Saving workflows

If you want to use workflows as part of an automated flow, save them and register them in the project. Use the *set_workflow()* method to register workflows, to specify a workflow name, the path to the workflow file, and the function `handler` name (or it looks for a handler named "pipeline"), and can set the default `engine` (local or kfp).

When setting the `embed` flag to True, the workflow code is embedded in the project file (can be used if you want to describe the entire project using a single YAML file).

You can define the schema for workflow arguments (data type, default, doc, etc.) by setting the `args_schema` with a list of **`EntrypointParam`** objects.

Example:

```
    # define agrument for the workflow
    arg = mlrun.model.EntrypointParam(
        "model_pkg_class",
        type="str",
        default="sklearn.linear_model.LogisticRegression",
```

(continues on next page)

(continued from previous page)

```
        doc="model package/algorithm",
    )

    # register the workflow in the project and save the project
    project.set_workflow("main", "./myflow.py", handler="newpipe", args_schema=[arg])
    project.save()

    # run the workflow
    project.run("main", arguments={"model_pkg_class": "sklearn.ensemble.
↪RandomForestClassifier"})
```

Running workflows

Use the *run()* method to execute workflows. Specify the workflow using its `name` or `workflow_path` (path to the workflow file) or `workflow_handler` (the workflow function handler). You can specify the input `arguments` for the workflow and can override the system default `artifact_path`.

Workflows are asynchronous by default. You can set the `watch` flag to True and the run operation blocks until completion and prints out the workflow progress. Alternatively you can use `.wait_for_completion()` on the run object.

The default workflow engine is `kfp`. You can override it by specifying the `engine` in the `run()` or `set_workflow()` methods. Using the `local` engine executes the workflow state machine locally (its functions still run as cluster jobs). If you set the `local` flag to True, the workflow uses the `local` engine AND the functions run as local process. This mode is used for local debugging of workflows.

When running workflows from a git enabled context it first verifies that there are no uncommitted git changes (to guarantee that workflows that load from git do not use old code versions). You can suppress that check by setting the `dirty` flag to True.

Examples:

```
# simple run of workflow 'main' with arguments, block until it completes (watch=True)
run = project.run("main", arguments={"param1": 6}, watch=True)

# run workflow specified with a function handler (my_pipe)
run = project.run(workflow_handler=my_pipe)
# wait for pipeline completion
run.wait_for_completion()

# run workflow in local debug mode
run = project.run(workflow_handler=my_pipe, local=True, arguments={"param1": 6})
```

Working with secrets

When executing jobs through MLRun, the code might need access to specific secrets, for example to access data residing on a data-store that requires credentials (such as a private S3 bucket), or many other similar needs.

MLRun provides some facilities that allow handling secrets and passing those secrets to execution jobs. However, it's important to understand how these facilities work, as this has implications on the level of security they provide and how much exposure they create for your secrets.

In this section

- *Overview*

- *Secret providers*
  - *Inline*
  - *Environment*
  - *File*
  - *Kubernetes*
  - *Azure Vault*

Overview

MLRun uses the concept of Tasks to encapsulate runtime parameters. Tasks are used to specify execution context such as hyper-parameters. They can also be used to pass details about secrets that are going to be used in the runtime.

To pass secret parameters, use the Task's *with_secrets()* function. For example, the following command passes secrets provided by a kubernetes secret to the execution context:

```
function = mlrun.code_to_function(
    name="secret_func",
    filename="my_code.py",
    handler="test_function",
    kind="job",
    image="mlrun/mlrun"
)
task = mlrun.new_task().with_secrets("kubernetes", ["AWS_KEY", "DB_PASSWORD"])
run = function.run(task, ...)
```

Within the code in `my_code.py`, the handler can access these secrets by using the *get_secret()* API:

```
def test_function(context, db_name):
    context.logger.info("running function")
    db_password = context.get_secret("DB_PASSWORD")
    # Rest of code can use db_password to perform processing.
    ...
```

The *with_secrets()* function tells MLRun what secrets the executed code needs to access. The MLRun framework prepares the needed infrastructure to make these secrets available to the runtime, and passes information about them to the execution framework by specifying those secrets in the spec of the runtime. For example, if running a kubernetes job, the secret keys are noted in the generated pod's spec.

The actual details of MLRun's handling of the secrets differ per the **secret provider** used. The following sections provide more details on these providers and how they handle secrets and their values.

Regardless of the type of secret provider used, the executed code uses the same *get_secret()* API to gain access to the value of the secrets passed to it, as shown in the above example.

Secret providers

As mentioned, MLRun provides the user with several secret providers. Each of those providers functions differently and has different traits with respect to what secrets can be passed and how they're handled. It's important to understand these parameters to make sure secrets are not compromised and that their secrecy is maintained.

Generally speaking, the *Inline*, *environment* and *file* providers do not guarantee confidentiality of the secret values handled by them, and should only be used for development and demo purposes. The *Kubernetes* and *Azure Vault* providers are secure and should be used for any other use-case.

Inline

The inline secrets provider is a very basic framework that should mostly be used for testing and demos. The secrets passed by this framework are exposed in the source code creating the MLRun function, as well as in the function spec, and in the generated pod specs. To add inline secrets to a job, perform the following:

```
task.with_secrets("inline", {"MY_SECRET": "12345"})
```

As can be seen, even the client code exposes the secret value. If this is used to pass secrets to a job running in a kubernetes pod the secret is also visible in the pod spec. This means that any user that can run `kubectl` and is permitted to view pod specs can also see the secret keys and their values.

Environment

Environment variables are similar to the `inline` secrets, but their client-side value is not specified directly in code but rather is extracted from a client-side environment variable. For example, if running MLRun on a Jupyter notebook and there are environment variables named `MY_SECRET` and `ANOTHER_SECRET` on Jupyter, the following codepasses those secrets to the executed runtime:

```
task.with_secrets("env", "MY_SECRET, ANOTHER_SECRET")
```

When generating the runtime execution environment (for example, pod for the `job` runtime), MLRun retrieves the value of the environment variable and places it in the pod spec. This means that a user with `kubectl` capabilities who can see pod specs can still see the secret values passed in this manner.

File

The file provider is used to pass secret values that are stored in a local file. The file needs to be made of lines, each containing a secret and its value separated by =. For example:

```
# secrets.txt
SECRET1=123456
SECRET2=abcdef
```

Use the following command to add these secrets:

```
task.with_secrets("file", "/path/to/file/secrets.txt")
```

In terms of exposure of secret values, this method is the same as for inline or env secrets.

Kubernetes

MLRun can use a Kubernetes (k8s) secret to store and retrieve secret values as required. This method is supported for all runtimes that generate k8s pods. The k8s provider creates a k8s secret per project, and can store multiple secret keys within this secret.

Populating the kubernetes secret

To populate the MLRun k8s secret with secret values, MLRun provides APIs that allow the user to perform operations on the secrets, by using the *HTTPRunDB* class. For example:

```
secrets = {'password': 'myPassw0rd', 'aws_key': '111222333'}
mlrun.get_run_db().create_project_secrets(
    "project_name",
    provider=mlrun.api.schemas.SecretProviderName.kubernetes,
    secrets=secrets
)
```

Warning: This action should not be part of the code committed to `git` or part of ongoing execution - it is only a setup action, which normally should only be executed once. After the secrets are populated, this code should be removed to protect the confidentiality of the secret values.

The *HTTPRunDB* API does not allow the user to see the secret values, but it does allow users to see the keys that belong to a given project, assuming the user has permissions on that specific project. See the *HTTPRunDB* class documentation for additional details.

When MLRun is executed in the Iguazio platform, the secret management APIs are protected by the platform such that only users with permissions to access and modify a specific project can alter its secrets.

Creating secrets in the Projects page

The Settings dialog in the Projects page, accessed with the Settings icon, has a Secrets tab where you can add secrets as key-value pairs. The secrets are automatically available to all jobs belonging to this project. Users with the Editor or Admin role can add, modify, and delete secrets, and assign new secret values. Viewers can only view the secret keys. The values themselves are not visible to any users.

Accessing the secrets

By default, any runtime not executed locally automatically gains access to all the secrets of the project it belongs to. To limit access of an executing job to a subset of these secrets, call the following:

```
task.with_secrets('kubernetes', ['password', 'aws_key'])
```

Note that only the secret keys are passed in this case, since the values are kept in the k8s secret. The MLRun framework adds environment variables to the pod spec whose value is retrieved through the `valueFrom` option, with `secretKeyRef` pointing at the secret maintained by MLRun.

As a result, this method does not expose the secret values at all, except at the actual pod executing the code where the secret value is exposed through an environment variable. This means that even a user with `kubectl` looking at the pod spec cannot see the secret values.

Users, however, can view the secrets using the following methods:

- Run `kubectl` to view the actual contents of the k8s secret.
- Perform `kubectl exec` into the running pod, and examine the environment variables.

To maintain the confidentiality of secret values, these operations must be strictly limited across the system by using k8s RBAC and ensuring that elevated permissions are granted to a very limited number of users (very few users have and use elevated permissions).

Accessing secrets in nuclio functions

The k8s secrets are passed to nuclio functions as environment variables, and their values can be retrieved directly from the environment variable of the same name. For example, to access the `aws_key` secret in a nuclio function use:

```
aws_key = os.environ.get("aws_key")
```

Azure Vault

MLRun can serve secrets from an Azure key Vault. Azure key Vaults support 3 types of entities - `keys`, `secrets` and `certificates`. MLRun only supports accessing `secret` entities.

Setting up access to Azure key vault

To enable this functionality, a secret must first be created in the k8s cluster which contains the Azure key Vault credentials. This secret should include credentials providing access to your specific Azure key Vault. To configure this, the following steps are needed:

1. Set up a key vault in your Azure subscription.
2. Create a service principal in Azure that will be granted access to the key vault. For creating a service principal through the Azure portal follow the steps listed in this page.
3. Assign a key vault access policy to the service principal, as described in this page.
4. Create a secret access key for the service principal, following the steps listed in this page. Make sure you have access to the following three identifiers:
   - Directory (tenant) id
   - Application (client) id
   - Secret key
5. Generate a k8s secret with those details. Use the following command:

```
kubectl -n <namespace> create secret generic <azure_key_vault_k8s_secret> \
   --from-literal=secret=<secret key> \
   --from-literal=tenant_id=<tenant id> \
   --from-literal=client_id=<client id>
```

Note: The names of the secret keys *must* be as shown in the above example, as MLRun queries them by these exact names.

Accessing Azure key vault secrets

Once these steps are done, use `with_secrets` in the following manner:

```
task.with_secrets(
    "azure_vault",
    {
        "name": <azure_key_vault_name>,
        "k8s_secret": <azure_key_vault_k8s_secret>,
        "secrets": [],
    },
)
```

The `name` parameter should point at your Azure key Vault name. The `secrets` parameter is a list of the secret keys to be accessed from that specific vault. If it's empty (as in the example above) then all secrets in the vault can be accessed by their key name.

For example, if the Azure Vault has a secret whose name is `MY_AZURE_SECRET` and using the above example for *`with_secrets()`*, the executed code can use the following statement to access this secret:

```
azure_secret = context.get_secret("MY_AZURE_SECRET")
```

In terms of confidentiality, the executed pod has the Azure secret provided by the user mounted to it. This means that the access-keys to the vault are visible to a user that `execs` into the pod in question. The same security rules should be followed as described in the *Kubernetes* section above.

2.1.7 MLRun serverless functions

All the executions in MLRun are based on Serverless Functions. The functions allow specifying code and all the operational aspects (image, required packages, cpu/mem/gpu resources, storage, environment, etc.). The different function runtimes take care of automatically transforming the code and spec to fully managed and elastic services over Kubernetes, which saves significant operational overhead, addresses scalability and reduces infrastructure costs.

MLRun supports:

- Real-time functions for: serving, APIs, and stream processing (based on the high-performance Nuclio engine).
- Batch functions (based on Kubernetes jobs, Spark, Dask, Horovod, etc.)

Function objects are all inclusive (code, spec, API, and metadata definitions), which allows placing them in a shared and versioned function market place. This means that different members of the team can produce or consume functions. Each function is versioned and stored in the MLRun database with a unique hash code, and gets a new hash code upon changes. There is also an open public marketplace that stores many pre-developed functions for use in your projects.

In this section

Overview

All the executions in MLRun are based on **Serverless Functions**. The functions allow specifying code and all the operational aspects (image, required packages, cpu/mem/gpu resources, storage, environment, etc.). The *different function runtimes* take care of automatically transforming the code and spec to fully managed and elastic services over Kubernetes, which saves significant operational overhead, addresses scalability and reduces infrastructure costs.

MLRun supports:

- Real-time functions for: serving, APIs, and stream processing (based on the high-performance Nuclio engine).
- Batch functions: based on Kubernetes jobs, Spark, Dask, Horovod, etc.

Function objects are all inclusive (code, spec, API, and metadata definitions), which allows placing them in a shared and versioned function market place. This means that different members of the team can produce or consume functions. Each function is versioned and stored in the MLRun database with a unique hash code, and gets a new hash code upon changes.

MLRun also has an open public marketplace that stores many pre-developed functions for use in your projects.

Function Runtimes

When you create an MLRun function you need to specify a runtime kind (e.g. `kind='job'`). Each runtime supports its own specific attributes (e.g. Jars for Spark, Triggers for Nuclio, Auto-scaling for Dask, etc.).

MLRun supports these runtimes:

Real-time runtimes:

- **nuclio** - real-time serverless functions over Nuclio
- **serving** - higher level real-time Graph (DAG) over one or more Nuclio functions

Batch runtimes:

- **handler** - execute python handler (used automatically in notebooks or for debug)
- **local** - execute a Python or shell program
- **job** - run the code in a Kubernetes Pod
- **dask** - run the code as a Dask Distributed job (over Kubernetes)
- **mpijob** - run distributed jobs and Horovod over the MPI job operator, used mainly for deep learning jobs
- **spark** - run the job as a Spark job (using Spark Kubernetes Operator)
- **remote-spark** - run the job on a remote Spark service/cluster (e.g. Iguazio Spark service)

Common attributes for Kubernetes based functions

All the Kubernetes based runtimes (Job, Dask, Spark, Nuclio, MPIJob, Serving) support a common set of spec attributes and methods for setting the PODs:

function.spec attributes (similar to k8s pod spec attributes):

- volumes
- volume_mounts
- env
- resources

- replicas
- image_pull_policy
- service_account
- image_pull_secret

common function methods:

- set_env(name, value)
- set_envs(env_vars)
- gpus(gpus, gpu_type)
- with_limits(mem, cpu, gpus, gpu_type)
- with_requests(mem, cpu)
- set_env_from_secret(name, secret, secret_key)

Distributed Functions

Many of the runtimes support horizontal scaling. You can specify the number of `replicas` or the min—max value range (for auto scaling in Dask or Nuclio). When scaling functions MLRun uses a high speed messaging protocol and shared storage (volumes, objects, databases, or streams). MLRun runtimes handle the orchestration and monitoring of the distributed task.

In this section

Dask Distributed Runtime

Quick Links

- *Running Dask Over MLRun*
- *Pipelines Using Dask, Kubeflow and MLRun*

Dask Overview

Source: Dask docs Dask is a flexible library for parallel computing in Python.

Dask is composed of two parts:

1. **Dynamic task scheduling** optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
2. **"Big Data" collections** like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask emphasizes the following virtues:

- **Familiar**: Provides parallelized NumPy array and Pandas DataFrame objects

- **Flexible**: Provides a task scheduling interface for more custom workloads and integration with other projects.
- **Native**: Enables distributed computing in pure Python with access to the PyData stack.
- **Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- **Scales up**: Runs resiliently on clusters with 1000s of cores
- **Scales down**: Trivial to set up and run on a laptop in a single process
- **Responsive**: Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans Dask collections and schedulers

Dask DataFrame mimics Pandas

```
import pandas as pd                          import dask.dataframe as dd
df = pd.read_csv('2015-01-01.csv')           df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean()          df.groupby(df.user_id).value.mean().compute()
```

Dask Array mimics NumPy - documentation

```
import numpy as np                           import dask.array as da
f = h5py.File('myfile.hdf5')                 f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])               x = da.from_array(f['/big-data'],
                                                               chunks=(1000, 1000))
x - x.mean(axis=1)                           x - x.mean(axis=1).compute()
```

Dask Bag mimics iterators, Toolz, and PySpark - documentation

```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(10, lambda pair: pair[1]).compute()
```

Dask Delayed mimics for loops and wraps custom code - documentation

```
from dask import delayed
L = []
for fn in filenames:                    # Use for loops to build up computation
    data = delayed(load)(fn)            # Delay execution of function
    L.append(delayed(process)(data))    # Build connections between variables

result = delayed(summarize)(L)
result.compute()
```

The concurrent.futures interface provides general submission of custom tasks: - documentation

```
from dask.distributed import Client
client = Client('scheduler:port')

futures = []
for fn in filenames:
    future = client.submit(load, fn)
    futures.append(future)
```

(continues on next page)

(continued from previous page)

```
summary = client.submit(summarize, futures)
summary.result()
```

Dask.distributed

`Dask.distributed` is a lightweight library for distributed computing in Python. It extends both the concurrent.futures and dask APIs to moderate sized clusters.

Motivation

Distributed serves to complement the existing PyData analysis stack. In particular it meets the following needs:

- **Low latency**: Each task suffers about 1ms of overhead. A small computation and network roundtrip can complete in less than 10ms.
- **Peer-to-peer data sharing**: Workers communicate with each other to share data. This removes central bottlenecks for data transfer.
- **Complex Scheduling**: Supports complex workflows (not just map/filter/reduce) which are necessary for sophisticated algorithms used in nd-arrays, machine learning, image processing, and statistics.
- **Pure Python**: Built in Python using well-known technologies. This eases installation, improves efficiency (for Python users), and simplifies debugging.
- **Data Locality**: Scheduling algorithms cleverly execute computations where data lives. This minimizes network traffic and improves efficiency.
- **Familiar APIs**: Compatible with the concurrent.futures API in the Python standard library. Compatible with dask API for parallel algorithms
- **Easy Setup**: As a Pure Python package distributed is pip installable and easy to set up on your own cluster.

Architecture

`Dask.distributed` is a centrally managed, distributed, dynamic task scheduler. The central `dask-scheduler` process coordinates the actions of several `dask-worker` processes spread across multiple machines and the concurrent requests of several clients.

The scheduler is asynchronous and event driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers. The event-driven and asynchronous nature makes it flexible to concurrently handle a variety of workloads coming from multiple users at the same time while also handling a fluid worker population with failures and additions. Workers communicate amongst each other for bulk data transfer over TCP.

Internally the scheduler tracks all work as a constantly changing directed acyclic graph of tasks. A task is a Python function operating on Python objects, which can be the results of other tasks. This graph of tasks grows as users submit more computations, fills out as workers complete tasks, and shrinks as users leave or become disinterested in previous results.

Users interact by connecting a local Python session to the scheduler and submitting work, either by individual calls to the simple interface `client.submit(function, *args, **kwargs)` or by using the large data collections and parallel algorithms of the parent dask library. The collections in the dask library like `dask.array` and `dask.dataframe` provide easy access to sophisticated algorithms and familiar APIs like NumPy and Pandas, while the

simple `client.submit` interface provides users with custom control when they want to break out of canned "big data" abstractions and submit fully custom workloads.

~5X Faster with Dask

Short example which demonstrates the power of Dask, in this notebook we will preform the following:

- Generate random text files
- Process the file by sorting and counting it's content
- Compare run times

Generate Random Text Files

```
import random
import string
import os

from collections import Counter
from dask.distributed import Client

import warnings
warnings.filterwarnings('ignore')
```

```
def generate_big_random_letters(filename, size):
    """
    generate big random letters/alphabets to a file
    :param filename: the filename
    :param size: the size in bytes
    :return: void
    """
    chars = ''.join([random.choice(string.ascii_letters) for i in range(size)]) #1

    with open(filename, 'w') as f:
        f.write(chars)
    pass
```

```
PATH = '/User/howto/dask/random_files'
SIZE = 10000000

for i in range(100):
    generate_big_random_letters(filename = PATH + '/file_' + str(i) + '.txt',
                                size = SIZE)
```

Set Function for Benchmark

```
def count_letters(path):
    """
    count letters in text file
    :param path:  path to file
    """
    # open file in read mode
    file = open(path, "r")

    # read the content of file
    data = file.read()

    # sort file
    sorted_file = sorted(data)

    # count file
    number_of_characters = len(sorted_file)

    return number_of_characters
```

```
def process_files(path):
    """
    list file and count letters
    :param path: path to folder with files
    """
    num_list = []
    files = os.listdir(path)

    for file in files:
        cnt = count_letters(os.path.join(path, file))
        num_list.append(cnt)

    l = num_list
    return print("done!")
```

Sort & Count Number of Letters with Python

```
%%time
PATH = '/User/howto/dask/random_files/'
process_files(PATH)
```

```
done!
CPU times: user 2min 19s, sys: 9.31 s, total: 2min 29s
Wall time: 2min 32s
```

Sort & Count Number of Letters with Dask

```
# get the dask client address
client = Client()
```

```
# list all files in folder
files = [PATH + x for x in os.listdir(PATH)]
```

```
%%time
# run the count_letter function on a list of files while using multiple workers
a = client.map(count_letters, files)
```

```
CPU times: user 13.2 ms, sys: 983 µs, total: 14.2 ms
Wall time: 12.2 ms
```

```
%%time
# gather results
l = client.gather(a)
```

```
CPU times: user 3.39 s, sys: 533 ms, total: 3.92 s
Wall time: 40 s
```

Additional Topics

Running Dask on the cluster with mlrun

The dask frameworks enables users to parallelize their python code and run it as a distributed process on Iguazio cluster and dramatically accelerate their performance. In this notebook you'll learn how to create a dask cluster and then an mlrun function running as a dask client. It also demonstrates how to run parallelize custom algorithm using Dask Delayed option

For more information on dask over kubernetes: https://kubernetes.dask.org/en/latest/

Set up the environment

```
# set mlrun api path and arrtifact path for logging
import mlrun
project_name = "dask-demo"
mlrun.set_environment(project=project_name, artifact_path = './')
```

```
('dask-demo', '/User/dask')
```

Create and Start Dask Cluster

Dask functions can be local (local workers), or remote (use containers in the cluster), in the case of remote users can specify the number of replica (optional) or leave blank for auto-scale. We use `new_function()` to define our Dask cluster and set the desired configuration of that clustered function.

if the dask workers need to access the shared file system we apply a shared volume mount (e.g. via v3io mount).

Dask function spec have several unique attributes (in addition to the standard job attributes):

- **.remote** - bool, use local or clustered dask
- **.replicas** - number of desired replicas, keep 0 for auto-scale
- **.min_replicas**, **.max_replicas** - set replicas range for auto-scale
- **.scheduler_timeout** - cluster will be killed after timeout (inactivity), default is '60 minutes'
- **.nthreads** - number of worker threads

If you want to access the dask dashboard or scheduler from remote you need to use NodePort service type (set `.service_type` to 'NodePort'), and the external IP need to be specified in mlrun configuration (mlconf.remote_host), this will be set automatically if you are running on an Iguazio cluster.

We specify the kind (dask) and the container image

```
# create an mlrun function which will init the dask cluster
dask_cluster_name = "dask-cluster"
dask_cluster = mlrun.new_function(dask_cluster_name, kind='dask', image='mlrun/ml-
↪models')
dask_cluster.apply(mlrun.mount_v3io())
```

```
<mlrun.runtimes.daskjob.DaskCluster at 0x7f7dbe4166d0>
```

```
# set range for # of replicas with replicas and max_replicas
dask_cluster.spec.min_replicas = 1
dask_cluster.spec.max_replicas = 4

# set the use of dask remote cluster (distributed)
dask_cluster.spec.remote = True
dask_cluster.spec.service_type = "NodePort"

# set dask memory and cpu limits
dask_cluster.with_requests(mem='2G', cpu='2')
```

Initialize the Dask Cluster

When we request the dask cluster `client` attribute it will verify the cluster is up and running

```
# init dask client and use the scheduler address as param in the following cell
dask_cluster.client
```

```
> 2021-01-24 23:48:54,057 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪b3c6e737-3.default-tenant:8786
> 2021-01-24 23:48:54,067 [info] using remote dask scheduler (mlrun-dask-cluster-
↪b3c6e737-3) at: tcp://mlrun-dask-cluster-b3c6e737-3.default-tenant:8786
```

```
/User/.pythonlibs/jupyter/lib/python3.7/site-packages/distributed/client.py:1129:␣
↪VersionMismatchWarning: Mismatched versions found

+---------+--------+-----------+---------+
| Package | client | scheduler | workers |
+---------+--------+-----------+---------+
| blosc   | 1.7.0  | 1.10.2    | 1.10.2  |
| lz4     | 3.1.0  | 3.1.3     | 3.1.3   |
| msgpack | 1.0.0  | 1.0.2     | 1.0.2   |
| numpy   | 1.19.2 | 1.18.1    | 1.18.1  |
| toolz   | 0.11.1 | 0.10.0    | 0.10.0  |
| tornado | 6.0.4  | 6.0.3     | 6.0.3   |
+---------+--------+-----------+---------+
Notes:
-  msgpack: Variation is ok, as long as everything is above 0.6
  warnings.warn(version_module.VersionMismatchWarning(msg[0]["warning"]))
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.0.51:8786' processes=1 threads=1, memory=4.12 GB>
```

Creating A Function Which Run Over Dask

```
# mlrun: start-code
```

Import mlrun and dask. nuclio is used just to convert the code into an mlrun function

```
import mlrun
```

```
%nuclio config kind = "job"
%nuclio config spec.image = "mlrun/ml-models"
```

```
%nuclio: setting kind to 'job'
%nuclio: setting spec.image to 'mlrun/ml-models'
```

```
from dask.distributed import Client
from dask import delayed
from dask import dataframe as dd

import warnings
import numpy as np
import os
import mlrun

warnings.filterwarnings("ignore")
```

python function code

This simple function reads a csv file using dask dataframe and run group by and describe function on the dataset and store the results as a dataset artifact

```
def test_dask(context,
              dataset: mlrun.DataItem,
              client=None,
              dask_function: str=None) -> None:

    # setup dask client from the MLRun dask cluster function
    if dask_function:
        client = mlrun.import_function(dask_function).client
    elif not client:
        client = Client()

    # load the dataitem as dask dataframe (dd)
    df = dataset.as_df(df_module=dd)

    # run describe (get statistics for the dataframe) with dask
    df_describe = df.describe().compute()

    # run groupby and count using dask
    df_grpby = df.groupby("VendorID").count().compute()

    context.log_dataset("describe",
                        df=df_grpby,
                        format='csv', index=True)
    return
```

```
# mlrun: end-code
```

Test Our Function Over Dask

Load sample data

```
DATA_URL="/User/examples/ytrip.csv"
```

```
!mkdir -p /User/examples/
!curl -L "https://s3.wasabisys.com/iguazio/data/Taxi/yellow_tripdata_2019-01_subset.
↪csv" > {DATA_URL}
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 84.9M  100 84.9M    0     0  17.3M      0  0:00:04  0:00:04 --:--:-- 19.1M
```

Convert the code to MLRun function

Use code_to_function to convert the code to MLRun and specify the configuration for the dask process (e.g. replicas, memory etc.) Note that the resource configurations are per worker

```
# mlrun will transform the code above (up to nuclio: end-code cell) into serverless␣
↪function
# which will run in k8s pods
fn = mlrun.code_to_function("test_dask",  kind='job', handler="test_dask").
↪apply(mlrun.mount_v3io())
```

Run the function

When running the function you would see a link as part of the result. click on this link takes you to the dask monitoring dashboard

```
# function URI is db://<project>/<name>
dask_uri = f'db://{project_name}/{dask_cluster_name}'
```

```
r = fn.run(handler = test_dask,
           inputs={"dataset": DATA_URL},
           params={"dask_function": dask_uri})
```

```
> 2021-01-24 23:49:37,858 [info] starting run test-dask-test_dask␣
↪uid=6410ec27b63e4a12b025696fcabc2dc9 DB=http://mlrun-api:8080
> 2021-01-24 23:49:38,069 [info] Job is running in the background, pod: test-dask-
↪test-dask-rmgkn
> 2021-01-24 23:49:41,647 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
> 2021-01-24 23:49:42,112 [info] using in-cluster config.
> 2021-01-24 23:49:42,113 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪b3c6e737-3.default-tenant:8786
> 2021-01-24 23:49:42,134 [info] using remote dask scheduler (mlrun-dask-cluster-
↪b3c6e737-3) at: tcp://mlrun-dask-cluster-b3c6e737-3.default-tenant:8786
remote dashboard: default-tenant.app.yh57.iguazio-cd0.com:30433
> 2021-01-24 23:49:48,334 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 6410ec27b63e4a12b025696fcabc2dc9 --project dask-demo , !mlrun logs␣
↪6410ec27b63e4a12b025696fcabc2dc9 --project dask-demo
> 2021-01-24 23:49:50,284 [info] run executed, status=completed
```

Track the progress in the UI

Users can view the progress and detailed information in the mlrun UI by clicking on the uid above. Also, to track the dask progress in the dask UI click on the "dashboard link" above the "client" section

Pipelines Using Dask, Kubeflow and MLRun

Create a project to host our functions, jobs and artifacts

Projects are used to package multiple functions, workflows, and artifacts. We usually store project code and definitions in a Git archive.

The following code creates a new project in a local dir and initialize git tracking on that

```
import os
import mlrun
import warnings
warnings.filterwarnings("ignore")

# set project name and dir
project_name = 'sk-project-dask'
project_dir = './project'

# specify artifacts target location
_, artifact_path = mlrun.set_environment(project=project_name)

# set project
sk_dask_proj = mlrun.new_project(project_name, project_dir, init_git=True)
```

```
> 2021-01-24 16:39:27,665 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-01-24 16:39:29,248 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
```

Init Dask Cluster

```
import mlrun
# set up function from local file
dsf = mlrun.new_function(name="mydask", kind="dask", image="mlrun/ml-models")

# set up function specs for dask
dsf.spec.remote = True
dsf.spec.replicas = 5
dsf.spec.service_type = 'NodePort'
dsf.with_limits(mem="6G")
dsf.spec.nthreads = 5
```

```
> 2021-01-24 16:39:36,831 [info] using in-cluster config.
```

```
# apply mount_v3io over our function so that our k8s pod which run our function
# will be able to access our data (shared data access)
dsf.apply(mlrun.mount_v3io())
```

```
<mlrun.runtimes.daskjob.DaskCluster at 0x7f5dfe154550>
```

```
dsf.save()
```

```
'52f5dcddb916b12943e9d44e9e2b75f48e286ec7'
```

```
# init dask cluster
dsf.client
```

```
> 2021-01-24 20:15:37,716 [info] trying dask client at: tcp://mlrun-mydask-997e6385-a.
↪default-tenant:8786
> 2021-01-24 20:15:48,564 [warning] remote scheduler at tcp://mlrun-mydask-997e6385-a.
↪default-tenant:8786 not ready, will try to restart Timed out trying to connect to
↪'tcp://mlrun-mydask-997e6385-a.default-tenant:8786' after 10 s: Timed out trying to␣
↪connect to 'tcp://mlrun-mydask-997e6385-a.default-tenant:8786' after 10 s: [Errno -
↪2] Name or service not known
> 2021-01-24 20:15:54,442 [info] using remote dask scheduler (mlrun-mydask-b4eb4ec5-
↪8) at: tcp://mlrun-mydask-b4eb4ec5-8.default-tenant:8786
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.0.53:8786' processes=0 threads=0, memory=0 B>
```

Load and run a functions

load the function object from .py .yaml file or function hub (marketplace)

```
# load function from the marketplace
sk_dask_proj.set_function('hub://describe_dask', name='describe')
sk_dask_proj.set_function('hub://sklearn_classifier_dask', name='dask_classifier')
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7f5dfc3b8d90>
```

```
sk_dask_proj.set_function('/User/dask/04-describe.py', name='describe', kind='job',␣
↪image='mlrun/ml-models')
```

Create a Fully Automated ML Pipeline

Add more functions to our project to be used in our pipeline (from the functions hub/marketplace)

Describe data, train and eval model with dask

Define and save a pipeline

The following workflow definition will be written into a file, it describes a Kubeflow execution graph (DAG) and how functions and data are connected to form an end to end pipeline.

- Ingest data
- Describe data
- Train, test and evaluate with dask

Check the code below to see how functions objects are initialized and used (by name) inside the workflow. The `workflow.py` file has two parts, initialize the function objects and define pipeline dsl (connect the function inputs and outputs).

> Note: the pipeline can include CI steps like building container images and deploying models as illustrated in the following example.

```
%%writefile project/workflow.py
from kfp import dsl
from mlrun import mount_v3io

# params
funcs       = {}
LABELS      = "label"
DROP        = 'congestion_surcharge'
#DATA_URL    = "/User/iris.csv"
DATA_URL    = "/User/iris.csv"
DASK_CLIENT = "db://sk-project-dask/mydask"

# init functions is used to configure function resources and local settings
def init_functions(functions: dict, project=None, secrets=None):
    for f in functions.values():
        f.apply(mount_v3io())
        pass

@dsl.pipeline(
    name="Demo training pipeline",
    description="Shows how to use mlrun"
)
def kfpipeline():

    # describe data
    describe = funcs['describe'].as_step(
        params={"dask_function"  : DASK_CLIENT},
        inputs={"dataset"       : DATA_URL}
    )

    # get data, train, test and evaluate
    train = funcs['dask_classifier'].as_step(
        name="train",
        handler="train_model",
        params={"label_column"    : LABELS,
                "dask_function"    : DASK_CLIENT,
                "test_size"       : 0.10,
                "model_pkg_class" : "sklearn.ensemble.RandomForestClassifier",
                "drop_cols"       : DROP},
        inputs={"dataset"         : DATA_URL},
        outputs=['model', 'test_set']
```

(continues on next page)

(continued from previous page)

```
    )

    train.after(describe)
```

```
Overwriting project/workflow.py
```

```
# register the workflow file as "main", embed the workflow code into the project YAML
sk_dask_proj.set_workflow('main', 'workflow.py', embed=True)
```

Save the project definitions to a file (project.yaml), it is recommended to commit all changes to a Git repo.

```
sk_dask_proj.save()
```

Run a pipeline workflow

use the `run` method to execute a workflow, you can provide alternative arguments and specify the default target for workflow artifacts. The workflow ID is returned and can be used to track the progress or you can use the hyperlinks

> Note: The same command can be issued through CLI commands: `mlrun project my-proj/ -r main -p "v3io:///users/admin/mlrun/kfp/{{workflow.uid}}/"`

The `dirty` flag allows us to run a project with uncommitted changes (when the notebook is in the same git dir it will always be dirty) The `watch` flag will wait for the pipeline to complete and print results

```
artifact_path = os.path.abspath('./pipe/{{workflow.uid}}')
run_id = sk_dask_proj.run(
    'main',
    arguments={},
    artifact_path=artifact_path,
    dirty=False, watch=True)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-01-24 21:30:12,077 [info] Pipeline run id=c1b351fc-073b-4cdd-a6c3-fc167afbce8e,
↪ check UI or DB for progress
> 2021-01-24 21:30:12,079 [info] waiting for pipeline run completion
```

```
<IPython.core.display.HTML object>
```

back to top

MPIJob and Horovod Runtime

Running distributed workloads

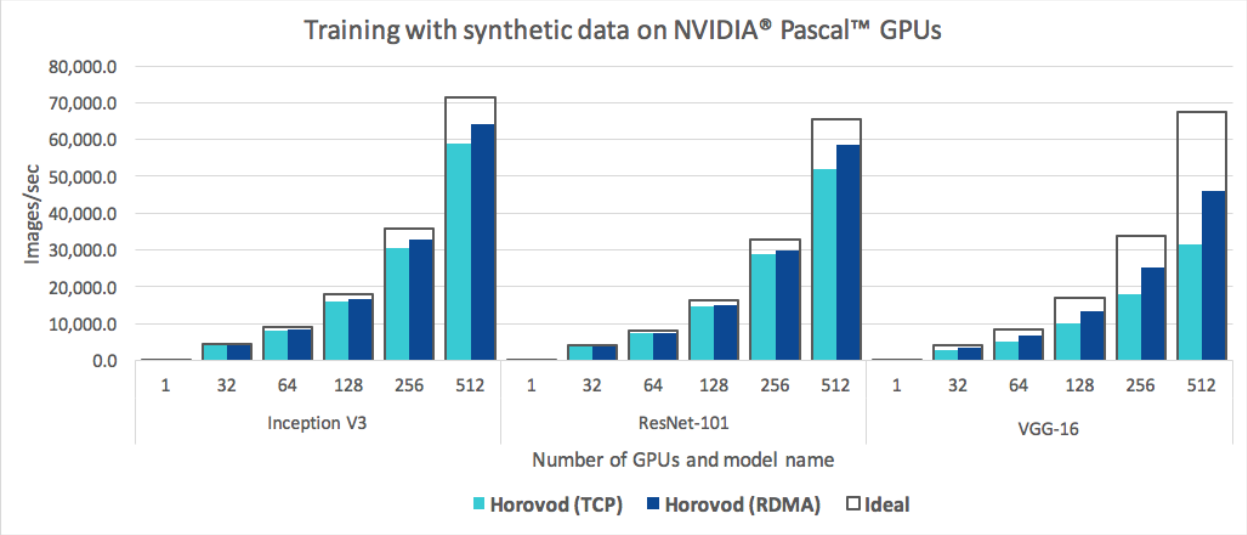
Training a Deep Neural Network is a hard task. With growing datasets, wider and deeper networks, training our Neural Network can require a lot of resources (CPUs / GPUs / Mem and Time).

There are two main reasons why we would like to distribute our Deep Learning workloads:

1. **Model Parallelism** — The **Model** is too big to fit a single GPU.In this case the model contains too many parameters to hold within a single GPU.To negate this we can use strategies like **Parameter Server** or slicing the model into slices of consecutive layers which we can fit in a single GPU.Both strategies require **Synchronization** between the layers held on different GPUs / Parameter Server shards.

2. **Data Parallelism** — The **Dataset** is too big to fit a single GPU.Using methods like **Stochastic Gradient Descent** we can send batches of data to our models for gradient estimation. This comes at the cost of longer time to converge since the estimated gradient may not fully represent the actual gradient.To increase the likelihood of estimating the actual gradient we could use bigger batches, by sending small batches to different GPUs running the same Neural Network, calculating the batch gradient and then running a **Synchronization Step** to calculate the average gradient over the batches and update the Neural Networks running on the different GPUs.

   It is important to understand that the act of distribution adds extra **Synchronization Costs** which may vary according to your cluster's configuration. As the gradients and NN needs to be propagated to each GPU in the cluster every epoch (or a number of steps), Networking can become a bottleneck and sometimes different configurations need to be used for optimal performance. **Scaling Efficiency** is the metric used to show by how much each additional GPU should benefit the training process with Horovod showing up to 90% (When running with a well written code and good parameters).



How can we distribute our training?

There are two different cluster configurations (which can be combined) we need to take into account.

- **Multi Node** — GPUs are distributed over multiple nodes in the cluster.
- **Multi GPU** — GPUs are within a single Node.

In this demo we show a **Multi Node Multi GPU** — **Data Parallel** enabled training using Horovod.However, you should always try and use the best distribution strategy for your use case (due to the added costs of the distribution itself, ability to run in an optimized way on specific hardware or other considerations that may arise).

How Horovod works?

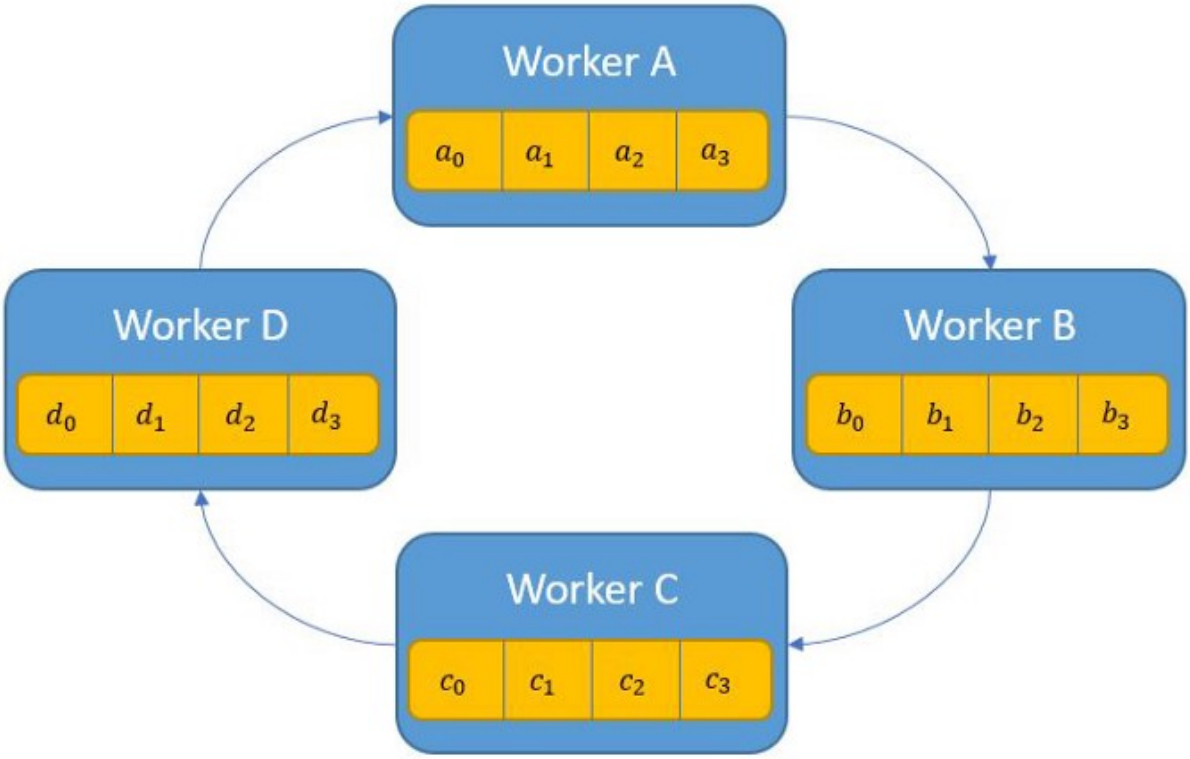
Horovod's primary motivation is to make it easy to take a single-GPU training script and successfully scale it to train across many GPUs in parallel. This has two aspects:

- How much modification does one have to make to a program to make it distributed, and how easy is it to run it?
- How much faster would it run in distributed mode?

Horovod Supports TensorFlow, Keras, PyTorch, and Apache MXNet.

in MLRun we use Horovod with MPI in order to create cluster resources and allow for optimized networking.**Note:** Horovod and MPI may use NCCL when applicable which may require some specific configuration arguments to run optimally.

Horovod uses this MPI and NCCL concepts for distributed computation and messaging to quickly and easily synchronize between the different nodes or GPUs.



Horovod will run your code on all the given nodes (Specific node can be addressed via `hvd.rank()`) while using an `hvd.DistributedOptimizer` wrapper to run the **synchronization cycles** between the copies of your Neural

Network running at each node.

Note: Since all the copies of your Neural Network must be the same, Your workers will adjust themselves to the rate of the slowest worker (simply by waiting for it to finish the epoch and receive its updates). Thus try not to make a specific worker do a lot of additional work on each epoch (Like a lot of saving, extra calculations, etc...) since this can affect the overall training time.

How do we integrate TF2 with Horovod?

As it's one of the main motivations, integration is fairly easy and requires only a few steps: (You can read the full instructions for all the different frameworks on Horovod's documentation website).

1. Run `hvd.init()`.
2. Pin each GPU to a single process. With the typical setup of one GPU per process, set this to local rank. The first process on the server will be allocated the first GPU, the second process will be allocated the second GPU, and so forth.

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
```

1. Scale the learning rate by the number of workers.Effective batch size in synchronous distributed training is scaled by the number of workers. An increase in learning rate compensates for the increased batch size.
2. Wrap the optimizer in `hvd.DistributedOptimizer`.The distributed optimizer delegates gradient computation to the original optimizer, averages gradients using allreduce or allgather, and then applies those averaged gradients.For TensorFlow v2, when using a `tf.GradientTape`, wrap the tape in `hvd.DistributedGradientTape` instead of wrapping the optimizer.
3. Broadcast the initial variable states from rank 0 to all other processes.This is necessary to ensure consistent initialization of all workers when training is started with random weights or restored from a checkpoint.For TensorFlow v2, use `hvd.broadcast_variables` after models and optimizers have been initialized.
4. Modify your code to save checkpoints only on worker 0 to prevent other workers from corrupting them.For TensorFlow v2, construct a `tf.train.Checkpoint` and only call `checkpoint.save()` when `hvd.rank() == 0`.

You can go to Horovod's Documentation to read more about horovod.

Image classification use case

See the end to end **Image Classification with Distributed Training Demo**

Spark Operator Runtime

Using spark operator for running spark job over k8s.

The `spark-on-k8s-operator` allows Spark applications to be defined in a declarative manner and supports one-time Spark applications with `SparkApplication` and cron-scheduled applications with `ScheduledSparkApplication`.

When sending a request with MLRun to Spark operator the request contains your full application configuration including the code and dependencies to run (packaged as a docker image or specified via URIs), the infrastructure parameters, (e.g. the memory, CPU, and storage volume specs to allocate to each Spark executor), and the Spark configuration.

Kubernetes takes this request and starts the Spark driver in a Kubernetes pod (a k8s abstraction, just a docker container in this case). The Spark driver can then directly talk back to the Kubernetes master to request executor pods, scaling them up and down at runtime according to the load if dynamic allocation is enabled. Kubernetes takes care of the bin-packing of the pods onto Kubernetes nodes (the physical VMs), and will dynamically scale the various node pools to meet the requirements.

When using Spark operator the resources will be allocated per task, means scale down to zero when the task is done.

```
import mlrun
import os

# set up new spark function with spark operator
# command will use our spark code which needs to be located on our file system
# the name param can have only non capital letters (k8s convention)
read_csv_filepath = os.path.join(os.path.abspath('.'), 'spark_read_csv.py')
sj = mlrun.new_function(kind='spark', command=read_csv_filepath, name='sparkreadcsv')

# set spark driver config (gpu_type & gpus=<number_of_gpus>  supported too)
sj.with_driver_limits(cpu="1300m")
sj.with_driver_requests(cpu=1, mem="512m")

# set spark executor config (gpu_type & gpus=<number_of_gpus> are supported too)
sj.with_executor_limits(cpu="1400m")
sj.with_executor_requests(cpu=1, mem="512m")

# adds fuse, daemon & iguazio's jars support
sj.with_igz_spark()

# args are also supported
sj.spec.args = ['-spark.eventLog.enabled','true']

# add python module
sj.spec.build.commands = ['pip install matplotlib']

# Number of executors
sj.spec.replicas = 2
```

```
# Rebuilds the image with MLRun - needed in order to support artifactlogging etc
sj.deploy()
```

```
# Run task while setting the artifact path on which our run artifact (in any) will be␣
↪saved
sj.run(artifact_path='/User')
```

Spark Code (spark_read_csv.py)

```
from pyspark.sql import SparkSession
from mlrun import get_or_create_ctx

context = get_or_create_ctx("spark-function")

# build spark session
spark = SparkSession.builder.appName("Spark job").getOrCreate()

# read csv
df = spark.read.load('iris.csv', format="csv",
                     sep=",", header="true")

# sample for logging
df_to_log = df.describe().toPandas()

# log final report
context.log_dataset("df_sample",
                     df=df_to_log,
                     format="csv")
spark.stop()
```

Nuclio real-time functions

Nuclio is a high-performance "serverless" framework focused on data, I/O, and compute intensive workloads. It is well integrated with popular data science tools, such as Jupyter and Kubeflow; supports a variety of data and streaming sources; and supports execution over CPUs and GPUs.

You can use Nuclio through a fully managed application service (in the cloud or on-prem) in the Iguazio Data Science Platform. MLRun serving utilizes serverless Nuclio functions to create multi-stage real-time pipelines.

The underlying Nuclio serverless engine uses a high-performance parallel processing engine that maximizes the utilization of CPUs and GPUs, supports 13 protocols and invocation methods (for example, HTTP, Cron, Kafka, Kinesis), and includes dynamic auto-scaling for HTTP and streaming. Nuclio and MLRun support the full life cycle, including auto-generation of micro-services, APIs, load-balancing, logging, monitoring, and configuration management—such that developers can focus on code, and deploy to production faster with minimal work.

Nuclio is extremely fast: a single function instance can process hundreds of thousands of HTTP requests or data records per second. To learn more about how Nuclio works, see the Nuclio architecture documentation.

Nuclio is secure: Nuclio is integrated with Kaniko to allow a secure and production-ready way of building Docker images at run time.

Read more in the Nuclio documentation and the open-source MLRun library.

Why another "serverless" project?

None of the existing cloud and open-source serverless solutions addressed all the desired capabilities of a serverless framework:

- Real-time processing with minimal CPU/GPU and I/O overhead and maximum parallelism
- Native integration with a large variety of data sources, triggers, processing models, and ML frameworks
- Stateful functions with data-path acceleration
- Simple debugging, regression testing, and multi-versioned CI/CD pipelines
- Portability across low-power devices, laptops, edge and on-prem clusters, and public clouds
- Open-source but designed for the enterprise (including logging, monitoring, security, and usability)

Nuclio was created to fulfill these requirements. It was intentionally designed as an extendable open-source framework, using a modular and layered approach that supports constant addition of triggers and data sources, with the hope that many will join the effort of developing new modules, developer tools, and platforms for Nuclio.

Node affinity for MLRun jobs

Node affinity can be applied to MLRun to determine on which nodes they can be placed. The rules are defined using custom labels on nodes and label selectors. Node affinity allows towards Spot or On Demand groups of nodes.

In this section

- *On demand vs spot*
- *Stateless and stateful applications*
- *Node selector*

On demand vs spot

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) Cloud. Using Amazon EC2 eliminates your need to invest in hardware up front, so you can develop and deploy applications faster.

Using the Iguazio platform you can deploy two different kinds of EC2 instances, on-demand and spot. On-Demand Instances provide full control over the EC2 instance lifecycle. You decide when to launch, stop, hibernate, start, reboot, or terminate it. With spot instances you request EC2 capacity from specific availability zones and is susceptible to spot capacity availability. This is a good choice if you can be flexible about when your applications run and if your applications can be interrupted.

Stateless and stateful applications

When deploying your MLRun jobs to specific nodes, please take into consideration that on demand nodes are best designed to run stateful applications while spot nodes are best designed to stateless applications. MLRun jobs which are stateful, and are assigned to run on spot nodes, may be subject to interruption and will to be designed so that the job/function state will be saved when scaling to zero.

Node selector

Using the **Node Selector** you can assign MLRun jobs to specific nodes within the cluster. **Node Selector** is available for all modes of deployment in the platform including the platform UI, command line, and programmable interfaces.

To assign MLRun jobs to specific nodes you use the Kubernetes node label `app.iguazio.com/lifecycle` with the values of:

- preemptible – assign to EC2 Spot instances
- non-preemptible – assign to EC2 On Demand instances

Note

By default Iguazio uses the key:value pair

`app.iguazio.com/lifecycle = preemptible`

or

`app.iguazio.com/lifecycle = non-preemptible`

to determine spot or on demand nodes.

You can use multiple labels to assign MLRun jobs to specific nodes. However, when you use multiple labels a logical `and` is performed on the labels.

Note

- Do not use node specific labels as this may result in eliminating all possible nodes.
- When assigning MLRun jobs to Spot instances it is the user's responsibility to deal with preempting issues within the running application/function.

To assign an MLRun job to a node:

1. From the platform dashboard, press projects in the left menu pane.
2. Press on a project, and then press Jobs and Workflows.
3. Press New Job, or select a job from the list of running jobs.
4. Scroll to and open the Resources pane.
5. In the **Node Selector** section, press **+**.
6. Enter a **key:value** pair. For example:

   or

   When complete press **Run now** or **Schedule for later**.

Assign an MLRun job to a node using the SDK

You can use node selection using the SDK by adding the key:value pairs in your Jupyter notebook. For On demand use the following function:

```
func.with_node_selection(node_selector={'app.iguazio.com/lifecycle':
'non-preemptible'})
```

For Spot instances use the following function:

```
func.with_node_selection(node_selector={'app.iguazio.com/lifecycle':
'preemptible'})
```

2.1.8 Data stores and feature store

One of the biggest challenge in distributed systems is handling data given the different access methods, APIs, and authentication mechanisms across types and providers.

MLRun provides three main abstractions to access structured and unstructured data:

- *Data Store* — defines a storage provider (e.g. file system, S3, Azure blob, Iguazio v3io, etc.)
- *Data items* — represent a data item or collection of such (file, dir, table, etc.)
- *Artifacts* — Metadata describing one or more data items. see Artifacts.

Working with the abstractions enable you to securely access different data sources through a single API, many continuance methods (e.g. to/from DataFrame, get, download, list, ..), automated data movement, and versioning.

In this section

Data stores

In this section

- *Shared data stores*
- *Storage credentials and parameters*
  - *v3io*
  - *S3*
  - *Azure Blob storage*
  - *Google cloud storage*

Shared data store

MLRun supports multiple data stores. (More can easily added by extending the `DataStore` class.) Data stores are referred to using the schema prefix (e.g. `s3://my-bucket/path`). The currently supported schemas and their urls:

- **files** — local/shared file paths, format: `/file-dir/path/to/file`
- **http, https** — read data from HTTP sources (read-only), format: `https://host/path/to/file`
- **s3** — S3 objects (AWS or other endpoints), format: `s3://<bucket>/path/to/file`
- **v3io, v3ios** — Iguazio v3io data fabric, format: `v3io://[<remote-host>]/<data-container>/path/to/file`
- **az** — Azure Blob storage, format: `az://<bucket>/path/to/file`
- **gs, gcs** — Google Cloud Storage objects, format: `gs://<bucket>/path/to/file`
- **store** — MLRun versioned artifacts *(see Artifacts)*, format: `store://artifacts/<project>/<artifact-name>[:tag]`

- **memory** — in memory data registry for passing data within the same process, format `memory://key`, use `mlrun.datastore.set_in_memory_item(key, value)` to register in memory data items (byte buffers or DataFrames).

Storage credentials and parameters

Data stores might require connection credentials. These can be provided through environment variables or project/job context secrets. The exact credentials depend on the type of the data store and are listed in the following table. Each parameter specified can be provided as an environment variable, or as a project-secret that has the same key as the name of the parameter.

MLRun jobs executed remotely run in independent pods, with their own environment. When setting an environment variable in the development environment (for example Jupyter), this has no effect on the executing pods. Therefore, before executing jobs that require access to storage credentials, these need to be provided by assigning environment variables to the MLRun runtime itself, assigning secrets to it, or placing the variables in project-secrets.

Warning: Passing secrets as environment variables to runtimes is discouraged, as they are exposed in the pod spec. Refer to *Working with secrets* for details on secret handling in MLRun.

For example, running a function locally:

```
# Access object in AWS S3, in the "input-data" bucket
source_url = "s3://input-data/input_data.csv"

os.environ["AWS_ACCESS_KEY_ID"] = "<access key ID>"
os.environ["AWS_SECRET_ACCESS_KEY"] = "<access key>"

# Execute a function that reads from the object pointed at by source_url.
# When running locally, the function can use the local environment variables.
local_run = func.run(name='aws_test', inputs={'source_url': source_url}, local=True)
```

Running the same function remotely:

```
# Executing the function remotely using env variables (not recommended!)
func.set_env("AWS_ACCESS_KEY_ID", "<access key ID>").set_env("AWS_SECRET_ACCESS_KEY",
↪"<access key>")
remote_run = func.run(name='aws_test', inputs={'source_url': source_url})

# Using project-secrets (recommended) - project secrets are automatically mounted to␣
↪project functions
secrets = {"AWS_ACCESS_KEY_ID": "<access key ID>", "AWS_SECRET_ACCESS_KEY": "<access␣
↪key>"}
db = mlrun.get_run_db()
db.create_project_secrets(project=project_name, provider="kubernetes",␣
↪secrets=secrets)

remote_run = func.run(name='aws_test', inputs={'source_url': source_url})
```

The following sections list the credentials and configuration parameters applicable to each storage type.

v3io

When running in an Iguazio system, MLRun automatically configures executed functions to use `v3io` storage, and passes the needed parameters (such as access-key) for authentication. Refer to the *auto-mount* section for more details on this process.

In some cases, the v3io configuration needs to be overridden. The following parameters can be configured:

- `V3IO_API` — URL pointing to the v3io web-API service.
- `V3IO_ACCESS_KEY` — access key used to authenticate with the web API.
- `V3IO_USERNAME` — the user-name authenticating with v3io. While not strictly required when using an access-key to authenticate, it is used in several use-cases, such as resolving paths to the home-directory.

S3

- `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` — access key parameters
- `S3_ENDPOINT_URL` — the S3 endpoint to use. If not specified, it defaults to AWS. For example, to access a storage bucket in Wasabi storage, use `S3_ENDPOINT_URL = "https://s3.wasabisys.com"`

Azure Blob storage

The Azure Blob storage can utilize several methods of authentication. Each requires a different set of parameters as listed here:

Authentication method	Parameters
Connection string	`AZURE_STORAGE_CONNECTION_STRING`
SAS token	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_SAS_TOKEN`
Account key	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_ACCOUNT_KEY`
Service principal with a client secret	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_CLIENT_IDAZURE_STORAGE_CLIENT_SEC`

Google cloud storage

- `GOOGLE_APPLICATION_CREDENTIALS` — path to the application credentials to use (in the form of a JSON file). This can be used if this file is located in a location on shared storage, accessible to pods executing MLRun jobs.
- `GCP_CREDENTIALS` — when the credentials file cannot be mounted to the pod, this environment variable may contain the contents of this file. If configured in the function pod, MLRun dumps its contents to a temporary file and points `GOOGLE_APPLICATION_CREDENTIALS` at it.

Data items

When running jobs or pipelines, data is passed using the *DataItem* objects. Data items objects abstract away the data backend implementation, provide a set of convenience methods (`.as_df`, `.get`, `.show`, ..), and enable auto logging/versioning of data and metadata.

Example function:

```
def prep_data(context, source_url: mlrun.DataItem, label_column='label'):
    # Convert the DataItem to a Pandas DataFrame
    df = source_url.as_df()
    df = df.drop(label_column, axis=1).dropna()
    context.log_dataset('cleaned_data', df=df, index=False, format='csv')
```

Running the function:

```
prep_data_run = data_prep_func.run(name='prep_data',
                                   handler=prep_data,
                                   inputs={'source_url': source_url},
                                   params={'label_column': 'userid'})
```

In order to call the function with an `input` you can use the `inputs` dictionary attribute. In order to pass a simple parameter, use the `params` dictionary attribute. The input value is the specific item uri (per data store schema) as explained in Shared data stores.

Reading the data results from the run, you can easily get a run output artifact as a `DataItem` (so that you can view/use the artifact) using:

```
# read the data locally as a Dataframe
prep_data_run.artifact('cleaned_data').as_df()
```

The *DataItem* supports multiple convenience methods such as:

- **get**(), **put**() - to read/write data
- **download**(), **upload**() - to download/upload files
- **as_df**() - to convert the data to a DataFrame object
- **local** - to get a local file link to the data (will be downloaded locally if needed)
- **listdir**(), **stat** - file system like methods
- **meta** - access to the artifact metadata (in case of an artifact uri)
- **show**() - visualizes the data in Jupyter (as image, html, etc.)

See the ***DataItem*** class documentation for details.

In order to get a DataItem object from a url use *get_dataitem()* or *get_object()* (returns the `DataItem.get()`).

For example:

```
df = mlrun.get_dataitem('s3://demo-data/mydata.csv').as_df()
print(mlrun.get_object('https://my-site/data.json'))
```

Feature store

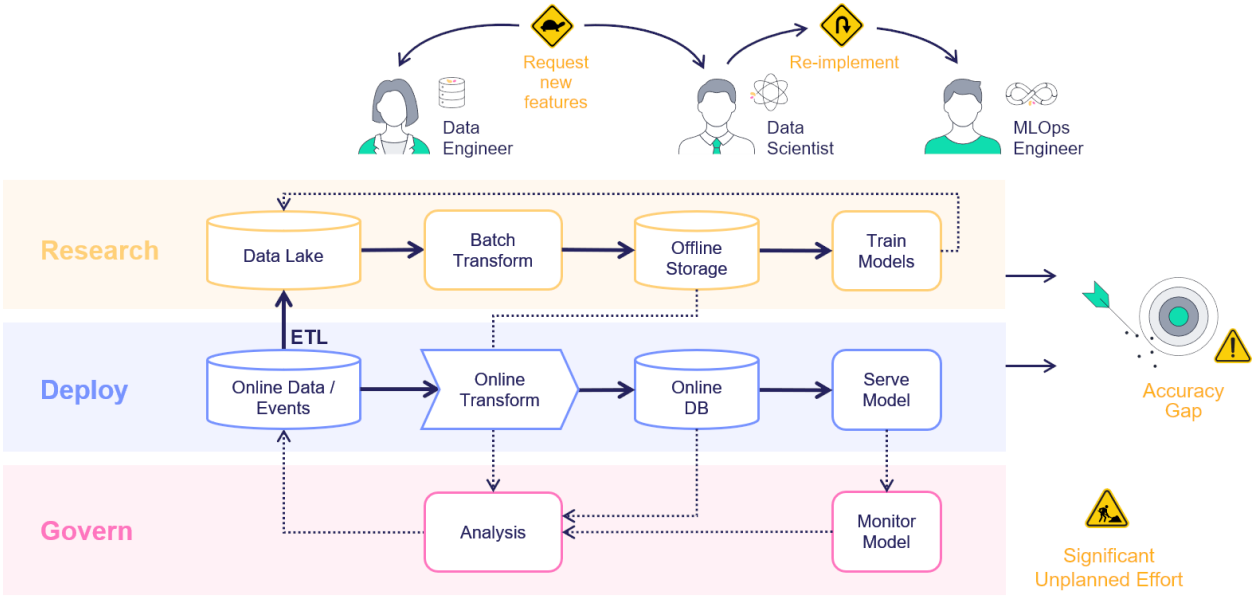
The feature store is a centralized and versioned catalog where everyone can engineer and store features along with their metadata and statistics, share them and reuse them, and analyze their impact on existing models. The feature store plugs seamlessly into the data ingestion, model training, model serving, and model monitoring components, eliminating significant development and operations overhead, and delivering exceptional performance. Users can simply collect a bunch of independent features into vectors, and use those from their jobs or real-time services. Iguazio's high performance engines take care of automatically joining and accurately computing the features.

In this section

Overview

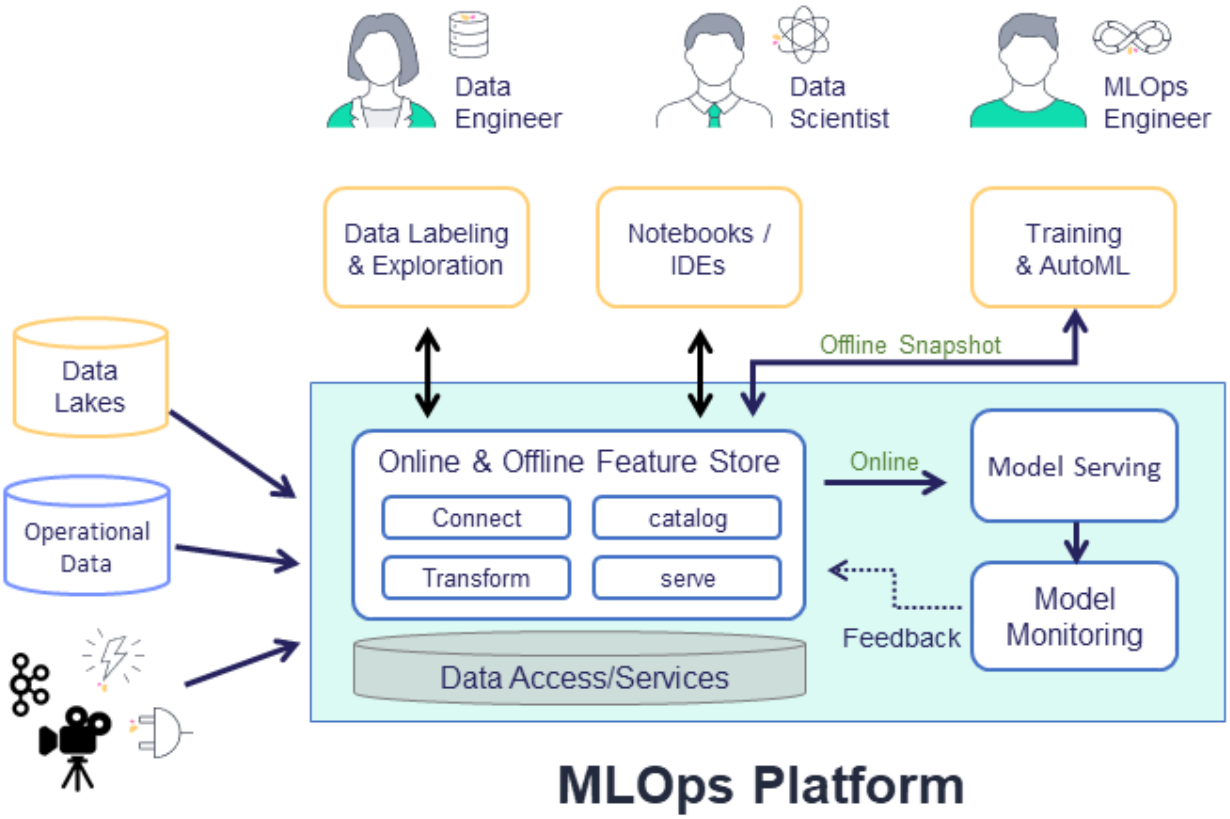
In machine-learning scenarios, generating a new feature, called feature engineering, takes a tremendous amount of work. The same features must be used both for training, based on historical data, and for the model prediction based on the online or real-time data. This creates a significant additional engineering effort, and leads to model inaccuracy when the online and offline features do not match. Furthermore, monitoring solutions must be built to track features and results and send alerts of data or model drift.

Consider, a scenario in which you train a model and one of its features is a comparison of the current amount to the average amount spent during the last 3 months by the same person. Creating such a feature is easy when you have the full dataset in training, but in serving, this feature must be calculated in an online manner. The "brute-force" way to address this is to have an ML engineer create an online pipeline that reimplements all the feature calculations done in the offline process. This is not just time-consuming and error-prone, but very difficult to maintain over time, and resulting in a lengthy deployment time. This is exacerbated when having to deal with thousands of features with an increasing number of data engineers and data scientists that are creating and using the features.



With MLRun's feature store you can easily define features in training, that are deployable to serving, without having to define all the "glue" code. You simply create the necessary building blocks to define features and integration, with

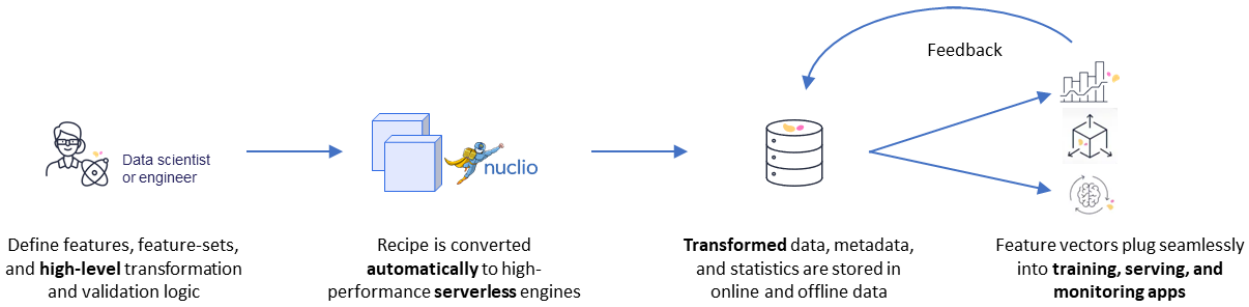
offline and online storage systems to access the features.



The feature store is comprised of the following:

- **Feature** — In machine-learning, a feature is an individual measurable property or characteristic of a phenomenon being observed. This can be raw data (e.g., transaction amount, image pixel, etc.) or a calculation derived from one or more other features (e.g., deviation from average, pattern on image, etc.).
- **Feature set** — A grouping of features. The grouping is done by setting the entity key or set-of keys. For example, a transaction can be grouped by the person ID performing the transfer or by the device identifier used to perform the transaction. It is also possible to define the timestamp source in the feature set. You can ingest data to a feature set. See more details in *feature sets*.
- **Execution graph** — A set of operations performed on the data while it is ingested. The graph contains steps that represent data sources and targets, and can also contain steps that transform and enrich the data passed through the feature set.
- **Feature vector** — A set of features, taken from one or more feature sets. The feature vector is defined prior to model training and serves as the input to the model training process. During model serving, the feature values in the vector are obtained from an online service.

How the feature store works



The common flow when working with the feature store is to first define the feature set with its source, transformation graph, and targets. MLRun's robust transformation engine performs complex operations with just a few lines of Python code. To test the execution process, call the `infer` method with a sample DataFrame. This runs all operations in memory without storing the results. Once the graph is defined, it's time to ingest the data.

You can ingest data directly from a DataFrame, by calling the feature set `ingest` method. You can also define an ingestion process that runs as a Kubernetes job. This is useful if there is a large ingestion process, or if there is a recurrent ingestion and you want to schedule the job.

MLRun can also leverage Nuclio to perform real-time ingestion by calling the `deploy_ingestion_service` function. This means that during serving, you can update feature values, and not just read them. For example, you can update a sliding window aggregation as part of a model serving process.

The next step is to define the *feature vector*. Call the `get_offline_features` function to join together features across different feature sets.

Training and serving using the feature store

Next, extract a versioned **offline** static dataset for training, based on the parquet target defined in the feature sets. You can train a model with the feature vector data by providing the input in the form of `'store://feature-vectors/{project}/{feature_vector_name}'`.

Training functions generate models and various model statistics. Use MLRun's `auto logging` capabilities to store the models along with all the relevant data, metadata and measurements.

MLRun can apply all the MLOps functionality by using the framework specific `apply_mlrun()` method, which manages the training process and automatically logs all the framework specific model details, data, metadata and metrics.

The training job automatically generates a set of results and versioned artifacts (run `train_run.outputs` to view the job outputs).

For serving, once you validate the feature vector, use the **online** feature service, based on the nosql target defined in the feature set for real-time serving. For serving, you define a serving class derived from `mlrun.serving.V2ModelServer`. In the class `load` method call the `get_online_feature_service` function with the vector name, which returns a feature service object. In the class `preprocess` method, call the feature service `get` method to get the values of those features.

Using this feature store centric process, using one computation graph definition for a feature set, you receive an automatic online and offline implementation for the feature vectors, with data versioning both in terms of the actual graph that was used to calculate each data point, and the offline datasets that were created to train each model.

Further reading

For more information, see *Feature store: data ingestion* and *Feature store: data retrieval*, as well as the *Feature Store tutorials*.

2.1.9 Runs, functions, and workflows

In this section

MLRun execution context

After running a job, you need to be able to track it. To gain the maximum value MLRun uses the job `context` object inside the code. This provides access to job metadata, parameters, inputs, secrets, and API for logging and monitoring the results, as well as log text, files, artifacts, and labels.

- If `context` is specified as the first parameter in the function signature, MLRun injects the current job context into it.
- Alternatively, if it does not run inside a function handler (e.g. in Python main or Notebook) you can obtain the `context` object from the environment using the *`get_or_create_ctx()`* function.

Common context methods:

- `get_secret(key: str)` — get the value of a secret
- `logger.info("started experiment..")` — textual logs
- `log_result(key: str, value)` — log simple values
- `set_label(key, value)` — set a label tag for that task
- `log_artifact(key, body=None, local_path=None, ...)` — log an artifact (body or local file)
- `log_dataset(key, df, ...)` — log a dataframe object
- `log_model(key, ...)` — log a model object

Example function and usage of the context object:

```
from mlrun.artifacts import ChartArtifact
import pandas as pd

def my_job(context, p1=1, p2="x"):
    # load MLRUN runtime context (will be set by the runtime framework)

    # get parameters from the runtime context (or use defaults)

    # access input metadata, values, files, and secrets (passwords)
    print(f"Run: {context.name} (uid={context.uid})")
    print(f"Params: p1={p1}, p2={p2}")
    print("accesskey = {}".format(context.get_secret("ACCESS_KEY")))
    print("file\n{}\n".format(context.get_input("infile.txt", "infile.txt").get()))

    # Run some useful code e.g. ML training, data prep, etc.

    # log scalar result values (job result metrics)
    context.log_result("accuracy", p1 * 2)
    context.log_result("loss", p1 * 3)
    context.set_label("framework", "sklearn")
```

(continues on next page)

(continued from previous page)

```
    # log various types of artifacts (file, web page, table), will be versioned and
↪visible in the UI
    context.log_artifact(
        "model",
        body=b"abc is 123",
        local_path="model.txt",
        labels={"framework": "xgboost"},
    )
    context.log_artifact(
        "html_result", body=b"<b> Some HTML <b>", local_path="result.html"
    )

    # create a chart output (will show in the pipelines UI)
    chart = ChartArtifact("chart")
    chart.labels = {"type": "roc"}
    chart.header = ["Epoch", "Accuracy", "Loss"]
    for i in range(1, 8):
        chart.add_row([i, i / 20 + 0.75, 0.30 - i / 20])
    context.log_artifact(chart)

    raw_data = {
        "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
        "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
        "age": [42, 52, 36, 24, 73],
        "testScore": [25, 94, 57, 62, 70],
    }
    df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age", "testScore
↪"])
    context.log_dataset("mydf", df=df, stats=True)
```

Example of creating the context objects from the environment:

```
if __name__ == "__main__":
    context = mlrun.get_or_create_ctx('train')
    p1 = context.get_param('p1', 1)
    p2 = context.get_param('p2', 'a-string')
    # do something
    context.log_result("accuracy", p1 * 2)
    # commit the tracking results to the DB (and mark as completed)
    context.commit(completed=True)
```

Note that MLRun context is also a python context and can be used in a `with` statement (eliminating the need for `commit`).

```
if __name__ == "__main__":
    with mlrun.get_or_create_ctx('train') as context:
        p1 = context.get_param('p1', 1)
        p2 = context.get_param('p2', 'a-string')
        # do something
        context.log_result("accuracy", p1 * 2)
```

Submitting tasks/jobs to functions

MLRun batch function objects support a *run()* method for invoking a job over them. The run method accepts various parameters such as `name`, `handler`, `params`, `inputs`, `schedule`, etc. Alternatively you can pass a **`Task`** object (see: *new_task()*) that holds all of the parameters plus the advanced options.

Run/simulate functions locally:

Functions can also run and be debugged locally by using the `local` runtime or by setting the `local=True` parameter in the *run()* method (for batch functions).

Functions can host multiple methods (handlers). You can set the default handler per function. You need to specify which handler you intend to call in the run command.

You can pass `parameters` (arguments) or data `inputs` (such as datasets, feature-vectors, models, or files) to the functions through the `run` method.

- Inside the function you can access the parameters/inputs by simply adding them as parameters to the function or you can get them from the context object (using `get_param()` and `get_input()`).
- Various data objects (files, tables, models, etc.) are passed to the function as data item objects. You can pass data objects using the inputs dictionary argument, where the dictionary keys match the function's handler argument names and the MLRun data urls are provided as the values. The data is passed into the function as a *DataItem* object that handles data movement, tracking and security in an optimal way. Read more about data objects in *Data Stores & Data Items*.

  run_results = fn.run(params={"label_column": "label"}, inputs={'data': data_url})

MLRun also supports iterative jobs that can run and track multiple child jobs (for hyperparameter tasks, AutoML, etc.). See *Hyperparam and Iterative jobs* for details and examples.

The `run()` command returns a run object that you can use to track the job and its results. If you pass the parameter `watch=True` (default) the *run()* command blocks until the job completes.

Run object has the following methods/properties:

- `uid()` — returns the unique ID.
- `state()` — returns the last known state.
- `show()` — shows the latest job state and data in a visual widget (with hyperlinks and hints).
- `outputs` — returns a dictionary of the run results and artifact paths.
- `logs(watch=True)` — returns the latest logs. Use `Watch=False` to disable the interactive mode in running jobs.
- `artifact(key)` — returns an artifact for the provided key (as *DataItem* object).
- `output(key)` — returns a specific result or an artifact path for the provided key.
- `wait_for_completion()` — wait for async run to complete
- `refresh()` — refresh run state from the db/service
- `to_dict()`, `to_yaml()`, `to_json()` — converts the run object to a dictionary, YAML, or JSON format (respectively).

You can view the job details, logs, and artifacts in the UI. When you first open the **Monitor Jobs** tab it displays the last jobs that ran and their data. Click a job name to view its run history, and click a run to view more of the run's data.

See full details and examples in *Creating and using functions*.

Multi-stage workflows

A workflow is a definition of execution of functions. It defines the order of execution of multiple dependent steps in a DAG. A workflow can reference the project's params, secrets, artifacts, etc. It can also use a function execution output as a function execution input (which, of course, defines the order of execution).

MLRun supports running workflows on a `local` or `kubeflow` pipeline engine. The `local` engine runs the workflow as a local process, which is simpler for debuggimg and running simple/sequential tasks. The `kubeflow` ("kfp") engine runs as a task over the cluster and supports more advanced operations (conditions, branches, etc.). You can select the engine at runtime. Kubeflow-specific directives like conditions and branches are not supported by the `local` engine.

Workflows are saved/registered in the project using the *`set_workflow()`*.Workflows are executed using the *`run()`* method or using the CLI command `mlrun project`.

See full details in *Project Workflows and Automation*.

Automated Logging and MLOps with `apply_mlrun()`

You can write custom training functions or use built-in marketplace functions for training models using common open-source frameworks and/or cloud services (such as AzureML, Sagemaker, etc.).

Inside the ML function you can use the `apply_mlrun()` method, which automates the tracking and MLOps functionality.

With `apply_mlrun()` the following outputs are generated automatically:

- Plots — loss convergence, ROC, confusion matrix, feature importance, etc.
- Metrics — accuracy, loss, etc.
- Dataset Artifacts — like the dataset used for training and / or testing
- Custom code — like custom layers, metrics, and so on
- Model Artifact — enables versioning, monitoring and automated deployment

In addition it handles automation of various MLOps tasks like scaling runs over multiple containers (with Dask, Horovod, and Spark), run profiling, hyperparameter tuning, ML Pipeline, and CI/CD integration, etc.

`apply_mlrun()` accepts the model object and various optional parameters. For example:

```
apply_mlrun(model=model, model_name="my_model",
            x_test=x_test, y_test=y_test)
```

When specifying the `x_test` and `y_test` data it generates various plots and calculations to evaluate the model. Metadata and parameters are automatically recorded (from the MLRun `context` object) and don't need to be specified.

`apply_mlrun` is framework specific and can be imported from MLRun's **frameworks** package — a collection of commonly used machine and deep learning frameworks fully supported by MLRun.

`apply_mlrun` can be used with its default settings, but it is highly flexible and rich with different options and configurations. Reading the docs of your favorite framework to get the most out of MLRun:

- SciKit-Learn
- TensorFlow (and Keras)

- PyTorch
- XGBoost
- LightGBM
- ONNX

2.1.10 Artifacts and models

An artifact is any data that is produced and/or consumed by functions, jobs, or pipelines.

Artifacts metadata is stored in the projects database. The main types of artifacts are:

- **Files** — files, directories, images, figures, and plotlines.
- **Datasets** — any data , such as tables and DataFrames.
- **Models** — all trained models.
- **Feature Store Objects** - Feature Sets and Feature Vectors

IN this section

Artifacts and versioning

Artifacts can be viewed and managed in the UI. In the project page, select the artifact type `models`, `files`, or `feature-store` (for datasets and feature store objects).

Example dataset artifact screen:

You can search the artifacts based on time and labels. For each artifact, you can view its location, the artifact type, labels, the producer of the artifact, the artifact owner, last update date, and type-specific information.

Artifacts can also be viewed from the job page (in the `artifacts` tab). For each artifact, you can view its content as well as download the artifact.

Artifact Path

Jobs use the default or job specific `artifact_path` parameter to determine where the artifacts are stored. The default `artifact_path` can be specified at the cluster level, client level, project level, or job level (at that precedence order), or can be specified as a parameter in the specific `log` operation.

You can set the default `artifact_path` for your environment using the *`set_environment()`* function.

You can override the default `artifact_path` configuration by setting the `artifact_path` parameter of the *`set_environment()`* function. You can use variables in the artifacts path, such as `{{project}}` for the name of the running project or `{{run.uid}}` for the current job/pipeline run UID. (The default artifacts path uses `{{project}}`.) The following example configures the artifacts path to an artifacts directory in the current active directory (`./artifacts`)

```
set_environment(project=project_name, artifact_path='./artifacts')
```

For Iguazio MLOps Platform users

In the platform, the default artifacts path is a /artifacts directory in the predefined "projects" data container: `/v3io/projects/<project name>/artifacts` (for example, `/v3io/projects/myproject/artifacts` for a "myproject" project).

Saving the Artifacts in Run-Specific Paths When you specify `{{run.uid}}`, the artifacts for each job are stored in a dedicated directory for each executed job. Under the artifact path you should see the source-data file in a new directory whose name is derived from the unique run ID. Otherwise, the same artifacts directory is used in all runs, and the artifacts for newer runs override those from the previous runs.

As previously explained, `set_environment` returns a tuple with the project name and artifacts path. You can optionally save your environment's artifacts path to a variable, as demonstrated in the previous steps. You can then use the artifacts-path variable to extract paths to task-specific artifact subdirectories. For example, the following code extracts the path to the artifacts directory of a `training` task, and saves the path to a `training_artifacts` variable:

```
from os import path
training_artifacts = path.join(artifact_path, 'training')
```

Note

The artifacts path uses *data store URLs* which are not necessarily local file paths (for example, `s3://bucket/path`). Be careful not to use such paths with general file utilities.

Artifact URIs, Metadata and Versioning

Artifacts have unique URIs in the form `store://<type>/<project>/<key/path>[:tag]`. The URI is automatically generated by `log_artifact` and can be used as input to jobs, functions, pipelines, etc..

Artifacts are versioned. Each unique version has a unique IDs (uid) and can have a `tag` label.When the tag is not specified, it uses the `latest` version.

Artifact metadata and objects can be accessed through the SDK or downloaded from the UI (as YAML files). They host common and object specific metadata such as:

- Common metadata: name, project, updated, version info

- How they were produced (user, job, pipeline, etc.)
- Lineage data (sources used to produce that artifact)
- Information about formats, schema, sample data
- Links to other artifacts (e.g. a model can point to a chart)
- Type-specific attributes

Artifacts can be obtained via the SDK through type specific APIs or using generic artifact APIs such as:

- *get_dataitem()* - get the *DataItem* object for reading/downloading the artifact content
- *get_store_resource()* - get the artifact object

Example artifact URLs:

```
store://artifacts/default/my-table
store://artifacts/sk-project/train-model:e95f757e-7959-4d66-b500-9f6cdb1f0bc7
store://feature-sets/stocks/quotes:v2
store://feature-vectors/stocks/enriched-ticker
```

File and dir artifacts

Dataset artifacts

Storing datasets is important in order to have a record of the data that was used to train the model, as well as storing any processed data. MLRun comes with built-in support for the DataFrame format. MLRun not only stores the DataFrame, but it also provides information about the data, such as statistics.

The simplest way to store a dataset is with the following code:

```
context.log_dataset(key='my_data', df=df)
```

Where `key` is the the name of the artifact and `df` is the DataFrame. By default, MLRun stores a short preview of 20 lines. You can change the number of lines by changing the value of the `preview` parameter.

MLRun also calculates statistics on the DataFrame on all numeric fields. You can enable statistics regardless to the DataFrame size by setting the `stats` parameter to `True`.

Logging a Dataset From a Job

The following example shows how to work with datasets from a job:

```
from os import path
from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem

# Ingest a data set into the platform
def get_data(context: MLClientCtx, source_url: DataItem, format: str = 'csv'):

    iris_dataset = source_url.as_df()

    target_path = path.join(context.artifact_path, 'data')
    # Optionally print data to your logger
    context.logger.info('Saving Iris data set to {} ...'.format(target_path))

```

(continues on next page)

(continued from previous page)

```
    # Store the data set in your artifacts database
    context.log_dataset('iris_dataset', df=iris_dataset, format=format,
                        index=False, artifact_path=target_path)
```

You can run this function locally or as a job. For example, to run it locally:

```
from os import path
from mlrun import new_project, run_local, mlconf

project_name = 'my-project'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

source_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'
# Run get-data function locally
get_data_run = run_local(name='get_data',
                         handler=get_data,
                         inputs={'source_url': source_url},
                         project=project_name,
                         artifact_path=artifact_path)
```

The dataset location is returned in the `outputs` field, therefore you can get the location by calling `get_data_run.artifact('iris_dataset')` to get the dataset itself.

```
# Read your data set
get_data_run.artifact('iris_dataset').as_df()

# Visualize an artifact in Jupyter (image, html, df, ..)
get_data_run.artifact('confusion-matrix').show()
```

Call `dataset.meta.stats` to obtain the data statistics. You can also get the data as a Pandas Dataframe by calling the `dataset.as_df()`.

Model artifacts

An essential piece of artifact management and versioning is storing a model version. This allows users to experiment with different models and compare their performance, without having to worry about losing their previous results.

The simplest way to store a model named `my_model` is with the following code:

```
from pickle import dumps
model_data = dumps(model)
context.log_model(key='my_model', body=model_data, model_file='my_model.pkl')
```

You can also store any related metrics by providing a dictionary in the `metrics` parameter, such as `metrics={'accuracy': 0.9}`. Furthermore, any additional data that you would like to store along with the model can be specified in the `extra_data` parameter. For example `extra_data={'confusion': confusion.target_path}`

A convenient utility method, `eval_model_v2`, which calculates mode metrics is available in `mlrun.utils`.

See example below for a simple model trained using scikit-learn (normally, you would send the data as input to the function). The last 2 lines evaluate the model and log the model.

```
from sklearn import linear_model
from sklearn import datasets
from sklearn.model_selection import train_test_split
from pickle import dumps

from mlrun.execution import MLClientCtx
from mlrun.mlutils import eval_model_v2

def train_iris(context: MLClientCtx):

    # Basic scikit-learn iris SVM model
    X, y = datasets.load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)
    model = linear_model.LogisticRegression(max_iter=10000)
    model.fit(X_train, y_train)

    # Evaluate model results and get the evaluation metrics
    eval_metrics = eval_model_v2(context, X_test, y_test, model)

    # Log model
    context.log_model("model",
                      body=dumps(model),
                      artifact_path=context.artifact_subpath("models"),
                      extra_data=eval_metrics,
                      model_file="model.pkl",
                      metrics=context.results,
                      labels={"class": "sklearn.linear_model.LogisticRegression"})
```

Save the code above to `train_iris.py`. The following code loads the function and runs it as a job. See the quick-start page to learn how to create the project and set the artifact path.

```
from mlrun import code_to_function

gen_func = code_to_function(name='train_iris',
                            filename='train_iris.py',
                            handler='train_iris',
                            kind='job',
                            image='mlrun/ml-models')

train_iris_func = project.set_function(gen_func).apply(auto_mount())

train_iris = train_iris_func.run(name='train_iris',
                                 handler='train_iris',
                                 artifact_path=artifact_path)
```

You can now use `get_model` to read the model and run it. This function will get the model file, metadata, and extra data. The input can be either the path of the model, or the directory where the model resides. If you provide a directory, the function will search for the model file (by default it searches for .pkl files)

The following example gets the model from `models_path` and test data in `test_set` with the expected label provided as a column of the test data. The name of the column containing the expected label is provided in `label_column`. The example then retrieves the models, runs the model with the test data and updates the model with the metrics and results of the test data.

```
from pickle import load

from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from mlrun.artifacts import get_model, update_model
from mlrun.mlutils import eval_model_v2

def test_model(context: MLClientCtx,
               models_path: DataItem,
               test_set: DataItem,
               label_column: str):

    if models_path is None:
        models_path = context.artifact_subpath("models")
    xtest = test_set.as_df()
    ytest = xtest.pop(label_column)

    model_file, model_obj, _ = get_model(models_path)
    model = load(open(model_file, 'rb'))

    extra_data = eval_model_v2(context, xtest, ytest.values, model)
    update_model(model_artifact=model_obj, extra_data=extra_data,
                 metrics=context.results, key_prefix='validation-')
```

To run the code, place the code above in `test_model.py` and use the following snippet. The model from the previous step is provided as the `models_path`:

```
from mlrun.platforms import auto_mount
gen_func = code_to_function(name='test_model',
                            filename='test_model.py',
                            handler='test_model',
                            kind='job',
                            image='mlrun/ml-models')

func = project.set_function(gen_func).apply(auto_mount())

run = func.run(name='test_model',
               handler='test_model',
               params={'label_column': 'label'},
               inputs={'models_path': train_iris.outputs['model'],
                       'test_set': 'https://s3.wasabisys.com/iguazio/data/iris/iris_
↪dataset.csv'}),
               artifact_path=artifact_path)
```

Plot artifacts

Storing plots is useful to visualize the data and to show any information regarding the model performance. For example, you can store scatter plots, histograms and cross-correlation of the data, and for the model store the ROC curve and confusion matrix.

The following code creates a confusion matrix plot using sklearn.metrics.plot_confusion_matrix and stores the plot in the artifact repository:

```
from mlrun.artifacts import PlotArtifact
from mlrun.mlutils import gcf_clear

```

(continues on next page)

(continued from previous page)

```
gcf_clear(plt)
confusion_matrix = metrics.plot_confusion_matrix(model,
                                                 xtest,
                                                 ytest,
                                                 normalize='all',
                                                 values_format = '.2g',
                                                 cmap=plt.cm.Blues)
confusion_matrix = context.log_artifact(PlotArtifact('confusion-matrix',␣
↪body=confusion_matrix.figure_),
                                        local_path='plots/confusion_matrix.html')
```

You can use the `update_dataset_meta` function to associate the plot with the dataset by assigning the value of the `extra_data` parameter:

```
from mlrun.artifacts import update_dataset_meta

extra_data = {'confusion_matrix': confusion_matrix}
update_dataset_meta(dataset, extra_data=extra_data)
```

Back to top

2.1.11 Deployment and monitoring

In this section

Real-time pipelines

MLRun graphs enable building and running DAGs (directed acyclic graphs). Graphs are composed of individual steps. The first graph element accepts an Event object, transforms/processes the event and passes the result to the next step in the graph. The final result can be written out to some destination (file, DB, stream, etc.) or returned back to the caller (one of the graph steps can be marked with `.respond()`).

MLRun graph capabilities include:

- Easy to build and deploy distributed real-time computation graphs
- Use the real-time serverless engine (Nuclio) for auto-scaling and optimized resource utilization
- Built-in operators to handle data manipulation, IO, machine learning, deep-learning, NLP, etc.
- Built-in monitoring for performance, resources, errors, data, model behaviour, and custom metrics
- Debug in the IDE/Notebook

The serving graphs are used by *MLRun's Feature Store* to build real-time feature engineering pipelines.

See full details and examples in *Real-time serving pipelines (graphs)*.

Model serving

MLRun Serving allow composition of multi-stage real-time pipelines made of serverless Nuclio functions, including data processing, advanced model serving, custom logic, and fast access to a variety of data systems, and deploying them quickly to production with minimal effort.

High-level transformation logic is automatically converted to real-time serverless processing engines that can accept events or online data, handle any type of structured or unstructured data, and run complex computation graphs and native user code.

Graphs are used to deploy and serve ML/DL models. Graphs can be deployed into a production serverless pipeline with a single command.

See full details and examples in *model serving using the graphs*.

Model monitoring

Model performance monitoring is a basic operational task that is implemented after an AI model has been deployed. MLRun:

- Monitors your models in production, and identifies and mitigates drift on the fly.
- Detects model drift based on feature drift via the integrated feature store, and auto-triggers retraining.

See full details and examples in *Model monitoring*.

CI/CD and automation

You can run your ML Pipelines using CI frameworks like Github Actions, GitLab CI/CD, etc. MLRun supports a simple and native integration with the CI systems.

- Build/run complex workflows composed of local/library functions or external cloud services (e.g. AutoML)
- Support various Pipeline/CI engines (Kubeflow, GitHub, Gitlab, Jenkins)
- Track & version code, data, params, results with minimal effort
- Elastic scaling of each step
- Extensive function marketplace

See full details and examples in *Github/Gitlab and CI/CD Integration*.

2.1.12 Feature Store: Data ingestion

Learn how to ingest data into the feature store, with transformations, using the supported engines.

In this section

Feature sets

In MLRun, a group of features can be ingested together and stored in logical group called feature set. Feature sets take data from offline or online sources, build a list of features through a set of transformations, and store the resulting features along with the associated metadata and statistics. A feature set can be viewed as a database table with multiple material implementations for batch and real-time access, along with the data pipeline definitions used to produce the features.

The feature set object contains the following information:

- **Metadata**—General information which is helpful for search and organization. Examples are project, name, owner, last update, description, labels, etc.
- **Key attributes**—Entity (the join key), timestamp key (optional), label column.
- **Features**—The list of features along with their schema, metadata, validation policies and statistics.
- **Source**—The online or offline data source definitions and ingestion policy (file, database, stream, http endpoint, etc.).
- **Transformation**—The data transformation pipeline (e.g. aggregation, enrichment etc.).
- **Target stores**—The type (i.e. parquet/csv or key value), location and status for the feature set materialized data.
- **Function**—The type (storey, pandas, spark) and attributes of the data pipeline serverless functions.

In this section

Create a Feature Set

Create a new `FeatureSet` with the base definitions:

- **name**—The feature set name is a unique name within a project.
- **entities**—Each feature set must be associated with one or more index column. When joining feature sets, the entity is used as the key column.
- **timestamp_key**—(optional) Used for specifying the time field when joining by time.
- **engine**—The processing engine type:
  - Spark
  - pandas
  - storey (some advanced functionalities are in the Beta state)

Example:

```
#Create a basic feature set example
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
```

To learn more about Feature Sets go to *FeatureSet*.

Note

Feature sets can also be created in the UI. To create a feature set:

1. Select a project and press **Feature store**, then press **Create Set**.
2. After completing the form, press **Save and Ingest** to start the process, or **Save** to save the set for later ingestion.

Add transformations

Define the data processing steps using a transformations graph (DAG).

A feature set data pipeline takes raw data from online or offline sources and transforms it to meaningful features. The MLRun feature store supports three processing engines (storey, pandas, spark) that can run in the client (e.g. Notebook) for interactive development or in elastic serverless functions for production and scale.

The data pipeline is defined using MLRun graph (DAG) language. Graph steps can be pre-defined operators (such as aggregate, filter, encode, map, join, impute, etc) or custom python classes/functions. Read more about the graph in *Real-time serving pipelines (graphs)*.

The `pandas` and `spark` engines are good for simple batch transformations, while the `storey` stream processing engine (the default engine) can handle complex workflows and real-time sources.

The results from the transformation pipeline are stored in one or more material targets. Data for offline access, such as training, is usually stored in Parquet files. Data for online access such as serving is stored in a NoSQL DB. You can use the default targets or add/replace with additional custom targets.

Graph example (storey engine):

```
import mlrun.feature_store as fstore
feature_set = fstore.FeatureSet("measurements", entities=[Entity(key)], timestamp_key=
↪"timestamp")
# Define the computational graph including the custom functions
feature_set.graph.to(DropColumns(drop_columns))\
                 .to(RenameColumns(mapping={'bad': 'bed'}))
feature_set.add_aggregation('hr', ['avg'], ["1h"])
feature_set.plot()
fstore.ingest(feature_set, data_df)
```

Graph example (pandas engine):

```
def myfunc1(df, context=None):
    df = df.drop(columns=["exchange"])
    return df

stocks_set = fstore.FeatureSet("stocks", entities=[Entity("ticker")], engine="pandas")
stocks_set.graph.to(name="s1", handler="myfunc1")
df = fstore.ingest(stocks_set, stocks_df)
```

The graph steps can use built-in transformation classes, simple python classes, or function handlers.

See more details in *Feature set transformations*.

Simulate and debug the data pipeline with a small dataset

During the development phase it's pretty common to check the feature set definition and to simulate the creation of the feature set before ingesting the entire dataset, since ingesting the entire feature set can take time. This allows you to get a preview of the results (in the returned dataframe). The simulation method is called `infer`. It infers the source data schema as well as processing the graph logic (assuming there is one) on a small subset of data. The infer operation also learns the feature set schema and does statistical analysis on the result by default.

```
df = fstore.preview(quotes_set, quotes)

# print the featue statistics
print(quotes_set.get_stats_table())
```

Ingest data into the Feature Store

Define the source and material targets, and start the ingestion process (as *local process*, *remote job*, or *Real-time ingestion*).

Data can be ingested as a batch process either by running the ingest command on demand or as a scheduled job. Batch ingestion can be done locally (i.e. running as a python process in the Jupyter pod) or as an MLRun job.

The data source can be a DataFrame or files (e.g. csv, parquet). Files can be either local files residing on a volume (e.g. v3io), or remote (e.g. S3, Azure blob). MLRun also supports Google BigQuery as a data source. If you define a transformation graph, then the ingestion process runs the graph transformations, infers metadata and stats, and writes the results to a target data store.

When targets are not specified, data is stored in the configured default targets (i.e. NoSQL for real-time and Parquet for offline).

Limitations

- Do not name columns starting with either `t_` or `aggr_`. They are reserved for internal use, and the data does not ingest correctly. See also general limitations in Attribute name restrictions.
- When using the pandas engine, do not use spaces ( ) or periods (.) in the column names. These cause errors in the ingestion.

Ingest data locally

Use a Feature Set to create the basic feature-set definition and then an ingest method to run a simple ingestion "locally" in the Jupyter Notebook pod.

```
# Simple feature set that reads a csv file as a dataframe and ingests it as is
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
stocks = pd.read_csv("stocks.csv")
df = ingest(stocks_set, stocks)

# Specify a csv file as source, specify a custom CSV target
source = CSVSource("mycsv", path="stocks.csv")
targets = [CSVTarget("mycsv", path="./new_stocks.csv")]
ingest(measurements, source, targets)
```

To learn more about ingest go to *ingest*.

Ingest data using an MLRun job

Use the ingest method with the `run_config` parameter for running the ingestion process using a serverless MLRun job. By doing that, the ingestion process runs on its own pod or service on the kubernetes cluster. This option is more robust since it can leverage the cluster resources, as opposed to running within the Jupyter Notebook. It also enables you to schedule the job or use bigger/faster resources.

```
# Running as remote job
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
config = RunConfig(image='mlrun/mlrun')
df = ingest(stocks_set, stocks, run_config=config)
```

Real-time ingestion

Real time use cases (e.g. real time fraud detection) require feature engineering on live data (e.g. z-score calculation) while the data is coming from a streaming engine (e.g. kafka) or a live http endpoint. The feature store enables you to start real-time ingestion service. When running the *deploy_ingestion_service* the feature store creates an elastic real time serverless function (the nuclio function) that runs the pipeline and stores the data results in the "offline" and "online" feature store by default. There are multiple data source options including http, kafka, kinesis, v3io stream, etc. Due to the asynchronous nature of feature store's execution engine, errors are not returned, but rather logged and pushed to the defined error stream.

```
# Create a real time function that receives http requests
# the "ingest" function runs the feature engineering logic on live events
source = HTTPSource()
func = mlrun.code_to_function("ingest", kind="serving").apply(mount_v3io())
config = RunConfig(function=func)
fstore.deploy_ingestion_service(my_set, source, run_config=config)
```

To learn more about deploy_ingestion_service go to *deploy_ingestion_service*.

Incremental ingestion

You can schedule an ingestion job for a feature set on an ongoing basis. The first scheduled job runs on all the data in the source and the subsequent jobs ingest only the deltas since the previous run (from the last timestamp of the previous run until `datetime.now`). Example:

```
cron_trigger = "* */1 * * *" #will run every hour source =
ParquetSource("myparquet", path=path, time_field="time", schedule=cron_trigger)
feature_set = fs.FeatureSet(name=name, entities=[fs.Entity("first_name")],
timestamp_key="time",) fs.ingest(feature_set, source, run_config=fs.
RunConfig())
```

The default value for the `overwrite` parameter in the ingest function for scheduled ingest is `False`, meaning that the target from the previous ingest is not deleted. For the storey engine, the feature is currently implemented for ParquetSource only. (CsvSource will be supported in a future release). For Spark engine, other sources are also supported.

Data sources

For batch ingestion the feature store supports dataframes and files (i.e. csv & parquet). The files can reside on S3, NFS, Azure blob storage, or the Iguazio platform. MLRun also supports Google BigQuery as a data source. When working with S3/Azure, there are additional requirements. Use pip install mlrun[s3] or pip install mlrun[azure-blob-storage] to install them.

- Azure: define the environment variable `AZURE_STORAGE_CONNECTION_STRING`.
- S3: define `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` and `AWS_BUCKET`.

For real time ingestion the source can be http, kafka or v3io stream, etc. When defining a source, it maps to nuclio event triggers.

Note that you can also create a custom `source` to access various databases or data sources.

Target stores

By default the feature sets are stored as both parquet file for training and as a key value table (in the Iguazio MLOps platform) for online serving. The parquet file is ideal for fetching large set of data for training while the key value is ideal for an online application since it supports low latency data retrieval based on key access.

Note

When working with the Iguazio MLOps platform the default feature set storage location is under the "Projects" container: /fs/.. folder. The default location can be modified in mlrun config or specified per ingest operation. The parquet/csv files can be stored in NFS, S3, Azure blob storage and on Iguazio DB/FS.

Feature set transformations

A feature set contains an execution graph of operations that are performed when data is ingested, or when simulating data flow for inferring its metadata. This graph utilizes MLRun's *serving graph*.

The graph contains steps that represent data sources and targets, and may also contain steps whose purpose is transformations and enrichment of the data passed through the feature set. These transformations can be provided in one of three ways:

- ***Aggregations*** — MLRun supports adding aggregate features to a feature set through the `add_aggregation()` function.
- ***Built-in transformations*** — MLRun is equipped with a set of transformations provided through the `storey.transformations` package. These transformations can be added to the execution graph to perform common operations and transformations.
- ***Custom transformations*** — You can extend the built-in functionality by adding new classes that perform any custom operation and use them in the serving graph.

Once a feature-set is created, its internal execution graph can be observed by calling the feature-set's `plot()` function, which generates a `graphviz` plot based on the internal graph. This is very useful when running within a Jupyter notebook, and produces a graph such as the following example:

This plot shows various transformations and aggregations being used as part of the feature-set processing, as well as the targets where results are saved to (in this case two targets). Feature-sets can also be observed in the MLRun UI, where the full graph can be seen and specific step properties can be observed:

For a full end-to-end example of feature-store and usage of the functionality described in this page, refer to the *feature store example*.

In this section

Aggregations

Aggregations, being a common tool in data preparation and ML feature engineering, are available directly through the MLRun `FeatureSet` class. These transformations allow adding a new feature to the feature-set that is created by performing some aggregate function over feature's values within a time-based sliding window.

For example, if a feature-set contains stock trading data including the specific bid price for each bid at any given time, you could introduce aggregate features that show the minimal and maximal bidding price over all the bids in the last hour, per stock ticker (which is the entity in question). To do that, use the code:

```
import mlrun.feature_store as fstore
# create a new feature set
quotes_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")])
quotes_set.add_aggregation("bid", ["min", "max"], ["1h"], "10m")
```

Once this is executed, the feature-set has new features introduced, with their names produced from the aggregate parameters, using this format: `{column}_{operation}_{window}`. Thus, the example above generates two new features: `bid_min_1h` and `bid_max_1h`. If the function gets an optional `name` parameter, features are produced in `{name}_{operation}_{window}` format. If the `name` parameter is not specified, features are produced in `{column_name}_{operation}_{window}` format. These features can then be fed into predictive models or be used for additional processing and feature generation.

Note

Internally, the graph step that is created to perform these aggregations is named `"Aggregates"`. If more than one aggregation steps are needed, a unique name must be provided to each, using the `state_name` parameter.

Aggregations that are supported using this function are:

- `count`
- `sum`
- `sqr` (sum of squares)
- `max`
- `min`
- `first`
- `last`
- `avg`
- `stdvar`
- `stddev`

For a full documentation of this function, see the *add_aggregation()* documentation.

Built-in transformations

MLRun, and the associated `storey` package, have a built-in library of transformation functions that can be applied as steps in the feature-set's internal execution graph. In order to add steps to the graph, it should be referenced from the *FeatureSet* object by using the *graph* property. Then, new steps can be added to the graph using the functions in *storey.transformations* (follow the link to browse the documentation and the list of existing functions). The transformations are also accessible directly from the `storey` module.

Note

Internally, MLRun makes use of functions defined in the `storey` package for various purposes. When creating a feature-set and configuring it with sources and targets, what MLRun does behind the scenes is to add steps to the execution graph that wraps methods and classes, which perform the actions. When defining an async execution graph, `storey` classes are used. For example, when defining a Parquet data-target in MLRun, a graph step is created that wraps storey's `WriteToParquet()` function.

To use a function:

1. Access the graph from the feature-set object, using the *graph* property.
2. Add steps to the graph using the various graph functions, such as `to()`. The function object passed to the step should point at the transformation function being used.

The following is an example for adding a simple `filter` to the graph, that drops any bid which is lower than 50USD:

```
quotes_set.graph.to("storey.Filter", "filter", _fn="(event['bid'] > 50)")
```

In the example above, the parameter `_fn` denotes a callable expression that is passed to the `storey.Filter` class as the parameter `fn`. The callable parameter can also be a Python function, in which case there's no need for parentheses around it. This call generates a step in the graph called `filter` that calls the expression provided with the event being propagated through the graph as data is fed to the feature-set.

Custom transformations

When a transformation is needed that is not provided by the built-in functions, new classes that implement transformations can be created and added to the execution graph. Such classes should extend the `MapClass` class, and the actual transformation should be implemented within their `do()` function, which receives an event and returns the event after performing transformations and manipulations on it. For example, consider the following code:

```
class MyMap(MapClass):
    def __init__(self, multiplier=1, **kwargs):
        super().__init__(**kwargs)
        self._multiplier = multiplier

    def do(self, event):
        event["multi"] = event["bid"] * self._multiplier
        return event
```

The `MyMap` class can then be used to construct graph steps, in the same way as shown above for built-in functions:

```
quotes_set.graph.add_step("MyMap", "multi", after="filter", multiplier=3)
```

This uses the `add_step` function of the graph to add a step called `multi` utilizing `MyMap` after the `filter` step that was added previously. The class is initialized with a multiplier of 3.

Using the Spark execution engine

The feature store supports using Spark for ingesting, transforming and writing results to data targets. When using Spark, the internal execution graph is executed synchronously, by utilizing a Spark session to perform read and write operations, as well as potential transformations on the data. Executing synchronously means that the source data is fully read into a data-frame that is processed, writing the output to the targets defined.

Spark execution can be done locally, utilizing a local Spark session provided to the ingestion call. To use Spark as the transformation engine in ingestion, follow these steps:

1. When constructing the *`FeatureSet`* object, pass an `engine` parameter and set it to `spark`. For example:

```
feature_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")],␣
↪engine="spark")
```

2. To use a local Spark session, pass a Spark session context when calling the *`ingest()`* function, as the `spark_context` parameter. This session is used for data operations and transformations.
3. To use a remote execution engine (remote spark or spark operator), pass a `RunConfig` object as the `run_config` parameter for the `ingest` API. The actual remote function to execute depends on the object passed:
   - A default `RunConfig`, in which case the ingestion code either generates a new MLRun function runtime of type `remote-spark`, or utilizes the function specified in `feature_set.spec.function` (in which case, it has to be of runtime type `remote-spark` or `spark`).
   - A `RunConfig` that has a function configured within it. As mentioned, the function runtime must be of type `remote-spark` or `spark`.

See full examples in:

- *Local Spark ingestion example*
- *Remote Spark ingestion example*
- *Spark operator ingestion example*
- *Spark execution engine over S3 - full flow example*

Local Spark ingestion example

The following code executes data ingestion using a local Spark session. When using a local Spark session, the `ingest` API would wait for its completion.

```
import mlrun
from mlrun.datastore.sources import CSVSource
import mlrun.feature_store as fstore
from pyspark.sql import SparkSession

mlrun.set_environment(project="stocks")
feature_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")], engine=
↪"spark")

# add_aggregation can be used in conjunction with Spark
feature_set.add_aggregation("price", ["min", "max"], ["1h"], "10m")
```

(continues on next page)

(continued from previous page)

```
source = CSVSource("mycsv", path="v3io:///projects/stocks.csv")

# Execution using a local Spark session
spark = SparkSession.builder.appName("Spark function").getOrCreate()
fstore.ingest(feature_set, source, spark_context=spark)
```

Remote Spark ingestion example

When using remote execution the MLRun run execution details are returned, allowing tracking of its status and results.

The following code should be executed only once to build the remote spark image before running the first ingest. It may take a few minutes to prepare the image.

```
from mlrun.runtimes import RemoteSparkRuntime
RemoteSparkRuntime.deploy_default_image()
```

Remote ingestion:

```
# mlrun: start-code
```

```
from mlrun.feature_store.api import ingest
def ingest_handler(context):
    ingest(mlrun_context=context) # The handler function must call ingest with the␣
↪mlrun_context
```

You can run your PySpark code for ingesting data into the feature store by adding:

```
def my_spark_func(df, context=None):
    return df.filter("bid>55") # PySpark code
```

```
# mlrun: end-code
```

```
from mlrun.datastore.sources import CSVSource
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

spark_service_name = "iguazio-spark-service" # As configured & shown in the Iguazio␣
↪dashboard

feature_set.graph.to(name="s1", handler="my_spark_func")
my_func = code_to_function("func", kind="remote-spark")
config = fstore.RunConfig(local=False, function=my_func, handler="ingest_handler")
fstore.ingest(feature_set, source, run_config=config, spark_context=spark_service_
↪name)
```

Spark operator ingestion example

When running with a Spark operator, the MLRun execution details are returned, allowing tracking of the job's status and results.

The following code should be executed only once to build the spark job image before running the first ingest. It may take a few minutes to prepare the image.

```
from mlrun.runtimes import Spark3Runtime
Spark3Runtime.deploy_default_image()
```

Spark operator ingestion:

```
# mlrun: start-code

from mlrun.feature_store.api import ingest

def ingest_handler(context):
    ingest(mlrun_context=context) # The handler function must call ingest with the␣
↪mlrun_context

# You can add your own PySpark code as a graph step:
def my_spark_func(df, context=None):
    return df.filter("bid>55") # PySpark code

# mlrun: end-code
```

```
from mlrun.datastore.sources import CSVSource
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

feature_set.graph.to(name="s1", handler="my_spark_func")

my_func = code_to_function("func", kind="spark")

my_func.with_driver_requests(cpu="200m", mem="1G")
my_func.with_executor_requests(cpu="200m", mem="1G")
my_func.with_igz_spark()

# Enables using the default image (can be replace with specifying a specific image␣
↪with .spec.image)
my_func.spec.use_default_image = True

# Not a must - default: 1
my_func.spec.replicas = 2

# If needed, sparkConf can be modified like this:
# my_func.spec.spark_conf['spark.specific.config.key'] = 'value'

config = fstore.RunConfig(local=False, function=my_func, handler="ingest_handler")
fstore.ingest(feature_set, source, run_config=config)
```

Spark execution engine over S3 - full flow example

For Spark to work with S3, it requires several properties to be set. The following example writes a feature set to S3 in the parquet format in a remote k8s job:

One-time setup:

1. Deploy the default image for your job (this takes several minutes but should be executed only once per cluster for any MLRun/Iguazio upgrade):

```
from mlrun.runtimes import RemoteSparkRuntime
RemoteSparkRuntime.deploy_default_image()
```

2. Store your S3 credentials in a k8s *secret*:

```
import mlrun
secrets = {'s3_access_key': AWS_ACCESS_KEY, 's3_secret_key': AWS_SECRET_KEY}
mlrun.get_run_db().create_project_secrets(
    project = "uhuh-proj",
    provider=mlrun.api.schemas.SecretProviderName.kubernetes,
    secrets=secrets
)
```

Ingestion job code (to be executed in the remote pod):

```
# mlrun: start-code

from pyspark import SparkConf
from pyspark.sql import SparkSession


from mlrun.feature_store.api import ingest
def ingest_handler(context):
    conf = (SparkConf()
            .set("spark.hadoop.fs.s3a.path.style.access", True)
            .set("spark.hadoop.fs.s3a.access.key", context.get_secret('s3_access_key
↪'))
            .set("spark.hadoop.fs.s3a.secret.key", context.get_secret('s3_secret_key
↪'))
            .set("spark.hadoop.fs.s3a.endpoint", context.get_param("s3_endpoint"))
            .set("spark.hadoop.fs.s3a.region", context.get_param("s3_region"))
            .set("spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem")
            .set("com.amazonaws.services.s3.enableV4", True)
            .set("spark.driver.extraJavaOptions", "-Dcom.amazonaws.services.s3.
↪enableV4=true"))
    spark = (
        SparkSession.builder.config(conf=conf).appName("S3 app").getOrCreate()
    )

    ingest(mlrun_context=context, spark_context=spark)

# mlrun: end-code
```

Ingestion invocation:

```
from mlrun.datastore.sources import CSVSource
from mlrun.datastore.targets import ParquetTarget
from mlrun import code_to_function
```

(continues on next page)

(continued from previous page)

```
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

spark_service_name = "spark" # As configured & shown in the Iguazio dashboard

fn = code_to_function(kind='remote-spark',  name='func')

run_config = fstore.RunConfig(local=False, function=fn, handler="ingest_handler")
run_config.with_secret('kubernetes', ['s3_access_key', 's3_secret_key'])
run_config.parameters = {
    "s3_endpoint" : "s3.us-east-2.amazonaws.com",
    "s3_region" : "us-east-2"
}

target = ParquetTarget(
    path = "s3://my-s3-bucket/some/path",
    partitioned = False,
)

fstore.ingest(feature_set, source, targets=[target], run_config=run_config, spark_
↪context=spark_service_name)
```

2.1.13 Feature Store: Data retrieval

Learn how to retrieve and use data with the feature store.

In this section

Creating and using feature vectors

You can define a group of features from different feature sets as a *FeatureVector*.Feature vectors are used as an input for models, allowing you to define the feature vector once, and in turn create and track the datasets created from it or the online manifestation of the vector for real-time prediction needs.

The feature vector handles all the merging logic for you using an `asof merge` type merge that accounts for both the time and the entity. It ensures that all the latest relevant data is fetched, without concerns about "seeing the future" or other types of common time related errors.

In this section

- *Creating a feature vector*
- *Using a feature vector*
  - *Creating an offline feature vector*
  - *Creating an online feature vector*

Creating a feature vector

The feature vector object holds the following information:

- Name — the feature vector's name as will be later addressed in the store reference `store://feature_vectors/<project>/<feature-vector-name>` and the UI (after saving the vector).
- Description — a string description of the feature vector.
- Features — a list of features that comprise the feature vector.The feature list is defined by specifying the `<feature-set>.<feature-name>` for specific features or `<feature-set>.*` for all the feature set's features.
- Label feature — the feature that is the label for this specific feature vector, as a `<feature-set>.<feature-name>` string specification.

Example of creating a feature vector:

```
import mlrun.feature_store as fstore

# Feature vector definitions
feature_vector_name = 'example-fv'
feature_vector_description = 'Example feature vector'
features = ['data_source_1.*',
            'data_source_2.feature_1',
            'data_source_2.feature_2',
            'data_source_3.*']
label_feature = 'label_source_1.label_feature'

# Feature vector creation
fv = fstore.FeatureVector(name=feature_vector_name,
                          features=features,
                          label_feature=label_feature,
                          description=feature_vector_description)

# Save the feature vector in the MLRun DB
# so it will could be referenced by the `store://`
# and show in the UI
fv.save()
```

After saving the feature vector, it appears in the UI:

You can also view some metadata about the feature vector, including all the features, their types, a preview and statistics:

Using the feature vector

After a feature vector is saved, it can be used to create both offline (static) datasets and online (real-time) instances to supply as input to a machine learning model.

Creating an offline feature vector

Use the feature store's *get_offline_features()* function to produce a `dataset` from the feature vector. It creates the dataset (asynchronously if possible), saves it to the requested target, and returns a *OfflineVectorResponse*.Due to the async nature of this action, the response object contains an `fv_response.status` indicator that, once completed, could be directly turned into a `dataframe`, `parquet` or a `csv`.

`get_offline_features` expects to receive:

- **feature_vector** — a feature vector store reference or object.
- **entity_rows** — an optional dataframe that the features will be joined to.Defaults to the first feature set defined in the features vector's features list, and acts as the base for the vector's joins.
- **entity_timestamp_column** — an optional specific timestamp column (from the defined features) to act as the base timestamp column.Defaults to the base feature set's timestamp entity.
- **target** — a Feature Store target to write the results to.Defaults to return as a return value to the caller.
- **run_config** — an optional function or a *RunConfig* to run the feature vector creation process in a remote function.
- **drop_columns** — a list of columns to drop from the resulting feature vector. Optional.
- **start_time** — datetime, low limit of time needed to be filtered. Optional.
- **end_time** — datetime, high limit of time needed to be filtered. Optional.

Here's an example of a new dataset from a parquet target:

```
# Import the Parquet Target, so you can build your dataset from a parquet file
from mlrun.datastore.targets import ParquetTarget

# Get offline feature vector based on vector and parquet target
offline_fv = fstore.get_offline_features(feature_vector_name, target=ParquetTarget())

# Return dataset
dataset = offline_fv.to_dataframe()
```

Once an offline feature vector is created with a static target (such as `ParquetTarget()`) the reference to this dataset is saved as part of the feature vector's metadata and can now be referenced directly through the store as a function input using `store://feature-vectors/{project}/{feature_vector_name}`.

For example:

```
fn = mlrun.import_function('hub://sklearn-classifier').apply(auto_mount())

# Define the training task, including the feature vector and label
task = mlrun.new_task('training',
                      inputs={'dataset': f'store://feature-vectors/{project}/{feature_
↪vector_name}'},
                      params={'label_column': 'label'}
                     )

# Run the function
run = fn.run(task)
```

You can see a full example of using the offline feature vector to create an ML model in *part 2 of the end-to-end demo*.

Creating an online feature vector

The online feature vector provides real-time feature vectors to the model using the latest data available.

First create an `Online Feature Service` using *`get_online_feature_service()`*. Then feed the `Entity` of the feature vector to the service and receive the latest feature vector.

To create the *`OnlineVectorService`* you only need to pass it the feature vector's store reference.

```
import mlrun.feature_store as fstore

# Create the Feature Vector Online Service
feature_vector = 'store://feature-vectors/{project}/{feature_vector_name}'
svc = fstore.get_online_feature_service(feature_vector)
```

The online feature service supports value imputing (substitute NaN/Inf values with statistical or constant value). You can set the `impute_policy` parameter with the imputing policy, and specify which constant or statistical value will be used instead of NaN/Inf value. This can be defined per column or for all the columns (`"*"`). The replaced value can be a fixed number for constants or `$mean`, `$max`, `$min`, `$std`, `$count` for statistical values. `"*"` is used to specify the default for all features, for example:

```
svc = fstore.get_online_feature_service(feature_vector, impute_policy={"*": "$mean",
↪"age": 33})
```

To use the online feature service you need to supply a list of entities you want to get the feature vectors for. The service returns the feature vectors as a dictionary of `{<feature-name>: <feature-value>}` or simply a list of values as numpy arrays.

For example:

```
# Define the wanted entities
entities = [{<feature-vector-entity-column-name>: <entity>}]

# Get the feature vectors from the service
svc.get(entities)
```

The `entities` can be a list of dictionaries as shown in the example, or a list of lists where the values in the internal list correspond to the entity values (e.g. `entities = [["Joe"], ["Mike"]]`). The `.get()` method returns a dict by default. If you want to return an ordered list of values, set the `as_list` parameter to `True`. The list input is required by many ML frameworks and this eliminates additional glue logic.

See a full example of using the online feature service inside a serving function in *part 3 of the end-to-end demo*.

Retrieve offline data and use it for training

In this section

- *Creating an offline dataset*
- *Training*

Creating an offline dataset

An offline dataset is a specific instance of the feature vector definition. To create this instance, use the feature store's `get_offline_features(<feature_vector>, <target>)` function on the feature vector using the `store://<project_name>/<feature_vector>` reference and an offline target (as in Parquet, CSV, etc.).

```
import mlrun.feature_store as fstore

feature_vector = '<feature_vector_name>'
offline_fv = fstore.get_offline_features(feature_vector=feature_vector,␣
↪target=ParquetTarget())
```

Behind the scenes, `get_offline_features()` runs a local or Kubernetes job (can be specific by the `run_config` parameter) to retrieve all the relevant data from the feature sets, merge them and return it to the specified `target` which can be a local parquet, AZ Blob store or any other type of available storage.

Once instantiated with a target, the feature vector holds a reference to the instantiated dataset and references it as its current offline source.

You can also use MLRun's `log_dataset()` to log the specific dataset to the project as a specific dataset resource.

Training

Training your model using the feature store is a fairly simple task. (The offline dataset can also be used for your EDA.)

To retrieve a feature vector's offline dataset, use MLRun's data item mechanism, referencing the feature vector and specifying to receive it as a DataFrame.

```
df = mlrun.get_dataitem(f'store://feature-vectors/{project}/patient-deterioration').
↪as_df()
```

When trying to retrieve the dataset in your training function, you can put the feature vector reference as an input to the function and use the `as_df()` function to retrieve it automatically.

```
# A sample MLRun training function
def my_training_function(context, # MLRun context
                         dataset, # our feature vector reference
                         **kwargs):

    # retreieve the dataset
    df = dataset.as_df()

    # The rest of your training code...
```

And now you can create the MLRun function and run it locally or over the kubernetes cluster:

```
# Creating the training MLRun function with the code
fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')

# Creating the task to run the function with its dataset
task = mlrun.new_task('training',
                      inputs={'dataset': f'store://feature-vectors/{project}/{feature_
↪vector_name}'}) # The feature vector is given as an input to the function
```

(continues on next page)

(continued from previous page)

```
# Running the function over the kubernetes cluster
fn.run(task) # Set local=True to run locally
```

Online access and serving

In this section

- *Get online features*
- *Incorporating to the serving model*

Get online features

The online features are created ad-hoc using MLRun's feature store online feature service and are served from the **nosql** target for real-time performance needs.

To use it, first create an online feature service with the feature vector.

```
import mlrun.feature_store as fstore

svc = fstore.get_online_feature_service(<feature vector name>)
```

After creating the service you can use the feature vector's entity to get the latest feature vector for it. Pass a list of `{<key name>: <key value>}` pairs to receive a batch of feature vectors.

```
fv = svc.get([{<key name>: <key value>}])
```

Incorporating to the serving model

You can serve your models using the *model server*. (See an example.) You define a serving model class and the computational graph required to run your entire prediction pipeline and deploy it as a serverless function using nuclio.

To embed the online feature service in your model server, just create the feature vector service once when the model initializes, and then use it to retrieve the feature vectors of incoming keys.

You can import ready-made classes and functions from the MLRun function marketplace or write your own. As example of a scikit-learn based model server (taken from the feature store demo):

```
from cloudpickle import load
import numpy as np
import mlrun
import os

class ClassifierModel(mlrun.serving.V2ModelServer):

    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

        # Setup FS Online service
        self.feature_service = mlrun.feature_store.get_online_feature_service(
↪'patient-deterioration')
```

(continues on next page)

(continued from previous page)

```
        # Get feature vector statistics for imputing
        self.feature_stats = self.feature_service.vector.get_stats_table()

    def preprocess(self, body: dict, op) -> list:
        # Get patient feature vector
        # from the patient_id given in the request
        vectors = self.feature_service.get([{'patient_id': patient_id} for patient_id␣
↪in body['inputs']])

        # Impute inf's in the data to the feature's mean value
        # using the collected statistics from the Feature store
        feature_vectors = []
        for fv in vectors:
            new_vec = []
            for f, v in fv.items():
                if np.isinf(v):
                    new_vec.append(self.feature_stats.loc[f, 'mean'])
                else:
                    new_vec.append(v)
            feature_vectors.append(new_vec)

        # Set the final feature vector as the inputs
        # to pass to the predict function
        body['inputs'] = feature_vectors
        return body

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

Which you can deploy with:

```
# Create the serving function from the code above
fn = mlrun.code_to_function(<function_name>,
                            kind='serving')

# Add a specific model to the serving function
fn.add_model(<model_name>,
             class_name='ClassifierModel',
             model_path=<store_model_file_reference>)

# Enable MLRun's model monitoring
fn.set_tracking()

# Add the system mount to the function so
# it will have access to the model files
fn.apply(mlrun.mount_v3io())

# Deploy the function to the cluster
fn.deploy()
```

And test using:

```
fn.invoke('/v2/models/infer', body={<key name>: <key value>})
```

2.1.14 Feature Store tutorials

In this section

Feature store example (stocks)

This notebook demonstrates the following:

- Generate features and feature-sets
- Build complex transformations and ingest to offline and real-time data stores
- Fetch feature vectors for training
- Save feature vectors for re-use in real-time pipelines
- Access features and their statistics in real-time

In this section

- [Get started](#get started)
- *Create sample data for demo*
- *Define, infer and ingest feature sets*
- *Get an offline feature vector for training*
- [Initialize an online feature service and use it for real-time inference](#initialize-an-online feature-service-and-use-it-for-real-time-inference)

Get started

Install the latest MLRun package and restart the notebook

Setting up the environment and project

```
import mlrun
mlrun.set_environment(project="stocks")
```

```
> 2021-05-23 09:04:04,507 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-05-23 09:04:07,033 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': '0.6.4-rc3', 'client_version': 'unstable'}
```

```
('stocks', 'v3io:///projects/{{run.project}}/artifacts')
```

Create sample data for demo

```
import pandas as pd
quotes = pd.DataFrame(
    {
        "time": [
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.030"),
            pd.Timestamp("2016-05-25 13:30:00.041"),
            pd.Timestamp("2016-05-25 13:30:00.048"),
            pd.Timestamp("2016-05-25 13:30:00.049"),
            pd.Timestamp("2016-05-25 13:30:00.072"),
            pd.Timestamp("2016-05-25 13:30:00.075")
        ],
        "ticker": [
                "GOOG",
                "MSFT",
                "MSFT",
                "MSFT",
                "GOOG",
                "AAPL",
                "GOOG",
                "MSFT"
           ],
           "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.01],
           "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.03]
    }
)

trades = pd.DataFrame(
       {
           "time": [
               pd.Timestamp("2016-05-25 13:30:00.023"),
               pd.Timestamp("2016-05-25 13:30:00.038"),
               pd.Timestamp("2016-05-25 13:30:00.048"),
               pd.Timestamp("2016-05-25 13:30:00.048"),
               pd.Timestamp("2016-05-25 13:30:00.048")
           ],
           "ticker": ["MSFT", "MSFT", "GOOG", "GOOG", "AAPL"],
           "price": [51.95, 51.95, 720.77, 720.92, 98.0],
           "quantity": [75, 155, 100, 100, 100]
       }
)


stocks = pd.DataFrame(
       {
           "ticker": ["MSFT", "GOOG", "AAPL"],
           "name": ["Microsoft Corporation", "Alphabet Inc", "Apple Inc"],
           "exchange": ["NASDAQ", "NASDAQ", "NASDAQ"]
       }
)

import datetime
def move_date(df, col):
    max_date = df[col].max()
```

(continues on next page)

(continued from previous page)

```
    now_date = datetime.datetime.now()
    delta = now_date - max_date
    df[col] = df[col] + delta
    return df

quotes = move_date(quotes, "time")
trades = move_date(trades, "time")
```

View the demo data

```
quotes
```

```
                        time ticker     bid     ask
0 2021-05-23 09:04:07.013574   GOOG  720.50  720.93
1 2021-05-23 09:04:07.013574   MSFT   51.95   51.96
2 2021-05-23 09:04:07.020574   MSFT   51.97   51.98
3 2021-05-23 09:04:07.031574   MSFT   51.99   52.00
4 2021-05-23 09:04:07.038574   GOOG  720.50  720.93
5 2021-05-23 09:04:07.039574   AAPL   97.99   98.01
6 2021-05-23 09:04:07.062574   GOOG  720.50  720.88
7 2021-05-23 09:04:07.065574   MSFT   52.01   52.03
```

```
trades
```

```
                        time ticker   price  quantity
0 2021-05-23 09:04:07.041766   MSFT   51.95        75
1 2021-05-23 09:04:07.056766   MSFT   51.95       155
2 2021-05-23 09:04:07.066766   GOOG  720.77       100
3 2021-05-23 09:04:07.066766   GOOG  720.92       100
4 2021-05-23 09:04:07.066766   AAPL   98.00       100
```

```
stocks
```

```
  ticker                  name exchange
0   MSFT  Microsoft Corporation   NASDAQ
1   GOOG             Alphabet Inc   NASDAQ
2   AAPL                Apple Inc   NASDAQ
```

Define, infer and ingest feature sets

```
import mlrun.feature_store as fstore
from mlrun.feature_store.steps import *
from mlrun.features import MinMaxValidator
```

Build and ingest simple feature set (stocks)

```
# add feature set without time column (stock ticker metadata)
stocks_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")])
fstore.ingest(stocks_set, stocks, infer_options=fstore.InferOptions.default())
```

```
                         name exchange
ticker
MSFT    Microsoft Corporation   NASDAQ
GOOG              Alphabet Inc   NASDAQ
AAPL                 Apple Inc   NASDAQ
```

Build an advanced feature set - with feature engineering pipeline

Define a feature set with custom data processing and time aggregation functions

```
# create a new feature set
quotes_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")])
```

Define a custom pipeline step (python class)

```
class MyMap(MapClass):
    def __init__(self, multiplier=1, **kwargs):
        super().__init__(**kwargs)
        self._multiplier = multiplier

    def do(self, event):
        event["multi"] = event["bid"] * self._multiplier
        return event
```

Build and show the transformatiom pipeline

Use `storey` stream processing classes along with library and custom classes

```
quotes_set.graph.to("MyMap", multiplier=3)\
                .to("storey.Extend", _fn="({'extra': event['bid'] * 77})")\
                .to("storey.Filter", "filter", _fn="(event['bid'] > 51.92)")\
                .to(FeaturesetValidator())

quotes_set.add_aggregation("ask", ["sum", "max"], "1h", "10m", name="asks1")
quotes_set.add_aggregation("ask", ["sum", "max"], "5h", "10m", name="asks5")
quotes_set.add_aggregation("bid", ["min", "max"], "1h", "10m", name="bids")

# add feature validation policy
quotes_set["bid"] = fstore.Feature(validator=MinMaxValidator(min=52, severity="info"))

# add default target definitions and plot
quotes_set.set_targets()
quotes_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7fa9a4154250>
```

Test and show the pipeline results locally (allow to quickly develop and debug)

```
fstore.preview(
    quotes_set,
    quotes,
    entity_columns=["ticker"],
    timestamp_key="time",
    options=fstore.InferOptions.default(),
)
```

```
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574␣
↪args={'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574␣
↪args={'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574␣
↪args={'min': 52, 'value': 51.99}
```

```
        asks1_sum_1h  asks1_max_1h  asks5_sum_5h  asks5_max_5h  bids_min_1h  \
ticker
GOOG          720.93        720.93        720.93        720.93       720.50
MSFT           51.96         51.96         51.96         51.96        51.95
MSFT          103.94         51.98        103.94         51.98        51.95
MSFT          155.94         52.00        155.94         52.00        51.95
GOOG         1441.86        720.93       1441.86        720.93       720.50
AAPL           98.01         98.01         98.01         98.01        97.99
GOOG         2162.74        720.93       2162.74        720.93       720.50
MSFT          207.97         52.03        207.97         52.03        51.95

        bids_max_1h                       time     bid     ask    multi  \
ticker
GOOG         720.50 2021-05-23 09:04:07.013574  720.50  720.93  2161.50
MSFT          51.95 2021-05-23 09:04:07.013574   51.95   51.96   155.85
MSFT          51.97 2021-05-23 09:04:07.020574   51.97   51.98   155.91
MSFT          51.99 2021-05-23 09:04:07.031574   51.99   52.00   155.97
GOOG         720.50 2021-05-23 09:04:07.038574  720.50  720.93  2161.50
AAPL          97.99 2021-05-23 09:04:07.039574   97.99   98.01   293.97
GOOG         720.50 2021-05-23 09:04:07.062574  720.50  720.88  2161.50
MSFT          52.01 2021-05-23 09:04:07.065574   52.01   52.03   156.03

           extra
ticker
GOOG    55478.50
MSFT     4000.15
MSFT     4001.69
MSFT     4003.23
GOOG    55478.50
AAPL     7545.23
GOOG    55478.50
MSFT     4004.77
```

```
# print the feature set object
print(quotes_set.to_yaml())
```

```
kind: FeatureSet
metadata:
  name: stock-quotes
spec:
  entities:
```

(continues on next page)

(continued from previous page)

```
  - name: ticker
    value_type: str
  features:
  - name: asks1_sum_1h
    value_type: float
    aggregate: true
  - name: asks1_max_1h
    value_type: float
    aggregate: true
  - name: asks5_sum_5h
    value_type: float
    aggregate: true
  - name: asks5_max_5h
    value_type: float
    aggregate: true
  - name: bids_min_1h
    value_type: float
    aggregate: true
  - name: bids_max_1h
    value_type: float
    aggregate: true
  - name: bid
    value_type: float
    validator:
      kind: minmax
      severity: info
      min: 52
  - name: ask
    value_type: float
  - name: multi
    value_type: float
  - name: extra
    value_type: float
  partition_keys: []
  timestamp_key: time
  source:
    path: None
  targets:
  - name: parquet
    kind: parquet
  - name: nosql
    kind: nosql
  graph:
    states:
      MyMap:
        kind: task
        class_name: MyMap
        class_args:
          multiplier: 3
      storey.Extend:
        kind: task
        class_name: storey.Extend
        class_args:
          _fn: '({''extra'': event[''bid''] * 77})'
        after:
        - MyMap
      filter:
```

(continues on next page)

(continued from previous page)

```
        kind: task
        class_name: storey.Filter
        class_args:
          _fn: (event['bid'] > 51.92)
        after:
        - storey.Extend
      FeaturesetValidator:
        kind: task
        class_name: mlrun.feature_store.steps.FeaturesetValidator
        class_args:
          featureset: .
          columns: null
        after:
        - filter
      Aggregates:
        kind: task
        class_name: storey.AggregateByKey
        class_args:
          aggregates:
          - name: asks1
            column: ask
            operations:
            - sum
            - max
            windows:
            - 1h
            period: 10m
          - name: asks5
            column: ask
            operations:
            - sum
            - max
            windows:
            - 5h
            period: 10m
          - name: bids
            column: bid
            operations:
            - min
            - max
            windows:
            - 1h
            period: 10m
          table: .
        after:
        - FeaturesetValidator
  output_path: v3io:///projects/{{run.project}}/artifacts
status:
  state: created
  stats:
    ticker:
      count: 8
      unique: 3
      top: MSFT
      freq: 4
    asks1_sum_1h:
      count: 8.0
```

(continues on next page)

(continued from previous page)

```
      mean: 617.9187499999999
      min: 51.96
      max: 2162.74
      std: 784.8779804245735
      hist:
      - - 4
        - 1
        - 0
        - 0
        - 0
        - 0
        - 1
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 1
        - 0
        - 0
        - 0
        - 0
        - 0
        - 1
      - - 51.96
        - 157.499
        - 263.03799999999995
        - 368.57699999999994
        - 474.11599999999993
        - 579.655
        - 685.194
        - 790.733
        - 896.2719999999999
        - 1001.8109999999999
        - 1107.35
        - 1212.889
        - 1318.4279999999999
        - 1423.9669999999999
        - 1529.5059999999999
        - 1635.0449999999998
        - 1740.5839999999998
        - 1846.1229999999998
        - 1951.6619999999998
        - 2057.2009999999996
        - 2162.74
    asks1_max_1h:
      count: 8.0
      mean: 308.59625
      min: 51.96
      max: 720.93
      std: 341.7989955655851
      hist:
      - - 4
        - 1
        - 0
        - 0
```

(continues on next page)

(continued from previous page)

```
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.96
      - 85.4085
      - 118.857
      - 152.3055
      - 185.754
      - 219.2025
      - 252.65099999999998
      - 286.0995
      - 319.54799999999994
      - 352.9964999999999
      - 386.44499999999994
      - 419.89349999999996
      - 453.3419999999999
      - 486.7904999999999
      - 520.2389999999999
      - 553.6875
      - 587.136
      - 620.5844999999999
      - 654.0329999999999
      - 687.4815
      - 720.93
  asks5_sum_5h:
    count: 8.0
    mean: 617.9187499999999
    min: 51.96
    max: 2162.74
    std: 784.8779804245735
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
```

(continues on next page)

(continued from previous page)


```
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 1
    - - 51.96
      - 157.499
      - 263.03799999999995
      - 368.57699999999994
      - 474.11599999999993
      - 579.655
      - 685.194
      - 790.733
      - 896.2719999999999
      - 1001.8109999999999
      - 1107.35
      - 1212.889
      - 1318.4279999999999
      - 1423.9669999999999
      - 1529.5059999999999
      - 1635.0449999999998
      - 1740.5839999999998
      - 1846.1229999999998
      - 1951.6619999999998
      - 2057.2009999999996
      - 2162.74
  asks5_max_5h:
    count: 8.0
    mean: 308.59625
    min: 51.96
    max: 720.93
    std: 341.7989955655851
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.96
      - 85.4085
```


(continues on next page)

(continued from previous page)

```
      - 118.857
      - 152.3055
      - 185.754
      - 219.2025
      - 252.65099999999998
      - 286.0995
      - 319.54799999999994
      - 352.9964999999999
      - 386.44499999999994
      - 419.89349999999996
      - 453.3419999999999
      - 486.7904999999999
      - 520.2389999999999
      - 553.6875
      - 587.136
      - 620.5844999999999
      - 654.0329999999999
      - 687.4815
      - 720.93
  bids_min_1h:
    count: 8.0
    mean: 308.41125
    min: 51.95
    max: 720.5
    std: 341.59667259325835
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
```

(continues on next page)

(continued from previous page)

```
      - 419.6524999999999
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
      - 720.5
  bids_max_1h:
    count: 8.0
    mean: 308.42625
    min: 51.95
    max: 720.5
    std: 341.58380276661245
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
      - 419.6524999999999
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
```

(continues on next page)

(continued from previous page)

```
      - 720.5
  time:
    count: 8
    mean: '2021-05-23 09:04:07.035699200'
    min: '2021-05-23 09:04:07.013574'
    max: '2021-05-23 09:04:07.065574'
  bid:
    count: 8.0
    mean: 308.42625
    min: 51.95
    max: 720.5
    std: 341.58380276661245
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
      - 419.6524999999999
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
      - 720.5
  ask:
    count: 8.0
    mean: 308.59
```

(continues on next page)

(continued from previous page)

```
      min: 51.96
      max: 720.93
      std: 341.79037903369954
      hist:
      - - 4
        - 1
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 0
        - 3
      - - 51.96
        - 85.4085
        - 118.857
        - 152.3055
        - 185.754
        - 219.2025
        - 252.65099999999998
        - 286.0995
        - 319.54799999999994
        - 352.9964999999999
        - 386.44499999999994
        - 419.89349999999996
        - 453.3419999999999
        - 486.7904999999999
        - 520.2389999999999
        - 553.6875
        - 587.136
        - 620.5844999999999
        - 654.0329999999999
        - 687.4815
        - 720.93
    multi:
      count: 8.0
      mean: 925.27875
      min: 155.85000000000002
      max: 2161.5
      std: 1024.7514082998375
      hist:
      - - 4
        - 1
        - 0
        - 0
        - 0
```

(continues on next page)

(continued from previous page)

```
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 155.85000000000002
      - 256.13250000000005
      - 356.415
      - 456.6975
      - 556.98
      - 657.2625
      - 757.545
      - 857.8275
      - 958.11
      - 1058.3925
      - 1158.6750000000002
      - 1258.9575
      - 1359.2399999999998
      - 1459.5225
      - 1559.8049999999998
      - 1660.0875
      - 1760.37
      - 1860.6525000000001
      - 1960.935
      - 2061.2175
      - 2161.5
  extra:
    count: 8.0
    mean: 23748.82125
    min: 4000.15
    max: 55478.5
    std: 26301.95281302916
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
```

(continues on next page)

(continued from previous page)

```
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 4000.15
      - 6574.0675
      - 9147.985
      - 11721.9025
      - 14295.82
      - 16869.7375
      - 19443.655000000002
      - 22017.572500000002
      - 24591.49
      - 27165.4075
      - 29739.325
      - 32313.2425
      - 34887.16
      - 37461.0775
      - 40034.995
      - 42608.9125
      - 45182.83
      - 47756.747500000005
      - 50330.665
      - 52904.582500000004
      - 55478.5
preview:
- - asks1_sum_1h
  - asks1_max_1h
  - asks5_sum_5h
  - asks5_max_5h
  - bids_min_1h
  - bids_max_1h
  - time
  - bid
  - ask
  - multi
  - extra
- - 720.93
  - 720.93
  - 720.93
  - 720.93
  - 720.5
  - 720.5
  - 2021-05-23T09:04:07.013574
  - 720.5
  - 720.93
  - 2161.5
  - 55478.5
- - 51.96
  - 51.96
  - 51.96
  - 51.96
  - 51.95
  - 51.95
  - 2021-05-23T09:04:07.013574
```

(continues on next page)

(continued from previous page)

```
    - 51.95
    - 51.96
    - 155.85000000000002
    - 4000.15
  - - 103.94
    - 51.98
    - 103.94
    - 51.98
    - 51.95
    - 51.97
    - 2021-05-23T09:04:07.020574
    - 51.97
    - 51.98
    - 155.91
    - 4001.69
  - - 155.94
    - 52.0
    - 155.94
    - 52.0
    - 51.95
    - 51.99
    - 2021-05-23T09:04:07.031574
    - 51.99
    - 52.0
    - 155.97
    - 4003.23
  - - 1441.86
    - 720.93
    - 1441.86
    - 720.93
    - 720.5
    - 720.5
    - 2021-05-23T09:04:07.038574
    - 720.5
    - 720.93
    - 2161.5
    - 55478.5
  - - 98.01
    - 98.01
    - 98.01
    - 98.01
    - 97.99
    - 97.99
    - 2021-05-23T09:04:07.039574
    - 97.99
    - 98.01
    - 293.96999999999997
    - 7545.23
  - - 2162.74
    - 720.93
    - 2162.74
    - 720.93
    - 720.5
    - 720.5
    - 2021-05-23T09:04:07.062574
    - 720.5
    - 720.88
```

(continues on next page)

(continued from previous page)

```
    - 2161.5
    - 55478.5
  - - 207.97
    - 52.03
    - 207.97
    - 52.03
    - 51.95
    - 52.01
    - 2021-05-23T09:04:07.065574
    - 52.01
    - 52.03
    - 156.03
    - 4004.77
```

Ingest data into offline and online stores

This writes to both targets (Parquet and NoSQL).

```
# save ingest data and print the FeatureSet spec
df = fstore.ingest(quotes_set, quotes)
```

```
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574␣
↪args={'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574␣
↪args={'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574␣
↪args={'min': 52, 'value': 51.99}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574␣
↪args={'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574␣
↪args={'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574␣
↪args={'min': 52, 'value': 51.99}
```

Get an offline feature vector for training

Example of combining features from 3 sources with time travel join of 3 tables with **time travel**.

Specify a set of features and request the feature vector offline result as a dataframe

```
features = [
    "stock-quotes.multi",
    "stock-quotes.asks5_sum_5h as total_ask",
    "stock-quotes.bids_min_1h",
    "stock-quotes.bids_max_1h",
    "stocks.*",
]

vector = fstore.FeatureVector("stocks-vec", features, description="stocks demo␣
↪feature vector")
vector.save()
```

```
resp = fstore.get_offline_features(vector, entity_rows=trades, entity_timestamp_
↪column="time")
resp.to_dataframe()
```

```
    price  quantity    multi  total_ask  bids_min_1h  bids_max_1h  \
0   51.95        75   155.97     155.94        51.95        51.99
1   51.95       155   155.97     155.94        51.95        51.99
2  720.77       100  2161.50    2162.74       720.50       720.50
3  720.92       100  2161.50    2162.74       720.50       720.50
4   98.00       100   293.97      98.01        97.99        97.99

                   name exchange
0  Microsoft Corporation   NASDAQ
1  Microsoft Corporation   NASDAQ
2          Alphabet Inc   NASDAQ
3          Alphabet Inc   NASDAQ
4             Apple Inc   NASDAQ
```

Initialize an online feature service and use it for real-time inference

```
service = fstore.get_online_feature_service("stocks-vec")
```

Request feature vector statistics, can be used for imputing or validation

```
service.vector.get_stats_table()
```

```
             count       mean     min      max          std  \
multi          8.0  925.27875  155.85  2161.50  1024.751408
total_ask      8.0  617.91875   51.96  2162.74   784.877980
bids_min_1h    8.0  308.41125   51.95   720.50   341.596673
bids_max_1h    8.0  308.42625   51.95   720.50   341.583803
name           3.0        NaN     NaN      NaN          NaN
exchange       3.0        NaN     NaN      NaN          NaN

                                                      hist  unique  \
multi        [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
total_ask    [[4, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0,...     NaN
bids_min_1h  [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
bids_max_1h  [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
name                                                       NaN     3.0
exchange                                                   NaN     1.0

                    top  freq
multi               NaN   NaN
total_ask           NaN   NaN
bids_min_1h         NaN   NaN
bids_max_1h         NaN   NaN
name        Alphabet Inc   1.0
exchange          NASDAQ   3.0
```

Real-time feature vector request

```
service.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
```

```
[{'asks5_sum_5h': 2162.74,
  'bids_min_1h': 720.5,
  'bids_max_1h': 720.5,
  'multi': 2161.5,
  'name': 'Alphabet Inc',
  'exchange': 'NASDAQ',
  'total_ask': None},
 {'asks5_sum_5h': 207.97,
  'bids_min_1h': 51.95,
  'bids_max_1h': 52.01,
  'multi': 156.03,
  'name': 'Microsoft Corporation',
  'exchange': 'NASDAQ',
  'total_ask': None}]
```

```
service.get([{"ticker": "AAPL"}])
```

```
[{'asks5_sum_5h': 98.01,
  'bids_min_1h': 97.99,
  'bids_max_1h': 97.99,
  'multi': 293.97,
  'name': 'Apple Inc',
  'exchange': 'NASDAQ',
  'total_ask': None}]
```

```
service.close()
```

Feature store end-to-end demo

This demo shows the usage of MLRun and the feature store:

- ***Data ingestion & preparation***
- ***Model training & testing***
- ***Model serving***
- ***Building an automated ML pipeline***

Fraud prevention, specifically, is a challenge since it requires processing raw transactions and events in real-time and being able to quickly respond and block transactions before they occur. Consider, for example, a case where you would like to evaluate the average transaction amount. When training the model, it is common to take a DataFrame and just calculate the average. However, when dealing with real-time/online scenarios, this average has to be calculated incrementally.

This demo illustrates how to **Ingest** different data sources to the **Feature Store**. Specifically, it covers two types of data:

- **Transactions**: Monetary activity between two parties to transfer funds.
- **Events**: Activity performed by a party, such as login or password change.

The demo walks through creation of an ingestion pipeline for each data source with all the needed preprocessing and validation. It runs the pipeline locally within the notebook and then launches a real-time function to **ingest live data** or schedule a cron to run the task when needed.

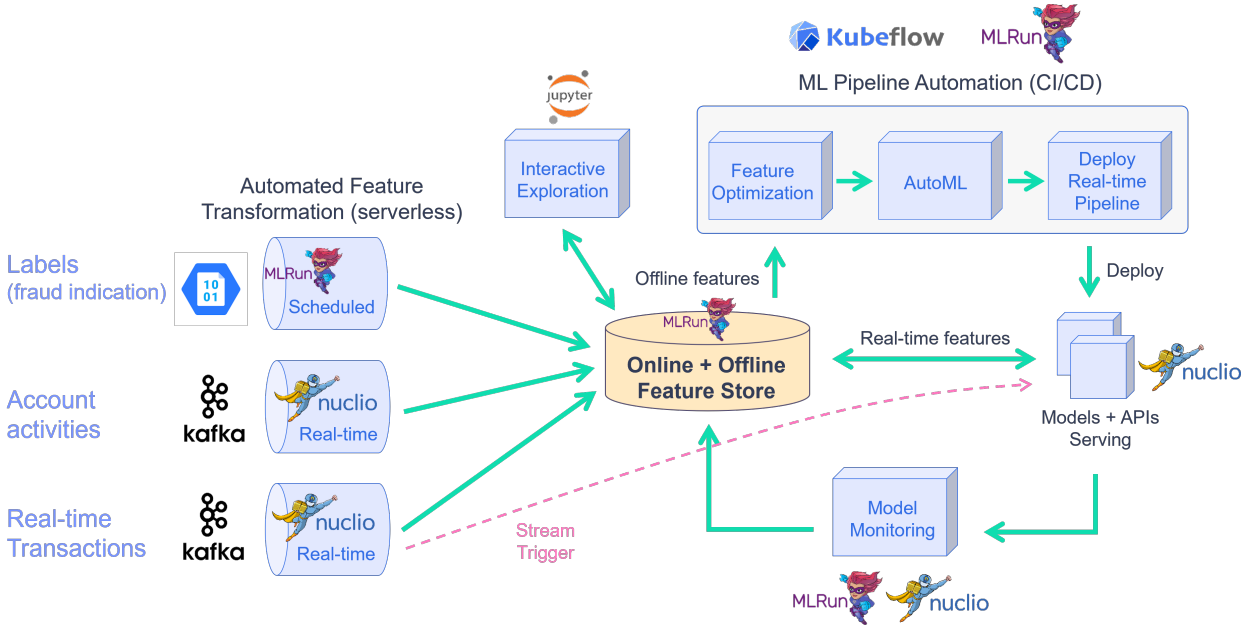
Following the ingestion, you create a feature vector, select the most relevant features and create a final model. Then you deploy the model and showcase the feature vector and model serving.

Part 1: Data Ingestion

This demo showcases financial fraud prevention. It uses the MLRun feature store to define complex features that help identify fraud.

Fraud prevention is a special challenge since it requires processing raw transaction and events in real-time and being able to quickly respond and block transactions before they occur.

To address this, you'll create a development pipeline and a production pipeline. Both pipelines share the same feature engineering and model code, but serve data very differently. Furthermore, MLRun automates the data and model monitoring process, drift identification, and trigger retraining in a CI/CD pipeline. This process is described in the diagram below:



The raw data is described as follows:

TRANSACTIONS			USER EVENTS	
age	age group value 0-6. Some values are marked as U for unknown		**source**	The party/entity related to the event
gender	A character to define the age		**event**	event, such as login or password change
zipcodeOri	ZIP code of the person originating the transaction		**timestamp**	The date and time of the event
zipMerchant	ZIP code of the merchant receiving the transaction			
category	category of the transaction (e.g., transportation, food, etc.)			
amount	the total amount of the transaction			
fraud	whether the transaction is fraudulent			
timestamp	the date and time in which the transaction took place			
source	the ID of the party/entity performing the transaction			
target	the ID of the party/entity receiving the transaction			
device	the device ID used to perform the transaction			

This notebook introduces how to **Ingest** different data sources to the **Feature Store**.

The following FeatureSets are created:

- **Transactions**: Monetary transactions between a source and a target.
- **Events**: Account events such as account login or a password change.
- **Label**: Fraud label for the data.

By the end of this tutorial you'll know how to:

- Create an ingestion pipeline for each data source.
- Define preprocessing, aggregation, and validation of the pipeline.
- Run the pipeline locally within the notebook.
- Launch a real-time function to ingest live data.
- Schedule a cron to run the task when needed.

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2022-03-16 05:45:07,703 [info] loaded project fraud-demo from MLRun DB
```

Step 1 - Fetch, process and ingest the datasets

1.1 - Transactions

Transactions

```
# Helper functions to adjust the timestamps of our data
# while keeping the order of the selected events and
# the relative distance from one event to the other

def date_adjustment(sample, data_max, new_max, old_data_period, new_data_period):
    '''
        Adjust a specific sample's date according to the original and new time periods
    '''
    sample_dates_scale = ((data_max - sample) / old_data_period)
    sample_delta = new_data_period * sample_dates_scale
    new_sample_ts = new_max - sample_delta
    return new_sample_ts

def adjust_data_timespan(dataframe, timestamp_col='timestamp', new_period='2d', new_
↪max_date_str='now'):
    '''
        Adjust the dataframe timestamps to the new time period
    '''
    # Calculate old time period
    data_min = dataframe.timestamp.min()
    data_max = dataframe.timestamp.max()
    old_data_period = data_max-data_min

    # Set new time period
    new_time_period = pd.Timedelta(new_period)
    new_max = pd.Timestamp(new_max_date_str)
    new_min = new_max-new_time_period
    new_data_period = new_max-new_min

    # Apply the timestamp change
    df = dataframe.copy()
    df[timestamp_col] = df[timestamp_col].apply(lambda x: date_adjustment(x, data_max,
↪ new_max, old_data_period, new_data_period))
    return df
```

```
import pandas as pd

# Fetch the transactions dataset from the server
transactions_data = pd.read_csv('https://s3.wasabisys.com/iguazio/data/fraud-demo-
↪mlrun-fs-docs/data.csv', parse_dates=['timestamp'], nrows=500)

# Adjust the samples timestamp for the past 2 days
transactions_data = adjust_data_timespan(transactions_data, new_period='2d')

# Preview
transactions_data.head(3)
```

```
   step age gender  zipcodeOri  zipMerchant          category  amount  fraud  \
0     0   4      M       28007        28007  es_transportation    4.55      0
```

(continues on next page)

(continued from previous page)

```
1     0   2      M       28007        28007  es_transportation   39.68      0
2     0   4      F       28007        28007  es_transportation   26.89      0

                      timestamp       source       target  \
0 2022-03-15 16:13:54.851486383  C1093826151   M348934600
1 2022-03-14 09:21:09.710448366   C352968107   M348934600
2 2022-03-15 22:41:20.666966912  C2054744914  M1823072687

                             device
0  f802e61d76564b7a89a83adcdfa573da
1  38ef7fc3eb7442c8ae64579a483f1d2b
2  7a851d0758894078b5846851ae32d5e3
```

Transactions - create a feature set and preprocessing pipeline

Create the feature set (data pipeline) definition for the **credit transaction processing** that describes the offline/online data transformations and aggregations. The feature store automatically adds an offline `parquet` target and an online `NoSQL` target by using `set_targets()`.

The data pipeline consists of:

- **Extracting** the data components (hour, day of week)
- **Mapping** the age values
- **One hot encoding** for the transaction category and the gender
- **Aggregating** the amount (avg, sum, count, max over 2/12/24 hour time windows)
- **Aggregating** the transactions per category (over 14 days time windows)
- **Writing** the results to **offline** (Parquet) and **online** (NoSQL) targets

```
# Import MLRun's Feature Store
import mlrun.feature_store as fstore
from mlrun.feature_store.steps import OneHotEncoder, MapValues, DateExtractor
```

```
# Define the transactions FeatureSet
transaction_set = fstore.FeatureSet("transactions",
                                    entities=[fstore.Entity("source")],
                                    timestamp_key='timestamp',
                                    description="transactions feature set")
```

```
# Define and add value mapping
main_categories = ["es_transportation", "es_health", "es_otherservices",
       "es_food", "es_hotelservices", "es_barsandrestaurants",
       "es_tech", "es_sportsandtoys", "es_wellnessandbeauty",
       "es_hyper", "es_fashion", "es_home", "es_contents",
       "es_travel", "es_leisure"]

# One Hot Encode the newly defined mappings
one_hot_encoder_mapping = {'category': main_categories,
                           'gender': list(transactions_data.gender.unique())}

# Define the graph steps
transaction_set.graph\
```

(continues on next page)

(continued from previous page)

```
    .to(DateExtractor(parts = ['hour', 'day_of_week'], timestamp_col = 'timestamp'))\
    .to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))\
    .to(OneHotEncoder(mapping=one_hot_encoder_mapping))


# Add aggregations for 2, 12, and 24 hour time windows
transaction_set.add_aggregation(name='amount',
                                column='amount',
                                operations=['avg','sum', 'count','max'],
                                windows=['2h', '12h', '24h'],
                                period='1h')


# Add the category aggregations over a 14 day window
for category in main_categories:
    transaction_set.add_aggregation(name=category,column=f'category_{category}',
                                    operations=['count'], windows=['14d'], period='1d
↪')

# Add default (offline-parquet & online-nosql) targets
transaction_set.set_targets()

# Plot the pipeline so we can see the different steps
transaction_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88af0f9cd0>
```

Transactions - ingestion

```
# Ingest the transactions dataset through the defined pipeline
transactions_df = fstore.ingest(transaction_set, transactions_data,
                 infer_options=fstore.InferOptions.default())

transactions_df.head(3)
```

```
persist count = 0
persist count = 100
persist count = 200
persist count = 300
persist count = 400
persist count = 500
persist count = 600
persist count = 700
persist count = 800
persist count = 900
persist count = 1000
```

```
              amount_count_2h  amount_count_12h  amount_count_24h  \
source
C1093826151              1.0               1.0               1.0
C352968107               1.0               1.0               1.0
C2054744914              1.0               1.0               1.0
```

(continues on next page)

(continued from previous page)

```
             amount_max_2h  amount_max_12h  amount_max_24h  amount_sum_2h  \
source
C1093826151           4.55            4.55            4.55           4.55
C352968107           39.68           39.68           39.68          39.68
C2054744914          26.89           26.89           26.89          26.89

             amount_sum_12h  amount_sum_24h  amount_avg_2h  ...  \
source                                                      ...
C1093826151            4.55            4.55           4.55  ...
C352968107            39.68           39.68          39.68  ...
C2054744914           26.89           26.89          26.89  ...

             category_es_contents  category_es_travel  category_es_leisure  \
source
C1093826151                     0                   0                    0
C352968107                      0                   0                    0
C2054744914                     0                   0                    0

             amount  fraud                     timestamp       target  \
source
C1093826151    4.55      0 2022-03-15 16:13:54.851486383   M348934600
C352968107    39.68      0 2022-03-14 09:21:09.710448366   M348934600
C2054744914   26.89      0 2022-03-15 22:41:20.666966912  M1823072687

                                  device  timestamp_hour  \
source
C1093826151  f802e61d76564b7a89a83adcdfa573da              16
C352968107   38ef7fc3eb7442c8ae64579a483f1d2b               9
C2054744914  7a851d0758894078b5846851ae32d5e3              22

             timestamp_day_of_week
source
C1093826151                      1
C352968107                       0
C2054744914                      1

[3 rows x 57 columns]
```

1.2 - User events

User events - fetching

```
# Fetch the user_events dataset from the server
user_events_data = pd.read_csv('https://s3.wasabisys.com/iguazio/data/fraud-demo-
↪mlrun-fs-docs/events.csv',
                               index_col=0, quotechar="\'", parse_dates=['timestamp'],
↪ nrows=500)

# Adjust to the last 2 days to see the latest aggregations in our online feature␣
↪vectors
user_events_data = adjust_data_timespan(user_events_data, new_period='2d')

# Preview
user_events_data.head(3)
```

```
        source           event                       timestamp
0  C1974668487  details_change 2022-03-15 15:03:17.518565985
1  C1973547259           login 2022-03-15 18:05:50.652706656
2   C515668508           login 2022-03-15 14:37:49.845093748
```

User events - create a feature set and preprocessing pipeline

Define the events feature set. This is a fairly straightforward pipeline in which you only "one hot encode" the event categories and save the data to the default targets.

```
user_events_set = fstore.FeatureSet("events",
                            entities=[fstore.Entity("source")],
                            timestamp_key='timestamp',
                            description="user events feature set")
```

```
# Define and add value mapping
events_mapping = {'event': list(user_events_data.event.unique())}

# One Hot Encode
user_events_set.graph.to(OneHotEncoder(mapping=events_mapping))

# Add default (offline-parquet & online-nosql) targets
user_events_set.set_targets()

# Plot the pipeline so we can see the different steps
user_events_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88ad6f1fd0>
```

User events - ingestion

```
# Ingestion of the newly created events feature set
events_df = fstore.ingest(user_events_set, user_events_data)
events_df.head(3)
```

```
persist count = 0
persist count = 100
persist count = 200
persist count = 300
persist count = 400
persist count = 500
```

```
             event_details_change  event_login  event_password_change  \
source
C1974668487                     1            0                      0
C1973547259                     0            1                      0
C515668508                      0            1                      0

                              timestamp
source
C1974668487 2022-03-15 15:03:17.518565985
```

(continues on next page)

(continued from previous page)

```
C1973547259 2022-03-15 18:05:50.652706656
C515668508  2022-03-15 14:37:49.845093748
```

Step 2 - Create a labels dataset for model training

Label set - create a feature set

This feature set contains the label for the fraud demo, it is ingested directly to the default targets without any changes

```
def create_labels(df):
    labels = df[['fraud','source','timestamp']].copy()
    labels = labels.rename(columns={"fraud": "label"})
    labels['timestamp'] = labels['timestamp'].astype("datetime64[ms]")
    labels['label'] = labels['label'].astype(int)
    labels.set_index('source', inplace=True)
    return labels
```

```
# Define the "labels" feature set
labels_set = fstore.FeatureSet("labels",
                           entities=[fstore.Entity("source")],
                           timestamp_key='timestamp',
                           description="training labels",
                           engine="pandas")

labels_set.graph.to(name="create_labels", handler=create_labels)

# specify only Parquet (offline) target since its not used for real-time
labels_set.set_targets(['parquet'], with_defaults=False)
labels_set.plot(with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88a3561f90>
```

Label set - ingestion

```
# Ingest the labels feature set
labels_df = fstore.ingest(labels_set, transactions_data)
labels_df.head(3)
```

```
             label               timestamp
source
C1093826151      0 2022-03-15 16:13:54.851
C352968107       0 2022-03-14 09:21:09.710
C2054744914      0 2022-03-15 22:41:20.666
```

Step 3 - Deploy a real-time pipeline

When dealing with real-time aggregation, it's important to be able to update these aggregations in real-time. For this purpose, you'll create live serving functions that update the online feature store of the `transactions` FeatureSet and `Events` FeatureSet.

Using MLRun's `serving` runtime, create a nuclio function loaded with the feature set's computational graph definition and an `HttpSource` to define the HTTP trigger.

Notice that the implementation below does not require any rewrite of the pipeline logic.

3.1 - Transactions

Transactions - deploy the feature set live endpoint

```
# Create iguazio v3io stream and transactions push API endpoint
transaction_stream = f'v3io:///projects/{project.name}/streams/transaction'
transaction_pusher = mlrun.datastore.get_stream_pusher(transaction_stream)
```

```
# Define the source stream trigger (use v3io streams)
# Define the `key` and `time` fields (extracted from the Json message).
source = mlrun.datastore.sources.StreamSource(path=transaction_stream , key_field=
↪'source', time_field='timestamp')

# Deploy the transactions feature set's ingestion service over a real-time (Nuclio)␣
↪serverless function
# you can use the run_config parameter to pass function/service specific configuration
transaction_set_endpoint = fstore.deploy_ingestion_service(featureset=transaction_set,
↪ source=source)
```

```
> 2022-03-16 05:45:43,035 [info] Starting remote function deploy
2022-03-16 05:45:43  (info) Deploying function
2022-03-16 05:45:43  (info) Building
2022-03-16 05:45:43  (info) Staging files and preparing base images
2022-03-16 05:45:43  (warn) Python 3.6 runtime is deprecated and will soon not be␣
↪supported. Please migrate your code and use Python 3.7 runtime (`python:3.7`) or␣
↪higher
2022-03-16 05:45:43  (info) Building processor image
2022-03-16 05:47:03  (info) Build complete
2022-03-16 05:47:08  (info) Function deploy complete
> 2022-03-16 05:47:08,835 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-fraud-demo-admin-transactions-ingest.default-tenant.svc.
↪cluster.local:8080'], 'external_invocation_urls': ['fraud-demo-admin-transactions-
↪ingest-fraud-demo-admin.default-tenant.app.xtvtjecfcssi.iguazio-cd1.com/']}
```

Transactions - test the feature set HTTP endpoint

By defining the `transactions` feature set you can now use MLRun and Storey to deploy it as a live endpoint, ready to ingest new data!

Using MLRun's `serving` runtime, create a nuclio function loaded with the feature set's computational graph definition and an `HttpSource` to define the HTTP trigger.

```
import requests
import json

# Select a sample from the dataset and serialize it to JSON
transaction_sample = json.loads(transactions_data.sample(1).to_json(orient='records
↪'))[0]
transaction_sample['timestamp'] = str(pd.Timestamp.now())
transaction_sample
```

```
{'step': 0,
 'age': '5',
 'gender': 'M',
 'zipcodeOri': 28007,
 'zipMerchant': 28007,
 'category': 'es_transportation',
 'amount': 2.19,
 'fraud': 0,
 'timestamp': '2022-03-16 05:47:08.884971',
 'source': 'C546957379',
 'target': 'M348934600',
 'device': '8ee3b24f2eb143759e938b6148da547c'}
```

```
# Post the sample to the ingestion endpoint
requests.post(transaction_set_endpoint, json=transaction_sample).text
```

```
'{"id": "c3f8a087-3932-45c0-8dcb-1fd8efabe681"}'
```

3.2 - User events

User events - deploy the feature set live endpoint

Deploy the events feature set's ingestion service using the feature set and all the previously defined resources.

```
# Create iguazio v3io stream and transactions push API endpoint
events_stream = f'v3io:///projects/{project.name}/streams/events'
events_pusher = mlrun.datastore.get_stream_pusher(events_stream)
```

```
# Define the source stream trigger (use v3io streams)
# Define the `key` and `time` fields (extracted from the Json message).
source = mlrun.datastore.sources.StreamSource(path=events_stream , key_field='source',
↪ time_field='timestamp')

# Deploy the transactions feature set's ingestion service over a real-time (Nuclio)␣
↪serverless function
# you can use the run_config parameter to pass function/service specific configuration
events_set_endpoint = fstore.deploy_ingestion_service(featureset=user_events_set,␣
↪source=source)
```

(continues on next page)

(continued from previous page)

```
> 2022-03-16 05:47:09,035 [info] Starting remote function deploy
2022-03-16 05:47:09  (info) Deploying function
2022-03-16 05:47:09  (info) Building
2022-03-16 05:47:09  (info) Staging files and preparing base images
2022-03-16 05:47:09  (warn) Python 3.6 runtime is deprecated and will soon not be␣
↪supported. Please migrate your code and use Python 3.7 runtime (`python:3.7`) or␣
↪higher
2022-03-16 05:47:09  (info) Building processor image
```

User events - test the feature set HTTP endpoint

```
# Select a sample from the events dataset and serialize it to JSON
user_events_sample = json.loads(user_events_data.sample(1).to_json(orient='records
↪'))[0]
user_events_sample['timestamp'] = str(pd.Timestamp.now())
user_events_sample
```

```
# Post the sample to the ingestion endpoint
requests.post(events_set_endpoint, json=user_events_sample).text
```

Done!

You've completed Part 1 of the data-ingestion with the feature store. Proceed to *Part 2* to learn how to train an ML model using the feature store data.

Part 2: Training

This part shows how to use MLRun's **Feature Store** to easily define a **Feature Vector** and create the dataset you need to run the training process.By the end of this tutorial you'll learn how to:

- Combine multiple data sources to a single Feature Vector
- Create training dataset
- Create a model using an MLRun Hub function

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-09-19 17:59:27,165 [info] loaded project fraud-demo from MLRun DB
```

Step 1 - Create a feature vector

In this section you create the Feature Vector.The Feature vector has a `name` so you can reference to it later via the URI or the serving function, and a list of `features` from the available FeatureSets. You can add a feature from a feature set by adding `<FeatureSet>.<Feature>` to the list, or add `<FeatureSet>.*` to add all the FeatureSet's available features.

By default, the first FeatureSet in the feature list acts as the spine, meaning that all the other features will be joined to it.For example, in this instance the spine is the early_sense sensor data, so for each early_sense event we will create produce a row in the resulted Feature Vector.

```
# Define the list of features you will be using
features = ['transactions.amount_max_2h',
            'transactions.amount_sum_2h',
            'transactions.amount_count_2h',
            'transactions.amount_avg_2h',
            'transactions.amount_max_12h',
            'transactions.amount_sum_12h',
            'transactions.amount_count_12h',
            'transactions.amount_avg_12h',
            'transactions.amount_max_24h',
            'transactions.amount_sum_24h',
            'transactions.amount_count_24h',
            'transactions.amount_avg_24h',
            'transactions.es_transportation_count_14d',
            'transactions.es_health_count_14d',
            'transactions.es_otherservices_count_14d',
            'transactions.es_food_count_14d',
            'transactions.es_hotelservices_count_14d',
            'transactions.es_barsandrestaurants_count_14d',
            'transactions.es_tech_count_14d',
            'transactions.es_sportsandtoys_count_14d',
            'transactions.es_wellnessandbeauty_count_14d',
            'transactions.es_hyper_count_14d',
            'transactions.es_fashion_count_14d',
            'transactions.es_home_count_14d',
            'transactions.es_travel_count_14d',
            'transactions.es_leisure_count_14d',
            'transactions.gender_F',
            'transactions.gender_M',
            'transactions.step',
            'transactions.amount',
            'transactions.timestamp_hour',
            'transactions.timestamp_day_of_week',
            'events.*']
```

```
# Import MLRun's Feature Store
import mlrun.feature_store as fstore

# Define the feature vector name for future reference
fv_name = 'transactions-fraud'

# Define the feature vector using our Feature Store (fstore)
transactions_fv = fstore.FeatureVector(fv_name,
                          features,
                          label_feature="labels.label",
                          description='Predicting a fraudulent transaction')
```

(continues on next page)

(continued from previous page)

```
# Save the feature vector in the Feature Store
transactions_fv.save()
```

Step 2 - Preview the feature vector data

Obtain the values of the features in the feature vector, to ensure the data appears as expected.

```
# Import the Parquet Target so you can directly save the dataset as a file
from mlrun.datastore.targets import ParquetTarget

# Get offline feature vector as dataframe and save the dataset to parquet
train_dataset = fstore.get_offline_features(fv_name, target=ParquetTarget())
```

```
> 2021-09-19 17:59:28,415 [info] wrote target: {'name': 'parquet', 'kind': 'parquet',
↪'path': 'v3io:///projects/fraud-demo-admin/FeatureStore/transactions-fraud/parquet/
↪vectors/transactions-fraud-latest.parquet', 'status': 'ready', 'updated': '2021-09-
↪19T17:59:28.415727+00:00', 'size': 1915182}
```

```
# Preview the dataset
train_dataset.to_dataframe().tail(5)
```

```
       amount_max_2h  amount_sum_2h  amount_count_2h  amount_avg_2h  \
49995           2.95           2.95              1.0          2.950
49996          37.40          37.40              1.0         37.400
49997           7.75           7.75              1.0          7.750
49998          28.89          28.89              1.0         28.890
49999          78.18         105.43              2.0         52.715

       amount_max_12h  amount_sum_12h  amount_count_12h  amount_avg_12h  \
49995            2.95            2.95               1.0           2.950
49996           37.40           37.40               1.0          37.400
49997            7.75           12.99               2.0           6.495
49998           38.35          107.76               4.0          26.940
49999           78.18          153.78               3.0          51.260

       amount_max_24h  amount_sum_24h  ...  gender_F  gender_M  step  amount  \
49995            2.95            2.95  ...         1         0    41    2.95
49996           37.40           37.40  ...         0         1    40   37.40
49997           61.23          112.76  ...         1         0    91    7.75
49998           52.97          249.41  ...         0         1    56   28.89
49999           78.18          220.19  ...         1         0    76   78.18

       timestamp_hour  timestamp_day_of_week  event_details_change  \
49995              17                      6                   0.0
49996              17                      6                   0.0
49997              17                      6                   1.0
49998              17                      6                   1.0
49999              17                      6                   1.0

       event_login  event_password_change  label
49995          0.0                    1.0      0
49996          0.0                    1.0      0
49997          0.0                    0.0      0
```

(continues on next page)

(continued from previous page)

```
49998          0.0                    0.0       0
49999          0.0                    0.0       0

[5 rows x 36 columns]
```

Step 3 - Train models and choose highest accuracy

With MLRun, one can easily train different models and compare the results. The code below trains three different models, and chooses the model with the highest accuracy. Each uses a different algorithm (random forest, XGBoost, adabost).

```
# Import the Sklearn classifier function from the functions hub
classifier_fn = mlrun.import_function('hub://sklearn-classifier')
```

```
# Prepare the parameters list for the training function
# Use 3 different models
training_params = {"model_name": ['transaction_fraud_rf',
                                  'transaction_fraud_xgboost',
                                  'transaction_fraud_adaboost'],

              "model_pkg_class": ['sklearn.ensemble.RandomForestClassifier',
                                  'sklearn.ensemble.GradientBoostingClassifier',
                                  'sklearn.ensemble.AdaBoostClassifier']}

# Define the training task, including the feature vector, label and hyperparams␣
↪definitions
train_task = mlrun.new_task('training',
                     inputs={'dataset': transactions_fv.uri},
                     params={'label_column': 'label'}
                    )

train_task.with_hyper_params(training_params, strategy='list', selector='max.accuracy
↪')

# Specify the cluster image
classifier_fn.spec.image = 'mlrun/mlrun'

# Run training
classifier_fn.run(train_task, local=False)
```

```
> 2021-09-19 17:59:28,799 [info] starting run training␣
↪uid=9349c60dd9f24a33b536c59e89978e7b DB=http://mlrun-api:8080
> 2021-09-19 17:59:29,042 [info] Job is running in the background, pod: training-2jntc
> 2021-09-19 17:59:47,926 [info] best iteration=1, used criteria max.accuracy
> 2021-09-19 17:59:48,990 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 17:59:51,574 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f464baf0c50>
```

Step 4 - Perform feature selection

As part of our data science process, try to reduce the training dataset's size to get rid of bad or unuseful features and save computation time.

Use the ready-made feature selection function from the hub `hub://feature_selection` to select the best features to keep on a sample from the dataset, and run the function on that.

```
feature_selection_fn = mlrun.import_function('hub://feature_selection')

feature_selection_run = feature_selection_fn.run(
            params={'sample_ratio':0.25,
                    'output_vector_name':fv_name + "-short",
                   'ignore_type_errors': True},

            inputs={'df_artifact': transactions_fv.uri},
            name='feature_extraction',
            handler='feature_selection',
    local=False)
```

```
> 2021-09-19 17:59:51,768 [info] starting run feature_extraction␣
↪uid=3a50bd0e4175459fb53873d8f78a440a DB=http://mlrun-api:8080
> 2021-09-19 17:59:52,004 [info] Job is running in the background, pod: feature-
↪extraction-lf46d
> 2021-09-19 17:59:59,099 [info] Couldn't calculate chi2 because of: Input X must be␣
↪non-negative.
> 2021-09-19 18:00:04,008 [info] votes needed to be selected: 3
> 2021-09-19 18:00:05,329 [info] wrote target: {'name': 'parquet', 'kind': 'parquet',
↪'path': 'v3io:///projects/fraud-demo-admin/FeatureStore/transactions-fraud-short/
↪parquet/vectors/transactions-fraud-short-latest.parquet', 'status': 'ready',
↪'updated': '2021-09-19T18:00:05.329695+00:00', 'size': 668722}
> 2021-09-19 18:00:05,677 [info] run executed, status=completed
Pass k=5 as keyword args. From version 0.25 passing these as positional arguments␣
↪will result in an error
Liblinear failed to converge, increase the number of iterations.
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 18:00:07,537 [info] run executed, status=completed
```

```
mlrun.get_dataitem(feature_selection_run.outputs['top_features_vector']).as_df().
↪tail(5)
```

```
       amount_max_2h  amount_sum_2h  amount_count_2h  amount_avg_2h  \
49996          37.40          37.40              1.0      37.400000
49997           7.75           7.75              1.0       7.750000
49998          28.89          28.89              1.0      28.890000
49999          78.18         105.43              2.0      52.715000
50000          19.37          24.61              3.0       8.203333

       amount_max_12h  label
49996           37.40      0
49997            7.75      0
49998           38.35      0
49999           78.18      0
50000           19.37      0
```

Step 5 - Train the models with top features

Following the feature selection, you train new models using the resultant features. You can observe the accuracy and other results remain high, meaning you get a model that requires less features to be accurate and thus less error-prone.

```
# Defining our training task, including our feature vector, label and hyperparams␣
↪definitions
ensemble_train_task = mlrun.new_task('training',
                     inputs={'dataset': feature_selection_run.outputs['top_features_
↪vector']},
                     params={'label_column': 'label'}
                    )
ensemble_train_task.with_hyper_params(training_params, strategy='list', selector='max.
↪accuracy')

classifier_fn.run(ensemble_train_task)
```

```
> 2021-09-19 18:00:07,661 [info] starting run training␣
↪uid=a6d9ae72cfd3462cace205f8b363d214 DB=http://mlrun-api:8080
> 2021-09-19 18:00:08,077 [info] Job is running in the background, pod: training-v2bt4
> 2021-09-19 18:00:20,781 [info] best iteration=3, used criteria max.accuracy
> 2021-09-19 18:00:21,696 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 18:00:27,561 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f464baed490>
```

Done!

You've completed Part 2 of the model training with the feature store. Proceed to *Part 3* to learn how to deploy and monitor the model.

Part 3: Serving

In this part you use MLRun's **serving runtime** to deploy the trained models from the previous stage a `Voting Ensemble` using **max vote** logic.You will also use MLRun's **Feature store** to receive the latest tag of the online **Feature Vector** we defined in the preveious stage.

By the end of this tutorial you'll learn how to:

- Define a model class to load the models, run preprocessing, and predict on the data
- Define Voting Ensemble function on top of our models
- Test the serving function locally using the `mock server`
- Deploy the function to the cluster and test it live

Environment setup

First, make sure SciKit-Learn is installed in the correct version:

```
!pip install -U scikit-learn==1.0.2
```

Restart your kernel post installing. Secondly, since the work is done in this project scope, define the project itself for all your MLRun work in this notebook.

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-10-28 11:59:01,033 [info] loaded project fraud-demo from MLRun DB
```

Define model class

- Load models
- Predict from the FS Online service via the `source` key

```
# mlrun: start-code
```

```
import numpy as np
from cloudpickle import load
from mlrun.serving.v2_serving import V2ModelServer

class ClassifierModel(V2ModelServer):

    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        print(f"Input -> {body['inputs']}")
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

```
# mlrun: end-code
```

Define a serving function

MLRun serving can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. Nuclio is a high-performance open-source serverless framework that's focused on data, I/O, and compute-intensive workloads.

The **EnrichmentVotingEnsemble** and the **EnrichmentModelRouter** router classes auto enrich the request with data from the feature store. The router input accepts lists of inference request (each request can be a dict or list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).

In many cases the features can have null values (None, NaN, Inf, ..). The `Enrichment` routers can substitute the null value with fixed or statistical value per feature. This is done through the `impute_policy` parameter, which accepts the impute policy per feature (where `*` is used to specify the default). The value can be a fixed number for constants or `$mean`, `$max`, `$min`, `$std`, `$count` for statistical values. to substitute the value with the equivalent feature stats (taken from the feature store).

The code below performs the following steps:

- Gather ClassifierModel code from this notebook
- Define `EnrichmentVotingEnsemble` - Max-Vote based ensemble with feature enrichment and imputing
- Add the previously trained models to the ensemble

```
# Create the serving function from the code above
serving_fn = mlrun.code_to_function('transaction-fraud', kind='serving', image="mlrun/
↪mlrun")

serving_fn.set_topology('router', 'mlrun.serving.routers.EnrichmentVotingEnsemble',␣
↪name='VotingEnsemble',
                        feature_vector_uri="transactions-fraud-short", impute_policy={
↪"*": "$mean"})

model_names = [
'RandomForestClassifier',
```

(continues on next page)

(continued from previous page)

```
'GradientBoostingClassifier',
'AdaBoostClassifier'
]

for i, name in enumerate(model_names, start=1):
    serving_fn.add_model(name, class_name="ClassifierModel", model_path=project.get_
↪artifact_uri(f"training_model#{i}:latest"))

# Plot the ensemble configuration
serving_fn.spec.graph.plot()
```

```
<graphviz.dot.Digraph at 0x7f5af5d471d0>
```

Test the server locally

Before deploying the serving function, test it in the current notebook and check the model output.

```
# Create a mock server from the serving function
local_server = serving_fn.to_mock_server()
```

```
> 2021-10-28 11:59:11,260 [info] model RandomForestClassifier was loaded
> 2021-10-28 11:59:11,306 [info] model GradientBoostingClassifier was loaded
> 2021-10-28 11:59:11,350 [info] model AdaBoostClassifier was loaded
```

```
# Choose an id for the test
sample_id = 'C76780537'

model_inference_path = '/v2/models/infer'

# Send our sample ID for prediction
local_server.test(path=model_inference_path,
            body={'inputs': [[sample_id]]})

# Notice the input vector is printed 3 times (once per child model) and is enriched␣
↪with data from the feature store
```

```
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
```

```
{'id': '757c736c985a4c42b3ebd58f3c50f1b2',
 'model_name': 'VotingEnsemble',
 'outputs': [0],
 'model_version': 'v1'}
```

Accessing the real-time feature vector directly

You can also directly query the feature store values using the `get_online_feature_service` method. This method is used internally in the EnrichmentVotingEnsemble router class

```
import mlrun.feature_store as fstore

# Create the online feature service
svc = fstore.get_online_feature_service('transactions-fraud-short:latest', impute_
↪policy={"*": "$mean"})

# Get sample feature vector
sample_fv = svc.get([{'source': sample_id}])
sample_fv
```

```
[{'amount_max_2h': 14.68,
  'amount_max_12h': 70.81,
  'amount_sum_2h': 14.68,
  'amount_count_2h': 1.0,
  'amount_avg_2h': 14.68}]
```

Deploying the function on the kubernetes cluster

You can now deploy the function. Once it's deployed you get a function with an http trigger that can be called from other locations.

```
import os

# Enable model monitoring
serving_fn.set_tracking()
project.set_model_monitoring_credentials(os.getenv('V3IO_ACCESS_KEY'))

# Deploy the serving function
serving_fn.deploy()
```

```
> 2021-10-28 11:59:17,554 [info] Starting remote function deploy
2021-10-28 11:59:17  (info) Deploying function
2021-10-28 11:59:17  (info) Building
2021-10-28 11:59:17  (info) Staging files and preparing base images
2021-10-28 11:59:17  (info) Building processor image
2021-10-28 11:59:19  (info) Build complete
2021-10-28 11:59:25  (info) Function deploy complete
> 2021-10-28 11:59:25,657 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.
↪cluster.local:8080'], 'external_invocation_urls': ['default-tenant.app.yh38.iguazio-
↪cd2.com:32287']}
```

```
'http://default-tenant.app.yh38.iguazio-cd2.com:32287'
```

Test the server

Test the serving function and examine the model output.

```
# Choose an id for the test
sample_id = 'C76780537'

model_inference_path = '/v2/models/infer'

# Send the sample ID for prediction
serving_fn.invoke(path=model_inference_path,
                  body={'inputs': [[sample_id]]})
```

```
> 2021-10-28 11:59:25,722 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
```

```
{'id': '4b9c4914-964f-4bd5-903d-c4885ed7c090',
 'model_name': 'VotingEnsemble',
 'outputs': [0],
 'model_version': 'v1'}
```

You can also directly query the feature store values, which are used in the enrichment.

Simulate incoming data

```
# Load the dataset
data = mlrun.get_dataitem('https://s3.wasabisys.com/iguazio/data/fraud-demo-mlrun-fs-
↪docs/data.csv').as_df()

# Sample 50k lines
data = data.sample(50000)

# keys
sample_ids = data['source'].to_list()
```

```
from random import choice, uniform
from time import sleep

# Sending random requests
for _ in range(4000):
    data_point = choice(sample_ids)
    try:
        resp = serving_fn.invoke(path=model_inference_path, body={'inputs': [[data_
↪point]]})
        print(resp)
        sleep(uniform(0.2, 1.7))
    except OSError:
        pass
```

```
> 2021-10-28 12:00:23,079 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': '6b813638-e9ef-4e92-85c8-cfbd0b74fe32', 'model_name': 'VotingEnsemble',
↪'outputs': [0], 'model_version': 'v1'}
```

(continues on next page)

(continued from previous page)

```
> 2021-10-28 12:00:23,857 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': 'f84bf2ec-a718-4e90-a7d5-fe08e254f3c8', 'model_name': 'VotingEnsemble',
↪'outputs': [0], 'model_version': 'v1'}
> 2021-10-28 12:00:24,545 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': '7bb023f7-edbc-47a6-937b-4a15c8380b74', 'model_name': 'VotingEnsemble',
↪'outputs': [0], 'model_version': 'v1'}
> 2021-10-28 12:00:24,921 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': '57882cca-537a-43e1-9986-1bbc72fb84b7', 'model_name': 'VotingEnsemble',
↪'outputs': [0], 'model_version': 'v1'}
```

Part 4: Automated ML pipeline

MLRun Project is a container for all your work on a particular activity: sll the associated code, functions, jobs/workflows and artifacts. Projects can be mapped to `git` repositories which enables versioning, collaboration, and CI/CD. Users can create project definitions using the SDK or a yaml file and store those in MLRun DB, file, or archive. Once the project is loaded you can run jobs/workflows which refer to any project element by name, allowing separation between configuration and code.

Projects contain `workflows` that execute the registered functions in a sequence/graph (DAG). It can reference project parameters, secrets and artifacts by name. The following notebook demonstrate how to build an automated workflow with **feature selection**, **training**, **testing**, and **deployment**.

Step 1: Setting up your project

To run a pipeline, you first need to get or create a project object and define/import the required functions for its execution. See Create and load projects for details.

The following code gets or creates a user project named "fraud-demo-<username>".

```
# Set the base project name
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-10-28 13:54:45,892 [info] loaded project fraud-demo from MLRun DB
```

Step 2: Updating project and function definitions

You need to save the definitions for the function you use in the projects so it is possible to automatically convert code to functions or import external functions whenever you load new versions of the code or when you run automated CI/CD workflows. In addition you may want to set other project attributes such as global parameters, secrets, and data.

The code can be stored in Python files, notebooks, external repositories, packaged containers, etc. Use the `project.set_function()` method to register the code in the project. The definitions are saved to the project object, as well as in a YAML file in the root of our project. Functions can also be imported from MLRun marketplace (using the `hub://` schema).

You used the following functions in this tutorial:

- `feature_selection` — the first function, which determines the top features to be used for training
- `train` — the model-training function
- `test-classifier` — the model-testing function
- `mlrun-model` — the model-serving function

```{admonition} Note: `set_function` uses the `code_to_function` and `import_function` methods under the hood (used in the previous notebooks), but in addition it saves the function configurations in the project spec for use in automated workflows and CI/CD.

```
Add the function definitions to the project along with parameters and data artifacts
↪and save the project.
```

```
project.set_function('hub://feature_selection', 'feature_selection')
project.set_function('hub://sklearn-classifier','train')
project.set_function('hub://test_classifier', 'test')
project.set_function('hub://v2_model_server', 'serving')
```

```
<mlrun.runtimes.serving.ServingRuntime at 0x7f6229497190>
```

```
# set project level parameters and save
project.spec.params = {'label_column': 'label'}
project.save()
```

When you save the project it stores the project definitions in the `project.yaml`. This means that you can load the project from the source control (GIT) and run it with a single command or API call.

The project YAML for this project can be printed using:

```
print(project.to_yaml())
```

```
kind: project
metadata:
  name: fraud-demo-admin
  created: '2021-08-05T15:59:59.434655'
spec:
  params:
    label_column: label
  functions:
```

(continues on next page)
```

(continued from previous page)

```
  - url: hub://feature_selection
    name: feature_selection
  - url: hub://sklearn-classifier
    name: train
  - url: hub://test_classifier
    name: test
  - url: hub://v2_model_server
    name: serving
  workflows:
  - name: main
    path: workflow.py
    engine: null
  artifacts: []
  desired_state: online
  disable_auto_mount: false
status:
  state: online
```

Saving and loading projects from GIT

After you save the project and its elements (functions, workflows, artifacts, etc.) you can commit all the changes to a GIT repository. Do this using standard GIT tools or using MLRun `project` methods such as `pull`, `push`, `remote` that call the Git API for you.

Projects can then be loaded from Git using MLRun `load_project` method, for example:

```
project = mlrun.load_project("./myproj", "git://github.com/mlrun/project-demo.git",␣
↪name=project_name)
```

or using MLRun CLI:

```
mlrun project -n myproj -u "git://github.com/mlrun/project-demo.git" ./myproj
```

Read Create and load projects for more details.

Using Kubeflow pipelines

You're now ready to create a full ML pipeline. This is done by using Kubeflow Pipelines — an open-source framework for building and deploying portable, scalable, machine-learning workflows based on Docker containers. MLRun leverages this framework to take your existing code and deploy it as steps in the pipeline.

Step 3: Defining and saving a pipeline workflow

A pipeline is created by running an MLRun **"workflow"**. The following code defines a workflow and writes it to a file in your local directory; (the file name is **workflow.py**). The workflow describes a directed acyclic graph (DAG) for execution using Kubeflow Pipelines, and depicts the connections between the functions and the data as part of an end-to-end pipeline. The workflow file has a definition of a pipeline DSL for connecting the function inputs and outputs.

The defined pipeline includes the following steps:

- Perform feature selection (`feature_selection`).
- Train and the model (`train`).
- Test the model with its test data set.
- Deploy the model as a real-time serverless function (`deploy`).

  Note: A pipeline can also include continuous build integration and deployment (CI/CD) steps, such as building container images and deploying models.

```
%%writefile workflow.py
import mlrun
from kfp import dsl
from mlrun.model import HyperParamOptions

from mlrun import (
    build_function,
    deploy_function,
    import_function,
    run_function,
)


@dsl.pipeline(
    name="Fraud Detection Pipeline",
    description="Detecting fraud from a transactions dataset"
)

def kfpipeline(vector_name='transactions-fraud'):

    project = mlrun.get_current_project()

    # Feature selection
    feature_selection = run_function(
        "feature_selection",
        name="feature_selection",
        params={'sample_ratio':0.25,'output_vector_name': "short",
                'ignore_type_errors': True},
        inputs={'df_artifact': project.get_artifact_uri(vector_name, 'feature-vector
↪')},
        outputs=['top_features_vector'])
```

(continues on next page)

(continued from previous page)

```
    # train with hyper-paremeters
    train = run_function(
        "train",
        name="train",
        params={"sample": -1,
                "label_column": project.get_param('label_column', 'label'),
                "test_size": 0.10},
        hyperparams={"model_name": ['transaction_fraud_rf',
                                    'transaction_fraud_xgboost',
                                    'transaction_fraud_adaboost'],

                     'model_pkg_class': ["sklearn.ensemble.RandomForestClassifier",
                                         "sklearn.linear_model.LogisticRegression",
                                         "sklearn.ensemble.AdaBoostClassifier"]},
        hyper_param_options=HyperParamOptions(selector="max.accuracy"),
        inputs={"dataset": feature_selection.outputs['top_features_vector']},
        outputs=['model', 'test_set'])


    # test and visualize our model
    test = run_function(
        "test",
        name="test",
        params={"label_column": project.get_param('label_column', 'label')},
        inputs={
            "models_path": train.outputs["model"],
            "test_set": train.outputs["test_set"]})

    # route the serving model to use enrichment
    funcs['serving'].set_topology('router',
                                  'mlrun.serving.routers.EnrichmentModelRouter',
                                  name='EnrichmentModelRouter',
                                  feature_vector_uri="transactions-fraud-short",
                                  impute_policy={"*": "$mean"},
                                  exist_ok=True)


    # deploy the model as a serverless function, you can pass a list of models to␣
↪serve
    deploy = deploy_function("serving", models=[{"key": 'fraud', "model_path": train.
↪outputs["model"]}])
```

```
Overwriting workflow.py
```

Step 4: Registering the workflow

Use the `set_workflow` MLRun project method to register your workflow with MLRun. The following code sets the `name` parameter to the selected workflow name ("main") and the `code` parameter to the name of the workflow file that is found in your project directory (**workflow.py**).

```
# Register the workflow file as "main"
project.set_workflow('main', 'workflow.py')
```

Step 5: Running a pipeline

First run the following code to save your project:

```
project.save()
```

Use the `run` MLRun project method to execute your workflow pipeline with Kubeflow Pipelines.

You can pass **`arguments`** or set the **`artifact_path`** to specify a unique path for storing the workflow artifacts.

```
run_id = project.run(
    'main',
    arguments={},
    dirty=True, watch=True)
```

```
<graphviz.dot.Digraph at 0x7f6234c28b50>
```

```
<IPython.core.display.HTML object>
```

Step 6: Test the model end point

Now that your model is deployed using the pipeline, you can invoke it as usual:

```
# Define the serving function
serving_fn = project.func('serving')

# Choose an id for the test
sample_id = 'C76780537'
model_inference_path = '/v2/models/fraud/infer'

# Send the sample ID for predcition
serving_fn.invoke(path=model_inference_path,
                  body={'inputs': [[sample_id]]})
```

```
> 2021-10-28 13:56:56,170 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-fraud-demo-admin-v2-model-server.default-tenant.svc.cluster.local:8080/v2/
↪models/fraud/infer'}
```

```
{'id': '90f4b67c-c9e0-4e35-917f-979b71c5ad75',
 'model_name': 'fraud',
 'outputs': [0.0]}
```

Done!

2.1.15 Creating and using functions

In this section

Configuring Functions

MLRun Functions (function objects) can be created by using any of the following methods:

- ***new_function()***: creates a function from code repository/archive.
- ***code_to_function()***: creates a function from local or remote source code (single file) or from a notebook (code file will be embedded in the function object).
- ***import_function()***: imports a function from a local or remote YAML function-configuration file or from a function object in the MLRun database (using a DB address of the format `db://<project>/<name>[:<tag>]`) or from the function marketplace (e.g. `hub://describe`). See *MLRun Functions Marketplace*.

When you create a function, you can:

- Use the *save()* function method to save a function object in the MLRun database.
- Use the *export()* method to save a YAML function-configuration to your preferred local or remote location.
- Use the *run()* method to execute a task.
- Use the *as_step()* method to convert a function to a Kubeflow pipeline step.
- Use the `.deploy()` method to build/deploy the function. (Deploy for batch functions builds the image and adds the required packages. For online/real-time runtimes like `nuclio` and `serving` it also deploys it as an online service.)

Functions are stored in the project and are versioned so you can always view previous code and go back to previous functions if needed.

The general concepts described in this section are illustrated in the following figure:

In this section

- *Providing Function Code*
- *Specifying the function's execution handler or command*
- *Function Runtimes*

Providing Function Code

When using `code_to_function()` or `new_function()`, you can provide code in several ways:

- *As part of the function object*
- *As part of the function image*
- *From the git/zip/tar archive into the function at runtime*

Provide code as part of the function object

This method is great for small and single file functions or for using code derived from notebooks. This example uses the mlrun *code_to_function()* method to create functions from code files or notebooks.

```
# create a function from py or notebook (ipynb) file, specify the default function␣
↪handler
my_func = mlrun.code_to_function(name='prep_data', filename='./prep_data.py', kind=
↪'job',
image='mlrun/mlrun', handler='my_func')
```

For more on how to create functions from notebook code, see *Converting notebook code to a function*.

Provide code as part of the function image

Providing code as part of the image is good for ensuring that the function image has the integrated code and dependencies, and it avoids the overhead of loading code at runtime.

Use the *deploy()* method to build a function image with source code, dependencies, etc. Specify the build configuration using the *build_config()* method.

```
    # create a new job function from base image and archive + custom build commands
    fn = mlrun.new_function('archive', kind='job', command='./myfunc.py')
    fn.build_config(base_image='mlrun/mlrun', source='git://github.com/org/repo.git
↪#master',
                    commands=["pip install pandas"])
    # deploy (build the container with the extra build commands/packages)
    fn.deploy()

    # run the function (specify the function handler to execute)
    run_results = fn.run(handler='my_func', params={"x": 100})
```

Alternatively, you can use a pre-built image:

```
# provide a pre-built image with your code and dependencies
fn = mlrun.new_function('archive', kind='job', command='./myfunc.py', image='some/pre-
↪built-image:tag')

# run the function (specify the function handler to execute)
run_results = fn.run(handler='my_func', params={"x": 100})
```

You can use this option with *new_function()* method.

Provide code from a git, zip, tar archive into the function at runtime

This option is the most efficient when doing iterative development with multiple code files and packages. You can make small code changes and re-run the job without building images, etc. You can use this option with the *new_function()* method.

The `local`, `job`, `mpijob` and `remote-spark` runtimes support dynamic load from archive or file shares. (Other runtimes will be added later.) Enable this by setting the `spec.build.source=<archive>` and `spec.build.load_source_on_run=True` or simply by setting the `source` attribute in `new_function`). In the CLI, use the `--source` flag.

```
fn = mlrun.new_function('archive', kind='job', image='mlrun/mlrun', command='./myfunc.
↪py',
                        source='git://github.com/mlrun/ci-demo.git#master')
run_results = fn.run(handler='my_func', params={"x": 100})
```

See more details and examples on *running jobs with code from Archives or shares*.

Specifying the function execution handler or command

The function is configured with code and dependencies, however you also need to set the main execution code either by handler or command.

Handler

A handler is a method (not a script) that executes the function, for either a one-time run or ongoing online services.

Command

The `command='./myfunc.py'` specifies the command that is executed in the function container/workdir.

By default MLRun tries to execute python code with the specified command. For executing non-python code, set `mode="pass"` (passthrough) and specify the full execution `command`, e.g.:

```
new_function(... command="bash main.sh --myarg xx", mode="pass")
```

If you need to add arguments in the command, use `"mode=args"` template (`{..}`) in the command to pass the task parameters as arguments for the execution command, for example:

```
new_function(... command='mycode.py' --x {xparam}", mode="args")
```

where `{xparam}` is substituted with the value of the `xparam` parameter. It is possible to use argument templates also when using `mode="pass"`.

See also Execute non Python code and Inject parameters into command line.

Converting Notebook Code to a Function

MLRun annotations are used to identify the code that needs to be converted into an MLRun function. They provide non-intrusive hints which indicate which parts of your notebook should be considered as the code of the function.

Annotations start a code block using `# mlrun: start-code` and end a code block(s) , with `# mlrun: end-code`. Use the `#mlrun: ignore` to exclude items from the code qualified annotations. Make sure that the annotations include anything required for the function to run.

```
# mlrun: start-code

def sub_handler():
    return "hello world"
```

The `# mlrun: ignore` annotation enables you to exclude the cell from the function code.

```
# mlrun: ignore

# the handler in the code section below will not call this sub_handler
def sub_handler():
    return "I will be ignored!"
```

```
def handler(context, event):
    return sub_handler()

# mlrun: end-code
```

Convert the function with `mlrun.code_to_function` and run the handler. Note the returned value under `results`.

```
from mlrun import code_to_function

some_function = code_to_function('some-function-name', kind='job', code_output='.')
some_function.run(name='some-function-name', handler='handler', local=True)
```

```
> 2021-11-01 07:42:44,930 [info] starting run some-function-name␣
↪uid=742e7d6e930c48f3a2f1d6175e971455 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:42:45,214 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9ed81d0>
```

In this section

Named annotations

The `# mlrun: start-code` and `# mlrun: end-code` annotations can be used to convert different code sections to different MLRun ,functions in the same notebook. To do so add the name of the MLRun function to the end of the annotation as shown in the example below.

```
# mlrun: start-code my-function-name

def handler(context, event):
    return "hello from my-function"

# mlrun: end-code my-function-name
```

Convert the function and run the handler. Notice that the handler that is being used and that there is a change in the returned value under `results`.

```
my_function = code_to_function('my-function-name', kind='job')
my_function.run(name='my-function-name', handler='handler', local=True)
```

```
> 2021-11-01 07:42:53,892 [info] starting run my-function-name uid=e4bbc3cae21042439cc1c3cb9631751c DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:42:54,137 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9ac71d0>
```

Note

Make sure to use the name given to the `code_to_function` parameter (`name='my-function-name'` in the example above) so that all relevant `start-code` and `end-code` annotations are included. If none of the annotations are marked with the function's name, all annotations without any name are used.

Multi section function

You can use the `# mlrun: start-code` and `# mlrun: end-code` annotations multiple times in a notebook since the whole notebook is scanned. The annotations can be named like the following example, and they can be nameless. If you choose nameless, remember all nameless annotations in the notebook are used.

```
# mlrun: start-code multi-section-function-name

function_name = "multi-section-function-name"

# mlrun: end-code multi-section-function-name
```

Any code between those sections are not included:

```
function_name = "I will be ignored!"
```

```
# mlrun: start-code multi-section-function-name
```

```
def handler(context, event):
    return f"hello from {function_name}"
```

```
# mlrun: end-code multi-section-function-name
```

```
my_multi_section_function = code_to_function('multi-section-function-name', kind='job')
my_multi_section_function.run(name='multi-section-function-name', handler='handler', local=True)
```

```
> 2021-11-01 07:43:05,587 [info] starting run multi-section-function-name uid=9ac6a0e977a54980b657bae067c2242a DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:43:05,834 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9a24e10>
```

Annotation's position in code cell

`# mlrun: start-code` and `# mlrun: end-code` annotations are relative to their positions inside the code block. Notice how the assignments to `function_name` below `# mlrun: end-code` don't override the assignment between the annotations in the function's context.

```
# mlrun: start-code part-cell-function

def handler(context, event):
    return f"hello from {function_name}"

function_name = "part-cell-function"

# mlrun: end-code part-cell-function

function_name = "I will be ignored"
```

```
my_multi_section_function = code_to_function('part-cell-function', kind='job')
my_multi_section_function.run(name='part-cell-function', handler='handler',␣
↪local=True)
```

```
> 2021-11-01 07:43:14,347 [info] starting run part-cell-function␣
↪uid=5426e665c7bc4ba492e0a704c5555fb6 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:43:14,628 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9a2bf50>
```

Guidelines

- Make sure that every `# mlrun: start-code` has a corresponding `# mlrun: end-code` before the next `# mlrun: start-code` in the notebook.
- Only one MLRun function can have a nameless annotation per notebook.
- Do not use multiple `# mlrun: start-code` nor multiple `# mlrun: end-code` annotations in a single code cell. Only the first appearance of each is used.
- Using single annotations:
  - Use a `# mlrun: start-code` alone, and all code blocks from the annotation to the end of the notebook are included.
  - Use a `# mlrun: end-code` alone, and all code blocks from the beginning of the notebook to the annotation are included.

Using code from archives or file shares

In this section

Archive URL options

- Git e.g. git://github.com/mlrun/something.git#master
- Zip/Tar archives e.g. https://github.com/mlrun/mlrun/raw/run-cli/examples/archive.zip
- File share e.g. `/User/mycode` (requires adding a file share to the function)

The archive is set as the working dir for the function and the file/params to execute should be set using the `command` parameter (with the relative path inside the archive).

Run from zip using the CLI

```
!python -m mlrun run --name tst1 --watch --source https://github.com/mlrun/mlrun/raw/
↪development/examples/archive.zip --handler handler --image mlrun/mlrun myfunc.py
```

```
> 2021-06-15 11:52:45,847 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-06-15 11:52:48,460 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': '0.6.4', 'client_version': 'unstable'}
> 2021-06-15 11:52:48,469 [info] starting run tst1␣
↪uid=ce4a3eab42ff43e0885d82ec27762949 DB=http://mlrun-api:8080
> 2021-06-15 11:52:48,612 [info] Job is running in the background, pod: tst1-6zpsv
> 2021-06-15 11:52:51,885 [info] extracting source from https://github.com/mlrun/
↪mlrun/raw/development/examples/archive.zip to /mlrun/code
Run: tst1 (uid=ce4a3eab42ff43e0885d82ec27762949)
my line
got text: some text
> 2021-06-15 11:52:51,957 [info] run executed, status=completed
final state: completed
> 2021-06-15 11:52:54,715 [info] run executed, status=completed
```

Using code from Git

```
import mlrun
from mlrun.platforms import auto_mount
fn = mlrun.new_function('archive', kind='job', image='mlrun/mlrun', command='./myfunc.
↪py',
                        source='git://github.com/mlrun/ci-demo.git#master')
run = fn.run(handler='my_func')
```

```
> 2021-06-15 11:58:59,002 [info] starting run archive-my_func␣
↪uid=0b6195fbf1844880a829d61505bd9a38 DB=http://mlrun-api:8080
> 2021-06-15 11:58:59,468 [info] Job is running in the background, pod: archive-my-
↪func-8frkp
> 2021-06-15 11:59:02,726 [info] extracting source from git://github.com/mlrun/ci-
↪demo.git#master to /mlrun/code
Run: archive-my_func (uid=0b6195fbf1844880a829d61505bd9a38)
Params: p1=1, p2=a-string
> 2021-06-15 11:59:02,764 [info] running function
> 2021-06-15 11:59:02,797 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 0b6195fbf1844880a829d61505bd9a38 --project default , !mlrun logs␣
↪0b6195fbf1844880a829d61505bd9a38 --project default
> 2021-06-15 11:59:05,633 [info] run executed, status=completed
```

Using code from file share

```
import mlrun
fn = mlrun.new_function('archive', kind='job', image='mlrun/mlrun', command='./code.py
↪',
                        source='/User/sample2')
# add shared volume mount so the function will have access to the mounted code
fn.apply(auto_mount())
run = fn.run(handler='handler')
```

```
> 2021-04-27 13:08:58,586 [info] starting run archive-handler␣
↪uid=a09a42808aff4551b2ba29c701f78395 DB=http://mlrun-api:8080
> 2021-04-27 13:08:58,801 [info] Job is running in the background, pod: archive-
↪handler-74crq
extracting source from /User/sample2 to ./
cwd=/mlrun, workdir=None
Run: archive-handler (uid=a09a42808aff4551b2ba29c701f78395)
my line, bla bla
> 2021-04-27 13:09:05,095 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run a09a42808aff4551b2ba29c701f78395 --project default , !mlrun logs␣
↪a09a42808aff4551b2ba29c701f78395 --project default
> 2021-04-27 13:09:08,022 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7eff90ab77d0>
```

Inject parameters into command line

The function `command` parameter is the command that executes inside the container (the archive path is set as the working directory). You can pass arguments in the command and also inject the job/task parameters into the command at runtime (by using `{}` around the parameter).

For example, define a function with the command template and pass a parameter during the run:

```
fn = mlrun.new_function('withargs', kind='job', image='mlrun/mlrun', command="main.py␣
↪--myarg {myarg}",
                        source='git://github.com/org/repo')
run = fn.run(params={'myarg': 'xx'})
```

Execute non-Python code

By default MLRun tries to execute Python code. You can run any other code by specifying the **Pass** (passthrough) mode (`mode="pass"`). In the `pass` mode the `command` is used as is, for example:

```
fn = mlrun.new_function('withargs', kind='job', image='mlrun/mlrun', command="bash␣
↪main.sh --myarg {myarg}",
                        source='git://github.com/org/repo', mode='pass')
run = fn.run(params={'myarg': 'xx'})
```

Attach storage to functions

In the vast majority of cases, an MLRun function requires access to storage. This storage might be used to provide inputs to the function including data-sets to process or data-streams that contain input events. Typically, storage is used to store function outputs and result artifacts. For example, trained models or processed data-sets.

Since MLRun functions can be distributed and executed in Kubernetes pods, the storage used would typically be shared, and execution pods would need some added configuration options applied to them so that the function code is able to access the designated storage. These configurations might be k8s volume mounts, specific environment variables that contain configuration and credentials, and other configuration of security settings. These storage configurations are not applicable to functions running locally in the development environment, since they are executed in the local context.

The common types of shared storage are:

1. `v3io` storage through API — When running as part of the Iguazio system, MLRun has access to the system's v3io storage through paths such as `v3io:///projects/my_projects/file.csv`. To enable this type of access, several environment variables need to be configured in the pod that provide the v3io API URL and access keys.
2. `v3io` storage through FUSE mount — Some tools cannot utilize the `v3io` API to access it and need basic filesystem semantics. For that purpose, `v3io` provides a FUSE (Filesystem in user-space) driver that can be used to mount `v3io` containers as specific paths in the pod itself. For example `/User`. To enable this, several specific volume mount configurations need to be applied to the pod spec.
3. NFS storage access — When MLRun is deployed as open-source, independent of Iguazio, the deployment automatically adds a pod running NFS storage. To access this NFS storage through pods, a kubernetes `pvc` mount is needed.
4. Others — As use-cases evolve, other cases of storage access may be needed. This will require various configurations to be applied to function execution pods.

MLRun attempts to offload this storage configuration task from the user by automatically applying the most common storage configuration to functions. As a result, most cases do not require any additional storage configurations before executing a function as a Kubernetes pod. The configurations applied by MLRun are:

- In an Iguazio system, apply configurations for `v3io` access through the API.
- In an open-source deployment where NFS is configured, apply configurations for `pvc` access to NFS storage.

This MLRun logic is referred to as **auto-mount**.

In this section

Disabling auto-mount

In cases where the default storage configuration does not fit the function needs, MLRun allows for function spec modifiers to be manually applied to functions. These modifiers can add various configurations to the function spec, adding environment variables, mounts and additional configurations. MLRun also provides a set of common modifiers that can be used to apply storage configurations. These modifiers can be applied by using the `.apply()` method on the function and adding the modifier to apply. You can see some examples of this later in this page.

When a different storage configuration is manually applied to a function, MLRun's auto-mount logic is disabled. This prevents conflicts between configurations. The auto-mount logic can also be disabled by setting `func.spec.disable_auto_mount = True` on any MLRun function.

Modifying the auto-mount default configuration

The default auto-mount behavior applied by MLRun is controlled by setting MLRun configuration parameters. For example, the logic can be set to automatically mount the `v3io` FUSE driver on all functions, or perform `pvc` mount for NFS storage on all functions. The following code demonstrates how to apply the `v3io` FUSE driver by default:

```
# Change MLRun auto-mount configuration
import mlrun.mlconf

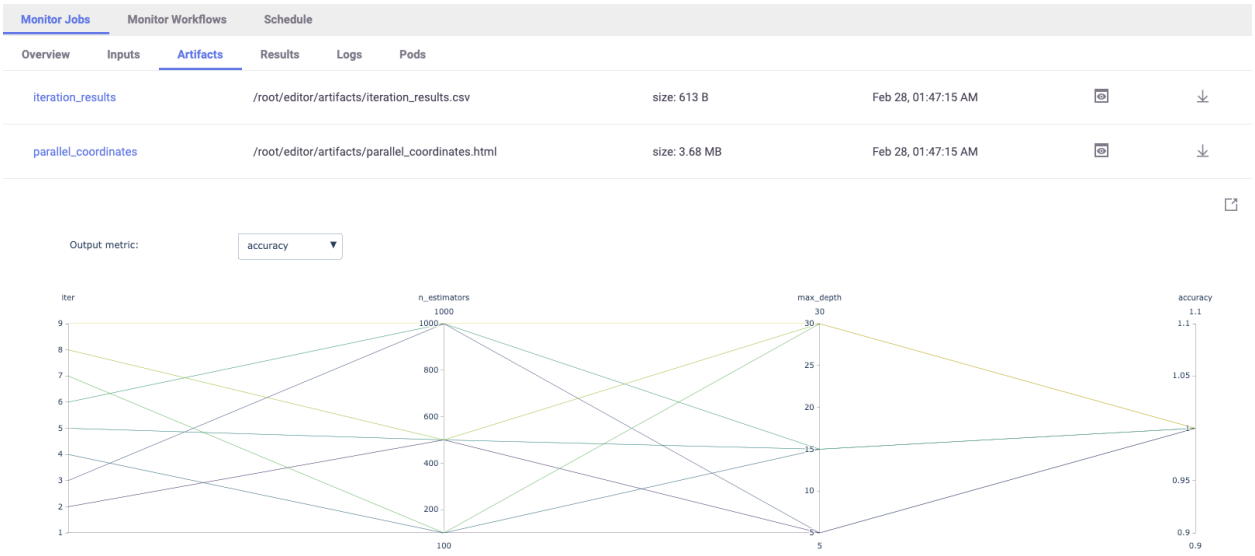
mlrun.mlconf.storage.auto_mount_type = "v3io_fuse"
```

Each of the auto-mount supported methods applies a specific modifier function. The supported methods are:

- `v3io_credentials` — apply `v3io` credentials needed for `v3io` API usage. Applies the *`v3io_cred()`* modifier.
- `v3io_fuse` — create Fuse driver mount. Applies the *`mount_v3io()`* modifier.
- `pvc` — create a `pvc` mount. Applies the *`mount_pvc()`* modifier.
- `auto` — the default auto-mount logic as described above (either `v3io_credentials` or `pvc`).
- `none` — perform no auto-mount (same as using `disable_auto_mount = True`).

The modifier functions executed by auto-mount can be further configured by specifying their parameters. These can be provided in the `storage.auto_mount_params` configuration parameters. Parameters can be passed as a string made of `key=value` pairs separated by commas. For example, the following code runs a `pvc` mount with specific parameters:

```
mlrun.mlconf.storage.auto_mount_type = "pvc"
pvc_params = {
    "pvc_name": "my_pvc_mount",
    "volume_name": "pvc_volume",
    "volume_mount_path": "/mnt/storage/nfs",
}
mlrun.mlconf.storage.auto_mount_params = ",".join(
    [f"{key}={value}" for key, value in pvc_params.items()]
)
```

Alternatively, the parameters can be provided as a base64-encoded JSON object, which can be useful when passing complex parameters or strings that contain special characters:

```
pvc_params_str = base64.b64encode(json.dumps(pvc_params).encode())
mlrun.mlconf.storage.auto_mount_params = pvc_params_str
```

MLRun Functions Marketplace

In this section

- *Running the function*

Overview

This tutorial demonstrates how to import a function from the marketplace into your project, and provides some basic instructions on how to run the function and view their results.

Functions Marketplace

MLRun marketplace has a wide range of functions that can be used for a variety of use cases. There are functions for ETL, data preparation, training (ML & Deep learning), serving, alerts and notifications and more. Each function has a docstring that explains how to use it. In addition, the functions are associated with categories to make it easier for you to find the relevant one.

Functions can be easily imported into your project and therefore help users to speed up their development cycle by reusing built-in code.

Searching for functions

The Marketplace is stored in this GitHub repo: https://github.com/mlrun/functions. See the README file for the list of functions in the marketplace and their categories.

Setting the project configuration

The first step for each project is to set the project name and path:

```
from os import path, getenv
from mlrun import new_project

project_name = 'load-func'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

print(f'Project path: {project_path}\nProject name: {project_name}')
```

Set the artifacts path

The artifact path is the default path for saving all the artifacts that the functions generate:

```
from mlrun import run_local, mlconf, import_function, mount_v3io

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

print(f'Artifacts path: {artifact_path}\nMLRun DB path: {mlconf.dbpath}')
```

Loading functions from the Marketplace

Run `project.set_function` to load a functions. `set_function` updates or adds a function object to the project.

`set_function(func, name='', kind='', image=None, with_repo=None)`

Parameters:

- **func** — function object or spec/code url.
- **name** — name of the function (under the project).
- **kind** — runtime kind e.g. job, nuclio, spark, dask, mpijob. Default: job.
- **image** — docker image to be used, can also be specified in the function object/yaml.
- **with_repo** — add (clone) the current repo to the build source.

Returns: project object

For more information see the `set_function` API documentation.

Load function example

This example loads the describe function. This function analyzes a csv or parquet file for data analysis.

```
project.set_function('hub://describe', 'describe')
```

Create a function object called my_describe:

```
my_describe = project.func('describe')
```

View the function params

To view the parameters, run the function with .doc():

```
my_describe.doc()
```

```
    function: describe
    describe and visualizes dataset stats
    default handler: summarize
    entry points:
      summarize: Summarize a table
        context(MLClientCtx)  - the function context, default=
        table(DataItem)  - MLRun input pointing to pandas dataframe (csv/parquet file␣
↪path), default=
        label_column(str)  - ground truth column label, default=None
        class_labels(List[str])  - label for each class in tables and plots,␣
↪default=[]
        plot_hist(bool)  - (True) set this to False for large tables, default=True
        plots_dest(str)  - destination folder of summary plots (relative to artifact_
↪path), default=plots
        update_dataset  - when the table is a registered dataset update the charts in-
↪place, default=False
```

Running the function

Use the `run` method to to run the function.

When working with functions pay attention to the following:

- Input vs. params — for sending data items to a function, send it via "inputs" and not as params.
- Working with artifacts — Artifacts from each run are stored in the `artifact_path`, which can be set globally with the environment variable (MLRUN_ARTIFACT_PATH) or with the config. If it's not already set you can create a directory and use it in the runs. Using `{{run.uid}}` in the path creates a unique directory per run. When using pipelines you can use the `{{workflow.uid}}` template option.

This example runs the describe function. This function analyzes a dataset (in this case it's a csv file) and generates HTML files (e.g. correlation, histogram) and saves them under the artifact path.

```
DATA_URL = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'

my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=artifact_path)
```

Saving the artifacts in a unique folder for each run

```
out = mlconf.artifact_path or path.abspath('./data')
my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=path.join(out, '{{run.uid}}'))
```

Viewing the jobs & the artifacts

There are few options to view the outputs of the jobs we ran:

- In Jupyter the result of the job is displayed in the Jupyter notebook. When you click on the artifacts it displays its content in Jupyter.
- In the MLRun UI, under the project name, you can view the job that was running as well as the artifacts it generated.

Images and their usage in MLRun

Every release of MLRun includes several images for different usages. The build and the infrastructure images are described, and located, in the README. They are also published to dockerhub and quay.io.

In this section

- *Using images*
- *MLRun images and how to build them*
- *MLRun images and external docker images*

Using images

See *Kubernetes Jobs & Images*.

MLRun images and how to build them

See README.

MLRun images and external docker images

There is no difference in the usage between the MLRun images and external docker images. However:

- MLRun images resolve auto tags: If you specify `image="mlrun/mlrun"` the API fills in the tag by the client version, e.g. changes it to `mlrun/mlrun:1.0.0`. So, if the client gets upgraded you'll automatically get a new image tag.
- Where the data node registry exists, MLRun: Appends the registry prefix, so the image loads from the datanode registry. This pulls the image more quickly, and also supports air-gapped sites. When you specify an MLRun image, for example `mlrun/mlrun:1.0.0`, the actual image used is similar to `datanode-registry.iguazio-platform.app.vm/mlrun/mlrun:1.0.0`.

These characteristics are great when you're working in a POC or development environment. But MLRun typically upgrades packages as part of the image, and therefore the default MLRun images can break your product flow.

Working with images in production

For production you should create your own images to ensure that the image is fixed.

- Pin the image tag, e.g. `image="mlrun/mlrun:1.0.0"`. This maintains the image tag at 0.10.0 even when the client is upgraded. Otherwise, an upgrade of the client would also upgrade the image. (If you specify an external (not MLRun images) docker image, like python, the result is the docker/k8s default behavior, which defaults to `latest` when the tag is not provided.)
- Pin the versions of requirements, again to avoid breakages, e.g. `pandas==1.4.0`. (If you only specify the package name, e.g. pandas, then pip/conda (python's package managers) just pick up the latest version.)

2.1.16 Run, track, and compare jobs

In this section

Running simple jobs

Run with CLI locally: CSV output

Run the data generation function `gen-iris` (see `gen_iris.py`:

```
mlrun run -f gen-iris --local
```

`--local` indicates that it's running locally (not over the cluster)

Notice the use of MLRun `with mlrun.get_or_create_ctx()` in the code. It wraps your code, automatically tracking the execution.

View the output CSV file in `artifacts/dataset.csv`

Run with CLI locally: python output

You can also run the function using the regular `Python` command:

```
python3 gen_iris.py
```

However, when using `python` you have limited control over the execution. See more in *MLRun execution context*.

Run with CLI locally: parquet output

Run the function again, and this time pass the `format=parquet` arg:

```
mlrun run -f gen-iris -p format=parquet --local
```

This time the dataset is created in `parquet` format:

`-p` is used to specify parameters. See *CLI* for more flags.

Results can be accessed via the CLI, SDK, or UI.

Run with SDK

Initialize an MLRun Project and Generate Data. First load the MLRun project:

```
project = mlrun.load_project("./", init_git=True)
```

Run the data generation function `gen-iris` as in the previous scenario:

```
gen_data_run = project.run_function("gen-iris", local=True)
```

Viewing the output

The auto-logging for SciKit-Learn includes many plots and metrics. The metrics and artifacts are chosen according to the model algorithm used (classification or regression). You can add more metrics and artifacts and even write your own.

Print the metrics and artifacts generated automatically by the apply_mlrun()` method:

```
pprint(trainer_run.outputs)
```

Once the run is complete open the MLRun UI and see the artifacts and results, similar to:

All of the metrics and artifacts are stored as metadata of the model, so it's easy to do a comparison between models later on.

###Model Registry

Models and their metadata are automatically stored in the **Model Registry**. Check the Model Artifacts Tab to see the models.

Model objects can be read from the MLRun DB and used in different applications. The following example lists all of themodel objects and prints one as YAML. Details that are automatically stored with the model include results, artifacts, statistics, schema, etc.

```
models = mlrun.get_run_db().list_artifacts(kind="model")
print(models.objects()[0].to_yaml())
```

Train the model

Now, assume you have a training function `trainer.py` to train a model on the training data. Execute the training function with specific data inputs and parameters. These are recorded and versioned with the run. The training function is already set in the project, so all you need to do is use the dataset from the `gen_data_run` outputs:

```
trainer_run = project.run_function(
    "trainer",
    inputs={"dataset": gen_data_run.outputs["dataset"]},
    params = {"n_estimators": 100, "max_depth": 5},
    local=True
)
```

Hyperparam and iterative jobs

MLRun supports iterative tasks for automatic and distributed execution of many tasks with variable parameters (hyperparams). Iterative tasks can be distributed across multiple containers. They can be used for:

- Parallel loading and preparation of many data objects
- Model training with different parameter sets and/or algorithms
- Parallel testing with many test vector options
- AutoML

MLRun iterations can be viewed as child runs under the main task/run. Each child run gets a set of parameters that are computed/selected from the input hyperparameters based on the chosen strategy (*Grid*, *List*, *Random* or *Custom*).

The different iterations can run in parallel over multiple containers (using Dask or Nuclio runtimes, which manage the workers). Read more in *Parallel execution over containers*.

The hyperparameters and options are specified in the `task` or the *`run()`* command through the `hyperparams` (for hyperparam values) and `hyper_param_options` (for *`HyperParamOptions`*) properties. See the examples below. Hyperparameters can also be loaded directly from a CSV or Json file (by setting the `param_file` hyper option).

The hyperparams are specified as a struct of `key: list` values for example: `{"p1": [1,2,3], "p2": [10,20]}`. The values can be of any type (int, string, float, ..). The lists are used to compute the parameter combinations using one of the following strategies:

- Grid Search (`grid`) — running all the parameter combinations.
- Random (`random`) — running a sampled set from all the parameter combinations.
- List (`list`) — running the first parameter from each list followed by the seco2nd from each list and so on. **All the lists must be of equal size**.
- Custom (`custom`) — determine the parameter combination per run programmatically.

You can specify a selection criteria to select the best run among the different child runs by setting the `selector` option. This marks the selected result as the parent (iteration 0) result, and marks the best result in the user interface.

You can also specify the `stop_condition` to stop the execution of child runs when some criteria, based on the returned results, is met (for example `stop_condition="accuracy>=0.9"`).

In this section

Basic code

Here's a basic example of running multiple jobs in parallel for **hyperparameters tuning**, selecting the best run with respect to the `max accuracy`.

Run the hyperparameters tuning job by using the keywords arguments:

- `hyperparams` for the hyperparameters options and values of choice.
- `selector` for specifying how to select the best model.

```
hp_tuning_run = project.run_function(
    "trainer",
    inputs={"dataset": gen_data_run.outputs["dataset"]},
    hyperparams={
        "n_estimators": [100, 500, 1000],
        "max_depth": [5, 15, 30]
    },
    selector="max.accuracy",
    local=True
)
```

The returned run object in this case represents the `parent` (and the **best** result). You can also access the individual child runs (called iterations) in the MLRun UI.

Review the results

When running a hyperparam job, the job `results` tab shows the list and marks the best run:

You can also view results by printing the artifact `iteration_results`:

`hp_tuning_run.artifact("iteration_results").as_df()`

MLRun also generates a `parallel coordinates plot` for the run, you can view it in the MLRun UI.



Examples

Base dummy function:

```
import mlrun
```

```
> 2021-10-23 12:47:39,982 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-10-23 12:47:43,488 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': '0.8.0-rc7', 'client_version': 'unstable'}
```

```
def hyper_func(context, p1, p2):
    print(f"p1={p1}, p2={p2}, result={p1 * p2}")
    context.log_result("multiplier", p1 * p2)
```

Grid search (default)

```
grid_params = {"p1": [2,4,1], "p2": [10,20]}
task = mlrun.new_task("grid-demo").with_hyper_params(grid_params, selector="max.
↪multiplier")
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:43,505 [info] starting run grid-demo␣
↪uid=29c9083db6774e5096a97c9b6b6c8e93 DB=http://mlrun-api:8080
p1=2, p2=10, result=20
p1=4, p2=10, result=40
p1=1, p2=10, result=10
p1=2, p2=20, result=40
p1=4, p2=20, result=80
p1=1, p2=20, result=20
> 2021-10-23 12:47:44,851 [info] best iteration=5, used criteria max.multiplier
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:45,071 [info] run executed, status=completed
```

UI Screenshot:

Random Search

MLRun chooses random parameter combinations. Limit the number of combinations using the `max_iterations` attribute.

```
grid_params = {"p1": [2,4,1,3], "p2": [10,20,30]}
task = mlrun.new_task("random-demo")
task.with_hyper_params(grid_params, selector="max.multiplier", strategy="random", max_
↪iterations=4)
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:45,077 [info] starting run random-demo␣
↪uid=cac368c7fc33455f97ca806e5c7abf2f DB=http://mlrun-api:8080
p1=2, p2=20, result=40
p1=4, p2=10, result=40
p1=3, p2=10, result=30
p1=3, p2=20, result=60
> 2021-10-23 12:47:45,966 [info] best iteration=4, used criteria max.multiplier
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:46,177 [info] run executed, status=completed
```

List search

This example also shows how to use the `stop_condition` option.

```
list_params = {"p1": [2,3,7,4,5], "p2": [15,10,10,20,30]}
task = mlrun.new_task("list-demo").with_hyper_params(
    list_params, selector="max.multiplier", strategy="list", stop_condition=
↪"multiplier>=70")
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:46,184 [info] starting run list-demo␣
↪uid=136edfb9c9404a61933c73bbbd35b18b DB=http://mlrun-api:8080
p1=2, p2=15, result=30
```

(continues on next page)

(continued from previous page)

```
p1=3, p2=10, result=30
p1=7, p2=10, result=70
> 2021-10-23 12:47:47,193 [info] reached early stop condition (multiplier>=70),␣
↪stopping iterations!
> 2021-10-23 12:47:47,195 [info] best iteration=3, used criteria max.multiplier
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:47,385 [info] run executed, status=completed
```

Custom iterator

You can define a child iteration context under the parent/main run. The child run is logged independently.

```
def handler(context: mlrun.MLClientCtx, param_list):
    best_multiplier = total = 0
    for param in param_list:
        with context.get_child_context(**param) as child:
            hyper_func(child, **child.parameters)
            multiplier = child.results['multiplier']
            total += multiplier
            if multiplier > best_multiplier:
                child.mark_as_best()
                best_multiplier = multiplier

    # log result at the parent
    context.log_result('avg_multiplier', total / len(param_list))
```

```
param_list = [{"p1":2, "p2":10}, {"p1":3, "p2":30}, {"p1":4, "p2":7}]
run = mlrun.new_function().run(handler=handler, params={"param_list": param_list})
```

```
> 2021-10-23 12:47:47,403 [info] starting run mlrun-a79c5c-handler␣
↪uid=c3eb08ebae02464ca4025c77b12e3c39 DB=http://mlrun-api:8080
p1=2, p2=10, result=20
p1=3, p2=30, result=90
p1=4, p2=7, result=28
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:48,734 [info] run executed, status=completed
```

Parallel execution over containers

When working with compute intensive or long running tasks you'll want to run your iterations over a cluster of containers. At the same time, you don't want to bring up too many containers, and you want to limit the number of parallel tasks.

MLRun supports distribution of the child runs over Dask or Nuclio clusters. This is handled automatically by MLRun. You only need to deploy the Dask or Nuclio function used by the workers, and set the level of parallelism in the task. The execution can be controlled from the client/notebook, or can have a job (immediate or scheduled) that controls the execution.

Code example (single task)

```
# mark the start of a code section that will be sent to the job
# mlrun: start-code
```

```
import socket
import pandas as pd
def hyper_func2(context, data, p1, p2, p3):
    print(data.as_df().head())
    context.logger.info(f"p2={p2}, p3={p3}, r1={p2 * p3} at {socket.gethostname()}")
    context.log_result("r1", p2 * p3)
    raw_data = {
        "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
        "age": [42, 52, 36, 24, 73],
        "testScore": [25, 94, 57, 62, 70],
    }
    df = pd.DataFrame(raw_data, columns=["first_name", "age", "testScore"])
    context.log_dataset("mydf", df=df, stats=True)
```

```
# mlrun: end-code
```

Running the workers using Dask

This example creates a new function and executes the parent/controller as an MLRun `job` and the different child runs over a Dask cluster (MLRun Dask function).

Define a Dask cluster (using MLRun serverless Dask)

```
dask_cluster = mlrun.new_function("dask-cluster", kind='dask', image='mlrun/ml-models
↪')
dask_cluster.apply(mlrun.mount_v3io())        # add volume mounts
dask_cluster.spec.service_type = "NodePort"   # open interface to the dask UI␣
↪dashboard
dask_cluster.spec.replicas = 2                # define two containers
uri = dask_cluster.save()
uri
```

```
'db://default/dask-cluster'
```

```
# initialize the dask cluster and get its dashboard url
dask_cluster.client
```

```
> 2021-10-23 12:48:49,020 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786
> 2021-10-23 12:48:49,049 [info] using remote dask scheduler (mlrun-dask-cluster-
↪eea516ff-5) at: tcp://mlrun-dask-cluster-eea516ff-5.default-tenant:8786
```

```
Mismatched versions found

+-------------+--------+-----------+---------+
| Package     | client | scheduler | workers |
+-------------+--------+-----------+---------+
| blosc       | 1.7.0  | 1.10.6    | None    |
| cloudpickle | 1.6.0  | 2.0.0     | None    |
| distributed | 2.30.0 | 2.30.1    | None    |
| lz4         | 3.1.0  | 3.1.3     | None    |
| msgpack     | 1.0.0  | 1.0.2     | None    |
| tornado     | 6.0.4  | 6.1       | None    |
+-------------+--------+-----------+---------+
Notes:
-  msgpack: Variation is ok, as long as everything is above 0.6
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.0.72:8786' processes=0 threads=0, memory=0 B>
```

Define the parallel work

Set the `parallel_runs` attribute to indicate how many child tasks to run in parallel. Set the `dask_cluster_uri` to point to the dask cluster (if it's not set the cluster uri uses dask local). You can also set the `teardown_dask` flag to free up all the dask resources after completion.

```
grid_params = {"p2": [2,1,4,1], "p3": [10,20]}
task = mlrun.new_task(params={"p1": 8}, inputs={'data': 'https://s3.wasabisys.com/
↪iguazio/data/iris/iris_dataset.csv'})
task.with_hyper_params(
    grid_params, selector="r1", strategy="grid", parallel_runs=4, dask_cluster_
↪uri=uri, teardown_dask=True
)
```

```
<mlrun.model.RunTemplate at 0x7f673d7b1910>
```

Define a job that will take the code (using `code_to_function`) and run it over the cluster

```
fn = mlrun.code_to_function(name='hyper-tst', kind='job', image='mlrun/ml-models')
```

```
run = fn.run(task, handler=hyper_func2)
```

```
> 2021-10-23 12:49:56,388 [info] starting run hyper-tst-hyper_func2␣
↪uid=50eb72f5b0734954b8b1c57494f325bc DB=http://mlrun-api:8080
> 2021-10-23 12:49:56,565 [info] Job is running in the background, pod: hyper-tst-
↪hyper-func2-9g6z8
```

(continues on next page)

(continued from previous page)

```
> 2021-10-23 12:49:59,813 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786
> 2021-10-23 12:50:09,828 [warning] remote scheduler at tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786 not ready, will try to restart Timed out trying to␣
↪connect to tcp://mlrun-dask-cluster-eea516ff-5.default-tenant:8786 after 10 s
> 2021-10-23 12:50:15,733 [info] using remote dask scheduler (mlrun-dask-cluster-
↪04574796-5) at: tcp://mlrun-dask-cluster-04574796-5.default-tenant:8786
remote dashboard: default-tenant.app.yh38.iguazio-cd2.com:32577
> --------------- Iteration: (1) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,353 [info] p2=2, p3=10, r1=20 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (3) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,459 [info] p2=4, p3=10, r1=40 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (4) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,542 [info] p2=1, p3=10, r1=10 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (6) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,629 [info] p2=1, p3=20, r1=20 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (7) ---------------
```

(continues on next page)

(continued from previous page)

```
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,792 [info] p2=4, p3=20, r1=80 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (8) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:22,052 [info] p2=1, p3=20, r1=20 at mlrun-dask-cluster-04574796-
↪5k5lhq

> --------------- Iteration: (2) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:23,134 [info] p2=1, p3=10, r1=10 at mlrun-dask-cluster-04574796-
↪5j6v59

> --------------- Iteration: (5) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:23,219 [info] p2=2, p3=20, r1=40 at mlrun-dask-cluster-04574796-
↪5k5lhq

> 2021-10-23 12:50:23,261 [info] tearing down the dask cluster..
> 2021-10-23 12:50:43,363 [info] best iteration=7, used criteria r1
> 2021-10-23 12:50:43,626 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:50:53,303 [info] run executed, status=completed
```

Running the workers using Nuclio

Nuclio is a high-performance serverless engine that can process many events in parallel. It can also separate initialization from execution. Certain parts of the code (imports, loading data, etc.) can be done once per worker vs. in any run.

Nuclio, by default, process events (http, stream, ..). There is a special Nuclio kind that runs MLRun jobs (nuclio:mlrun).

Notes

- Nuclio tasks are relatively short (preferably under 5 minutes), use it for running many iterations where each individual run is less than 5 min.
- Use `context.logger` to drive text outputs (vs `print()`).

Create a nuclio:mlrun function

```
fn = mlrun.code_to_function(name='hyper-tst2', kind='nuclio:mlrun', image='mlrun/mlrun
↪')
# replicas * workers need to match or exceed parallel_runs
fn.spec.replicas = 2
fn.with_http(workers=2)
fn.deploy()
```

```
> 2021-10-23 12:51:10,152 [info] Starting remote function deploy
2021-10-23 12:51:10  (info) Deploying function
2021-10-23 12:51:10  (info) Building
2021-10-23 12:51:10  (info) Staging files and preparing base images
2021-10-23 12:51:10  (info) Building processor image
2021-10-23 12:51:11  (info) Build complete
2021-10-23 12:51:19  (info) Function deploy complete
> 2021-10-23 12:51:22,296 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-default-hyper-tst2.default-tenant.svc.cluster.local:8080
↪'], 'external_invocation_urls': ['default-tenant.app.yh38.iguazio-cd2.com:32760']}
```

```
'http://default-tenant.app.yh38.iguazio-cd2.com:32760'
```

Run the parallel task over the function

```
# this is required to fix Jupyter issue with asyncio (not required outside of Jupyter)
# run it only once
import nest_asyncio
nest_asyncio.apply()
```

```
grid_params = {"p2": [2,1,4,1], "p3": [10,20]}
task = mlrun.new_task(params={"p1": 8}, inputs={'data': 'https://s3.wasabisys.com/
↪iguazio/data/iris/iris_dataset.csv'})
task.with_hyper_params(
    grid_params, selector="r1", strategy="grid", parallel_runs=4, max_errors=3
)
run = fn.run(task, handler=hyper_func2)
```

```
> 2021-10-23 12:51:31,618 [info] starting run hyper-tst2-hyper_func2␣
↪uid=97cc3e255f3c4c93822b0154d63f47f5 DB=http://mlrun-api:8080
> --------------- Iteration: (4) ---------------
2021-10-23 12:51:32.130812  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=1
2021-10-23 12:51:32.401258  info   p2=1, p3=10, r1=10 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=1

> --------------- Iteration: (2) ---------------
2021-10-23 12:51:32.130713  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=0
2021-10-23 12:51:32.409468  info   p2=1, p3=10, r1=10 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> --------------- Iteration: (1) ---------------
2021-10-23 12:51:32.130765  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=0
2021-10-23 12:51:32.432121  info   p2=2, p3=10, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-2gdtc  worker_id=0

> --------------- Iteration: (5) ---------------
2021-10-23 12:51:32.568848  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=0
2021-10-23 12:51:32.716415  info   p2=2, p3=20, r1=40 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> --------------- Iteration: (7) ---------------
2021-10-23 12:51:32.855399  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=1
2021-10-23 12:51:33.054417  info   p2=4, p3=20, r1=80 at nuclio-default-hyper-tst2-
↪5d4976b685-2gdtc  worker_id=1

> --------------- Iteration: (6) ---------------
2021-10-23 12:51:32.970002  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=0
2021-10-23 12:51:33.136621  info   p2=1, p3=20, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0
```

(continues on next page)

(continued from previous page)

```
> --------------- Iteration: (3) ---------------
2021-10-23 12:51:32.541187  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=1
2021-10-23 12:51:33.301200  info   p2=4, p3=10, r1=40 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=1

> --------------- Iteration: (8) ---------------
2021-10-23 12:51:33.419442  info   logging run results to: http://mlrun-api:8080 ␣
↪worker_id=0
2021-10-23 12:51:33.672165  info   p2=1, p3=20, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> 2021-10-23 12:51:34,153 [info] best iteration=7, used criteria r1
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:51:34,420 [info] run executed, status=completed
```

Build and use function images (Kubernetes)

This topic describes running a kubernetes-based job using shared data, and building custom container images.

In this section

- *Define a new function and its dependencies*
- *Convert the code to a serverless job*
- *Deploy (build) the function container*
- *Run the function on the cCluster*
- *Create and run a Kubeflow pipeline*

Define a new function and its dependencies

Define a single serverless function with two `handlers`, one for training and one for validation.

```
import mlrun
```

```
> 2021-01-24 00:04:38,841 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-01-24 00:04:40,691 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
```

```
import time
import pandas as pd
from mlrun.artifacts import get_model, update_model
```

(continues on next page)

(continued from previous page)

```
def training(
    context,
    p1: int = 1,
    p2: int = 2
) -> None:
    """Train a model.

    :param context: The runtime context object.
    :param p1: A model parameter.
    :param p2: Another model parameter.
    """
    # access input metadata, values, and inputs
    print(f'Run: {context.name} (uid={context.uid})')
    print(f'Params: p1={p1}, p2={p2}')
    context.logger.info('started training')

    # <insert training code here>

    # log the run results (scalar values)
    context.log_result('accuracy', p1 * 2)
    context.log_result('loss', p1 * 3)

    # add a label/tag to this run
    context.set_label('category', 'tests')

    # log a simple artifact + label the artifact
    # If you want to upload a local file to the artifact repo add src_path=<local-
↪path>
    context.log_artifact('somefile',
                         body=b'abc is 123',
                         local_path='myfile.txt')

    # create a dataframe artifact
    df = pd.DataFrame([{'A':10, 'B':100}, {'A':11,'B':110}, {'A':12,'B':120}])
    context.log_dataset('mydf', df=df)

    # Log an ML Model artifact, add metrics, params, and labels to it
    # and place it in a subdir ('models') under artifacts path
    context.log_model('mymodel', body=b'abc is 123',
                      model_file='model.txt',
                      metrics={'accuracy':0.85}, parameters={'xx':'abc'},
                      labels={'framework': 'xgboost'},
                      artifact_path=context.artifact_subpath('models'))
```

```
def validation(
    context,
    model: mlrun.DataItem
) -> None:
    """Model validation.

    Dummy validation function.

    :param context: The runtime context object.
    :param model: The extimated model object.
    """
```

(continues on next page)

(continued from previous page)

```
    # access input metadata, values, files, and secrets (passwords)
    print(f'Run: {context.name} (uid={context.uid})')
    context.logger.info('started validation')

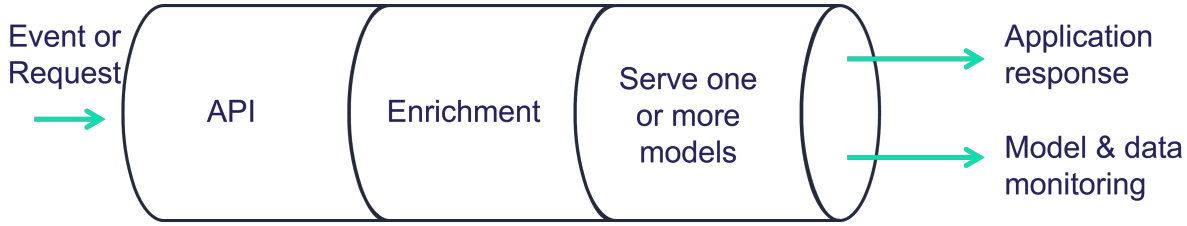
    # get the model file, class (metadata), and extra_data (dict of key: DataItem)
    model_file, model_obj, _ = get_model(model)

    # update model object elements and data
    update_model(model_obj, parameters={'one_more': 5})

    print(f'path to local copy of model file - {model_file}')
    print('parameters:', model_obj.parameters)
    print('metrics:', model_obj.metrics)
    context.log_artifact('validation',
                         body=b'<b> validated </b>',
                         format='html')
```

The following end-code annotation tells MLRun to stop parsing the notebook from this cell. ***Do not remove this cell***:

```
# mlrun: end-code
```

Convert the code to a serverless job

Create a `function` that defines the runtime environment (type, code, image, ..) and `run()` a job or experiment using that function. In each run, you can specify the function, inputs, parameters/hyper-parameters, etc.

Use the `job` runtime for running container jobs, or alternatively use another distributed runner like MpiJob, Spark, Dask, and Nuclio.

Setting up the environment

```
project_name, artifact_path = mlrun.set_environment(project='jobs-demo', artifact_
↪path='./data/{{run.uid}}')
```

Define the cluster jobs, build images, and set dependencies

To use the function in a cluster you need to package the code and its dependencies.

The `code_to_function` call automatically generates a `function` object from the current notebook (or specified file) with its list of dependencies and runtime configuration. In this example the code depends on the pandas package, so so it's specified in the `code_to_function` call.

```
# create an ML function from the notebook, attach it to iguazio data fabric (v3io)
trainer = mlrun.code_to_function(name='my-trainer', kind='job', image='mlrun/mlrun',␣
↪requirements=['pandas'])
```

The functions need a shared storage media (file or object) to pass and store artifacts.

You can add ***Kubernetes*** resources like volumes, environment variables, secrets, cpu/mem/gpu, etc. to a function.

`mlrun` uses ***KubeFlow*** modifiers (apply) to configure resources. You can build your own resources or use predefined resources e.g. AWS resources.

The example above uses built-in images. When you move to production, use specific tags. For more details on built-in and custom images, see MLRun images and external docker images.

Option 1: Using file volumes for artifacts

MLRun automatically applies the most common storage configuration to functions. As a result, most cases do not require any additional storage configurations before executing a function. See more details in *Applying storage configurations to functions*.

If you're using the Iguazio MLOps platform, and want to configure manually, use the `mount_v3io()` auto-mount modifier. If you're using another k8s PVC volume, use the `mlrun.platforms.mount_pvc(..)` modifier with the required parameters.

This example uses the `auto_mount()` modifier. It auto-selects between the k8s PVC volume and the Iguazio data fabric. You can set the PVC volume configuration with the env var below or with the `auto_mount` params:

```
MLRUN_PVC_MOUNT=<pvc-name>:<mount-path>
```

If you apply `mount_v3io()` or `auto_mount()` when running the function in the MLOps platform, it attaches the function to Iguazio's real-time data fabric (mounted by default to ***home*** of the current user).

Note: If the notebook is not on the managed platform (it's running remotely) you might need to use secrets.

For the current `training` function, run:

```
# for PVC volumes set the env var for PVC: MLRUN_PVC_MOUNT=<pvc-name>:<mount-path>,␣
↪pass the relevant parameters
from mlrun.platforms import auto_mount
trainer.apply(auto_mount())
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7fdf29b1c190>
```

Option 2: Using AWS S3 for artifacts

When using AWS, you can use S3. See more details in S3.

Deploy (build) the function container

The `deploy()` command builds a custom container image (creates a cluster build job) from the outlined function dependencies.

If a pre-built container image already exists, pass the `image` name instead. ***The code and params can be updated per run without building a new image***.

The image is stored in a container repository. By default it uses the repository configured on the MLRun API service. You can specify your own docker registry by first creating a secret, and adding that secret name to the build configuration:

```
kubectl create -n <namespace> secret docker-registry my-docker
--docker-server=https://index.docker.io/v1/ --docker-username=<your-user>
--docker-password=<your-password> --docker-email=<your-email>
```

And then run this:

```
trainer.build_config(image='target/image:tag', secret='my_docker')
```

```
trainer.deploy(with_mlrun=False)
```

```
> 2021-01-24 00:05:18,384 [info] starting remote build, image: .mlrun/func-jobs-demo-
↪my-trainer-latest
INFO[0020] Retrieving image manifest mlrun/mlrun:unstable
INFO[0020] Retrieving image manifest mlrun/mlrun:unstable
INFO[0021] Built cross stage deps: map[]
INFO[0021] Retrieving image manifest mlrun/mlrun:unstable
INFO[0021] Retrieving image manifest mlrun/mlrun:unstable
INFO[0021] Executing 0 build triggers
INFO[0021] Unpacking rootfs as cmd RUN pip install pandas requires it.
INFO[0037] RUN pip install pandas
INFO[0037] Taking snapshot of full filesystem...
INFO[0050] cmd: /bin/sh
INFO[0050] args: [-c pip install pandas]
INFO[0050] Running: [/bin/sh -c pip install pandas]
Requirement already satisfied: pandas in /usr/local/lib/python3.7/site-packages (1.2.
↪0)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/site-packages␣
↪(from pandas) (2020.5)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/
↪site-packages (from pandas) (2.8.1)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.7/site-
↪packages (from pandas) (1.19.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/site-packages␣
↪(from python-dateutil>=2.7.3->pandas) (1.15.0)
WARNING: You are using pip version 20.2.4; however, version 21.0 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade␣
↪pip' command.
INFO[0051] Taking snapshot of full filesystem...
```

```
True
```

Run the function on the cluster

Use `with_code` to inject the latest code into the function (without requiring a new build).

```
trainer.with_code()
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7fdf29b1c190>
```

```
# run our training task with params
train_run = trainer.run(name='my-training', handler='training', params={'p1': 9})
```

```
> 2021-01-24 00:09:14,760 [info] starting run my-training␣
↪uid=30b8131285a74f87b16d957fabc5fac3 DB=http://mlrun-api:8080
> 2021-01-24 00:09:14,928 [info] Job is running in the background, pod: my-training-
↪lhtxt
> 2021-01-24 00:09:18,972 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
Run: my-training (uid=30b8131285a74f87b16d957fabc5fac3)
Params: p1=9, p2=2
> 2021-01-24 00:09:19,050 [info] started training
```

(continues on next page)

(continued from previous page)

```
> 2021-01-24 00:09:19,299 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 30b8131285a74f87b16d957fabc5fac3 --project jobs-demo , !mlrun logs␣
↪30b8131285a74f87b16d957fabc5fac3 --project jobs-demo
> 2021-01-24 00:09:21,253 [info] run executed, status=completed
```

```
# running validation, use the model result from the previous step
model = train_run.outputs['mymodel']
validation_run = trainer.run(name='validation', handler='validation', inputs={'model
↪': model}, watch=True)
```

```
> 2021-01-24 00:09:21,259 [info] starting run validation␣
↪uid=c757ffcdc36d4412b4bcba1df75f079d DB=http://mlrun-api:8080
> 2021-01-24 00:09:21,536 [info] Job is running in the background, pod: validation-
↪dwd78
> 2021-01-24 00:09:25,570 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
Run: validation (uid=c757ffcdc36d4412b4bcba1df75f079d)
> 2021-01-24 00:09:25,719 [info] started validation
path to local copy of model file - /User/data/30b8131285a74f87b16d957fabc5fac3/models/
↪model.txt
parameters: {'xx': 'abc', 'one_more': 5}
metrics: {'accuracy': 0.85}
> 2021-01-24 00:09:25,873 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run c757ffcdc36d4412b4bcba1df75f079d --project jobs-demo , !mlrun logs␣
↪c757ffcdc36d4412b4bcba1df75f079d --project jobs-demo
> 2021-01-24 00:09:27,647 [info] run executed, status=completed
```

Create and run a Kubeflow pipeline

Kubeflow pipelines are used for workflow automation, creating a graph of functions and their specified parameters, inputs, and outputs.

You can chain the outputs and inputs of the pipeline steps, as illustrated below.

```
import kfp
from kfp import dsl
from mlrun import run_pipeline
from mlrun import run_function, deploy_function
```

```
@dsl.pipeline(
    name = 'job test',
    description = 'demonstrating mlrun usage'
```

(continues on next page)

(continued from previous page)

```
)
def job_pipeline(
    p1: int = 9
) -> None:
    """Define our pipeline.

    :param p1: A model parameter.
    """

    train = run_function('my-trainer',
                         handler='training',
                         params={'p1': p1},
                         outputs=['mymodel'])

    validate = run_function('my-trainer',
                            handler='validation',
                            inputs={'model': train.outputs['mymodel']},
                            outputs=['validation'])
```

Running the pipeline

Pipeline results are stored at the `artifact_path` location. The artifact path for workflows can be one of:

- The project's `artifact_path` (set by `project.spec.artifact_path = '<some path>'`). MLRun adds `/{{workflow.uid}}` to the path if it does not already include it.
- MLRun's default `artifact-path`, if set. MLRun adds `/{{workflow.uid}}'` to the path if it does not already include it.
- The `artifact_path` as passed to the specific call for `run()`, as shown below. In this case, MLRun does not modify the user-provided path.

If you want to customize the path, per workflow, use:

```
artifact_path = 'v3io:///users/admin/kfp/{{workflow.uid}}/'
```

```
arguments = {'p1': 8}
run_id = run_pipeline(job_pipeline, arguments, experiment='my-job', artifact_
↪path=artifact_path)
```

```
> 2021-01-24 00:09:46,670 [info] using in-cluster config.
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-01-24 00:09:46,940 [info] Pipeline run id=26ac4209-8505-47a3-b807-e9c51061bf0d,
↪ check UI or DB for progress
```

```
from mlrun import wait_for_pipeline_completion, get_run_db
wait_for_pipeline_completion(run_id)
db = get_run_db().list_runs(project=project_name, labels=f'workflow={run_id}').show()
```

```
<IPython.core.display.HTML object>
```

Viewing the pipeline on the dashboard (UI)

In the **Projects > Jobs and Workflows > Monitor Workflows** tab, press the workflow name to view a graph of the workflow. Press any step to open a pane with full details of the step: either the job's overview, inputs, artifacts, etc.; or the deploy / build function's overview, code, and log. The color of the step, after pressing, indicates the status. See the status description in the **Log** tab. The graph is refreshed while the pipeline is running.

Back to top

See also:

- *Automated Logging and MLOps with apply_mlrun()*

2.1.17 Real-time serving pipelines (graphs)

MLRun graphs enable building and running DAGs (directed acyclic graph). Graphs are composed of individual steps. The first graph element accepts an `Event` object, transform/process the event and pass the result to the next steps in the graph. The final result can be written out to some destination (file, DB, stream, etc.) or returned back to the caller (one of the graph steps can be marked with `.respond()`).

The serving graphs can be composed of *pre-defined graph steps*, block-type elements (model servers, routers, ensembles, data readers and writers, data engineering tasks, validators, etc.), *custom steps*, or from native python classes/functions. A graph can have data processing steps, model ensembles, model servers, post-processing, etc. (see the *Advanced Model Serving Graph Notebook Example*). Graphs can auto-scale and span multiple function containers (connected through streaming protocols).



Different steps can run on the same local function, or run on a remote function. You can call existing functions from the graph and reuse them from other graphs, as well as scale up and down different components individually.

Graphs can run inside your IDE or Notebook for test and simulation. Serving graphs are built on top of Nuclio (real-time serverless engine), MLRun Jobs, MLRun Storey (native Python async and stream processing engine), and other MLRun facilities.

In this section

Getting started

This example uses a custom class and custom function. See *custom steps* for more details.

In this section

Steps

The following code defines basic steps that illustrate building a graph. These steps are:

- **inc**: increments the value by 1.
- **mul**: multiplies the value by 2.
- **WithState**: class that increments an internal counter, prints an output, and adds the input value to the current counter.

```
# mlrun: start-code

def inc(x):
    return x + 1

def mul(x):
    return x * 2

class WithState:
    def __init__(self, name, context, init_val=0):
        self.name = name
        self.context = context
        self.counter = init_val

    def do(self, x):
        self.counter += 1
        print(f"Echo: {self.name}, x: {x}, counter: {self.counter}")
        return x + self.counter

# mlrun: end-code
```

Create a function

Now take the code above and create an MLRun function called `serving-graph`.

```
import mlrun
fn = mlrun.code_to_function("simple-graph", kind="serving", image="mlrun/mlrun")
graph = fn.set_topology("flow")
```

Build the graph

Use `graph.to()` to chain steps. Use `.respond()` to mark that the output of that step is returned to the caller (as an http response). By default the graph is async with no response.

```
graph.to(name="+1", handler='inc')\
     .to(name="*2", handler='mul')\
     .to(name="(X+counter)", class_name='WithState').respond()
```

```
<mlrun.serving.states.TaskStep at 0x7f821e504450>
```

Visualize the graph

Using the `plot` method, you can visualize the graph.

```
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f82294f2f90>
```

Test the function

Create a mock server and test the graph locally. Since this graph accepts a numeric value as the input, that value is provided in the `body` parameter.

```
server = fn.to_mock_server()
server.test(body=5)
```

```
Echo: (X+counter), x: 12, counter: 1
```

```
13
```

Run the function again. This time, the counter should be 2 and the output should be 14.

```
server.test(body=5)
```

```
Echo: (X+counter), x: 12, counter: 2
```

```
14
```

Deploy the function

Use the `deploy` method to deploy the function.

```
fn.deploy(project='basic-graph-demo')
```

```
> 2021-11-08 07:30:21,571 [info] Starting remote function deploy
2021-11-08 07:30:21  (info) Deploying function
2021-11-08 07:30:21  (info) Building
2021-11-08 07:30:21  (info) Staging files and preparing base images
2021-11-08 07:30:21  (info) Building processor image
2021-11-08 07:30:26  (info) Build complete
2021-11-08 07:30:31  (info) Function deploy complete
> 2021-11-08 07:30:31,785 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.
↪local:8080'], 'external_invocation_urls': ['basic-graph-demo-simple-graph-basic-
↪graph-demo.default-tenant.app.aganefaibuzg.iguazio-cd2.com/']}
```

```
'http://basic-graph-demo-simple-graph-basic-graph-demo.default-tenant.app.
↪aganefaibuzg.iguazio-cd2.com/'
```

Test the deployed function

Use the `invoke` method to call the function.

```
fn.invoke('', body=5)
```

```
> 2021-11-08 07:30:43,241 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.local:8080/'}
```

```
13
```

```
fn.invoke('', body=5)
```

```
> 2021-11-08 07:30:48,359 [info] invoking function: {'method': 'POST', 'path': 'http:/
↪/nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.local:8080/'}
```

```
14
```

Use cases

In this section

- *Data and feature engineering*
- *Example of Simple model serving router*
- *Example of Advanced data processing and serving ensemble*
- *Example of NLP processing pipeline with real-time streaming*

In addition to the examples in this section, see the:

- *Distributed (multi-function) pipeline example* that details how to run a pipeline that consists of multiple serverless functions (connected using streams).
- *Advanced Model Serving Graph Notebook Example* that illustrates the flow, task, model, and ensemble router states; building tasks from custom handlers; classes and storey components; using custom error handlers; testing graphs locally; deploying a graph as a real-time serverless function.
- MLRun demos repository for additional use cases and full end-to-end examples, including fraud prevention using the Iguazio feature store, a mask detection demo, and converting existing ML code to an MLRun project.

Data and feature engineering (using the feature store)

You can build a feature set transformation using serving graphs.

High-level transformation logic is automatically converted to real-time serverless processing engines that can read from any online or offline source, handle any type of structures or unstructured data, run complex computation graphs and native user code. Iguazio's solution uses a unique multi-model database, serving the computed features consistently through many different APIs and formats (like files, SQL queries, pandas, real-time REST APIs, time-series, streaming), resulting in better accuracy and simpler integration.

Read more in the *Feature Store Overview*, and *Feature set transformations*.

Example of a simple model serving router

Graphs are used for serving models with different transformations.

To deploy a serving function, you need to import or create the serving function, add models to it, and then deploy it.

```
import mlrun
# load the sklearn model serving function and add models to it
fn = mlrun.import_function('hub://v2_model_server')
fn.add_model("model1", model_path={model1-url})
fn.add_model("model2", model_path={model2-url})

# deploy the function to the cluster
fn.deploy()

# test the live model endpoint
fn.invoke('/v2/models/model1/infer', body={"inputs": [5]})
```

The Serving function supports the same protocol used in KFServing V2 and Triton Serving framework. To invoke the model, to use following url: `<function-host>/v2/models/model1/infer`.

See the ***serving protocol specification*** for details.

Note: Model url is either an MLRun model store object (starts with `store://`) or URL of a model directory (in NFS, s3, v3io, azure, for example `s3://{bucket}/{model-dir}`). Note that credentials might need to be added to the serving function via environment variables or MLRun secrets.

See the **scikit-learn classifier example**, which explains how to create/log MLRun models.

Writing your own serving class

You can implement your own model serving or data processing classes. All you need to do is:

1. Inherit the base model serving class.
2. Add your implementation for model `load()` (download the model file(s) and load the model into memory).
3. `predict()` (accept the request payload and return the prediction/inference results).

You can override additional methods: `preprocess`, `validate`, `postprocess`, `explain`. You can add custom API endpoints by adding the method `op_xx(event)` (which can be invoked by calling the `<model-url>/xx`, where operation = xx). See model class API.

For an example of writing the minimal serving functions, see Minimal sklearn serving function example.

See the full V2 Model Server (SKLearn) example that tests one or more classifier models against a held-out dataset.

Example of advanced data processing and serving ensemble

MLRun Serving graphs can host advanced pipelines that handle event/data processing, ML functionality, or any custom task. The following example demonstrates an asynchronous pipeline that pre-processes data, passes the data into a model ensemble, and finishes off with post processing.

For a complete example, see the *Advanced graph example notebook*.

Create a new function of type serving from code and set the graph topology to `async flow`.

```
import mlrun
function = mlrun.code_to_function("advanced", filename="demo.py",
                                  kind="serving", image="mlrun/mlrun",
                                  requirements=['storey'])
graph = function.set_topology("flow", engine="async")
```

Build and connect the graph (DAG) using the custom function and classes and plot the result. Add steps using the `step.to()` method (adds a new step after the current one), or using the `graph.add_step()` method.

If you want the error from the graph or the step to be fed into a specific step (catcher), use the `graph.error_handler()` (apply to all steps) or `step.error_handler()` (apply to a specific step).

Specify which step is the responder (returns the HTTP response) using the `step.respond()` method. If the responder is not specified, the graph is non-blocking.

```
# use built-in storey class or our custom Echo class to create and link Task steps
graph.to("storey.Extend", name="enrich", _fn='({"tag": "something"})') \
     .to(class_name="Echo", name="pre-process", some_arg='abc').error_handler("catcher
↪")

# add an Ensemble router with two child models (routes), the "*" prefix mark it is a␣
↪router class
router = graph.add_step("*mlrun.serving.VotingEnsemble", name="ensemble", after="pre-
↪process")
router.add_route("m1", class_name="ClassifierModel", model_path=path1)
router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# add the final step (after the router) which handles post processing and respond to␣
↪the client
graph.add_step(class_name="Echo", name="final", after="ensemble").respond()
```

(continues on next page)

(continued from previous page)

```
# add error handling step, run only when/if the "pre-process" step fail (keep after="
↪")
graph.add_step(handler="error_catcher", name="catcher", full_event=True, after="")

# plot the graph (using Graphviz) and run a test
graph.plot(rankdir='LR')
```

Create a mock (test) server, and run a test. Use `wait_for_completion()` to wait for the async event loop to complete.

```
server = function.to_mock_server()
resp = server.test("/v2/models/m2/infer", body={"inputs": data})
server.wait_for_completion()
```

And deploy the graph as a real-time Nuclio serverless function with one command:

```
function.deploy()
```

Note: If you test a Nuclio function that has a serving graph with the async engine via the Nuclio UI, the UI might not display the logs in the output.

Example of an NLP processing pipeline with real-time streaming

In some cases it's useful to split your processing to multiple functions and use streaming protocols to connect those functions. In this example the data processing is in the first function/container and the NLP processing is in the second function. In this example the GPU contained in the second function.

See the *full notebook example*.

```
# define a new real-time serving function (from code) with an async graph
fn = mlrun.code_to_function("multi-func", filename="./data_prep.py", kind="serving",␣
↪image='mlrun/mlrun')
graph = fn.set_topology("flow", engine="async")

# define the graph steps (DAG)
graph.to(name="load_url", handler="load_url")\
     .to(name="to_paragraphs", handler="to_paragraphs")\
     .to("storey.FlatMap", "flatten_paragraphs", _fn="(event)")\
     .to(">>", "q1", path=internal_stream)\
     .to(name="nlp", class_name="ApplyNLP", function="enrich")\
     .to(name="extract_entities", handler="extract_entities", function="enrich")\
     .to(name="enrich_entities", handler="enrich_entities", function="enrich")\
     .to("storey.FlatMap", "flatten_entities", _fn="(event)", function="enrich")\
     .to(name="printer", handler="myprint", function="enrich")\
     .to(">>", "output_stream", path=out_stream)

# specify the "enrich" child function, add extra package requirements
child = fn.add_child_function('enrich', './nlp.py', 'mlrun/mlrun')
child.spec.build.commands = ["python -m pip install spacy",
                             "python -m spacy download en_core_web_sm"]
graph.plot()
```

Currently queues only support iguazio v3io stream, Kafka support will soon be added.

Graph concepts and state machine

A graph is composed of the following:

- Step: A Step runs a function or class handler or a REST API call. MLRun comes with a list of *pre-built steps* that include data manipulation, readers, writers and model serving. You can also write your own steps using standard Python functions or custom functions/classes, or can be a external REST API (the special `$remote` class).
- Router: A special type of step is a router with routing logic and multiple child routes/models. The basic routing logic is to route to the child routes based on the `event.path`. More advanced or custom routing can be used, for example, the ensemble router sends the event to all child routes in parallel, aggregates the result and responds.
- Queue: A queue or stream that accepts data from one or more source steps and publishes to one or more output steps. Queues are best used to connect independent functions/containers. Queues can run in-memory or be implemented using a stream, which allows it to span processes/containers.

The Graph server has two modes of operation (topologies):

- Router topology (default): A minimal configuration with a single router and child tasks/routes. This can be used for simple model serving or single hop configurations.
- Flow topology: A full graph/DAG. The flow topology is implemented using two engines: async (the default) is based on Storey and asynchronous event loop; and `sync`, which supports a simple sequence of steps.

This section presents:

The Event object

The Graph state machine accepts an Event object (similar to a Nuclio Event) and passes it along the pipeline. An Event object hosts the event `body` along with other attributes such as `path` (http request path), `method` (GET, POST, ..), and`id` (unique event ID).

In some cases the events represent a record with a unique `key`, which can be read/set through the `event.key`. Records have associated `event.time` that, by default, is the arrival time, but can also be set by a step.

The Task steps are called with the `event.body` by default. If a task step needs to read or set other event elements (key, path, time, ..) you should set the task `full_event` argument to `True`.

Task steps support optional `input_path` and `result_path` attributes that allow controlling which portion of the event is sent as input to the step, and where to update the returned result.

For example, for an event body `{"req": {"body": "x"}}`, `input_path="req.body"` and `result_path="resp"` the step gets `"x"` as the input. The output after the step is `{"req": {"body": "x"}, "resp": <step output>}`. Note that `input_path` and `result_path` do not work together with `full_event=True`.

The Context object

The step classes are initialized with a `context` object (when they have `context` in their `__init__` args). The context is used to pass data and for interfacing with system services. The context object has the following attributes and methods.

Attributes:

- **logger**: Central logger (Nuclio logger when running in Nuclio).
- **verbose**: True if in verbose/debug mode.
- **root**: The graph object.
- **current_function**: When running in a distributed graph, the current child function name.

Methods:

- **get_param(key, default=None)**: Get the graph parameter by key. Parameters are set at the serving function (e.g. `function.spec.parameters = {"param1": "x"}`).
- **get_secret(key)**: Get the value of a project/user secret.
- **get_store_resource(uri, use_cache=True)**: Get the mlrun store object (data item, artifact, model, feature set, feature vector).
- **get_remote_endpoint(name, external=False)**: Return the remote nuclio/serving function http(s) endpoint given its [project/]function-name[:tag].
- **Response(headers=None, body=None, content_type=None, status_code=200)**: Create a nuclio response object, for returning detailed http responses.

Example, using the context:

```
 if self.context.verbose:
        self.context.logger.info('my message', some_arg='text')
    x = self.context.get_param('x', 0)
```

Topology

Router

Once you have a serving function, you need to choose the graph topology. The default is `router` topology. With the `router` topology you can specify different machine learning models. Each model has a logical name. This name is used to route to the correct model when calling the serving function.

```
from sklearn.datasets import load_iris

# set the topology/router
graph = fn.set_topology("router")

# Add the model
fn.add_model("model1", class_name="ClassifierModel", model_path="https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl")

# Add additional models
#fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
```

(continues on next page)

(continued from previous page)

```
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:36,925 [info] model model1 was loaded
> 2021-11-02 04:18:36,926 [info] Initializing endpoint records
> 2021-11-02 04:18:36,965 [info] Loaded ['model1']
{'id': '6bd11e864805484ea888f58e478d1f91', 'model_name': 'model1', 'outputs': [0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 2]}
```

Flow

Using the `flow` topology, you can specify tasks, which typically manipulate the data. The most common scenario is pre-processing of data prior to the model execution.

Note: Once the topology is set, you cannot change an existing function topology.

In this topology, you build and connect the graph (DAG) by adding steps using the `step.to()` method, or by using the `graph.add_step()` method.

> The `step.to()` is typically used to chain steps together. `graph.add_step` can add steps anywhere on the graph and has `before` and `after` parameters to specify the location of the step.

```
fn2 = mlrun.code_to_function("serving_example_flow",
                             kind="serving",
                             image="mlrun/mlrun")

graph2 = fn2.set_topology("flow")

graph2_enrich = graph2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})
↪')

# add an Ensemble router with two child models (routes)
router = graph2.add_step(mlrun.serving.ModelRouter(), name="router", after="enrich")
router.add_route("m1", class_name="ClassifierModel", model_path='https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl')
router.respond()

# Add additional models
#router.add_route("m2", class_name="ClassifierModel", model_path=path2)

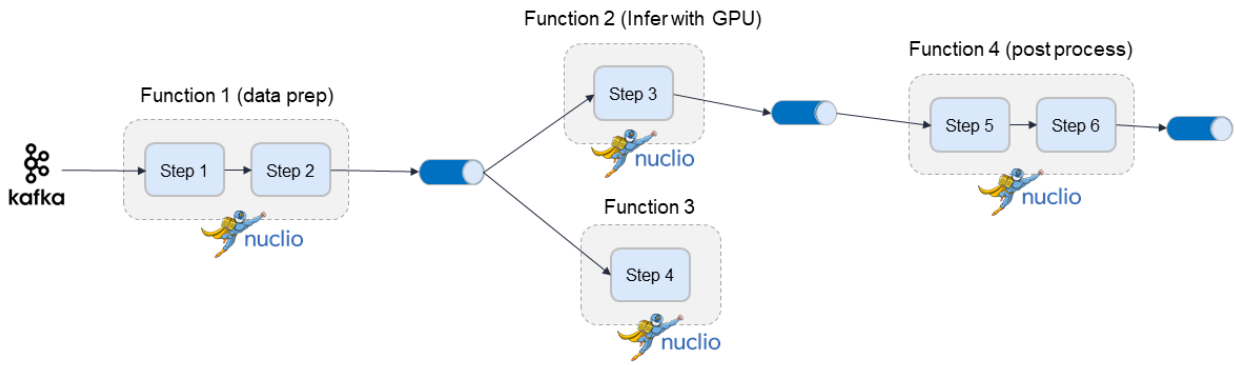
# plot the graph (using Graphviz)
graph2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fd46e4dda50>
```

```
fn2_server = fn2.to_mock_server()

result = fn2_server.test("/v2/models/m1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:42,142 [info] model m1 was loaded
> 2021-11-02 04:18:42,142 [info] Initializing endpoint records
> 2021-11-02 04:18:42,183 [info] Loaded ['m1']
{'id': 'f713fd7eedeb431eba101b13c53a15b5'}
```

Building distributed graphs

Graphs can be hosted by a single function (using zero to n containers), or span multiple functions where each function can have its own container image and resources (replicas, GPUs/CPUs, volumes, etc.). It has a `root` function, which is where you configure triggers (http, incoming stream, cron, ..), and optional downstream child functions.

You can specify the `function` attribute in `Task` or `Router` steps. This indicates where this step should run. When the `function` attribute is not specified it runs on the root function. `function="*"` means the step can run in any of the child functions.

Steps on different functions should be connected using a `Queue` step (a stream).

Adding a child function:

```
```python
fn.add_child_function('enrich',
                      './entity_extraction.ipynb',
                      image='mlrun/mlrun',
                      requirements=["storey", "sklearn"])
```
```

See a full example with child functions.

A distributed graph looks like this:



Error handling

Graph steps might raise an exception. If you want to have an error handling flow, you can specify an exception handling step/branch that is triggered on error. The error handler step receives the event that entered the failed step, with two extra attributes: `event.origin_state` indicates the name of the failed step; and `event.error` holds the error string.

Use the `graph.error_handler()` (apply to all steps) or `step.error_handler()` (apply to a specific step) if you want the error from the graph or the step to be fed into a specific step (catcher).

Example of setting an error catcher per step:

```
graph.add_step("MyClass", name="my-class", after="pre-process").error_handler("catcher
↪")
graph.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

Note: Additional steps can follow the `catcher` step.

Using the example in Getting started with model serving, you can add an error handler as follows:

```
graph2_enrich.error_handler("catcher")
graph2.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

```
<mlrun.serving.states.TaskStep at 0x7fd46e557750>
```

Now, display the graph again:

```
graph2.plot(rankdir='LR')
```

```
<mlrun.serving.states.TaskStep at 0x7fd46e557750>
```

Exception stream

The graph errors/exceptions can be pushed into a special error stream. This is very convenient in the case of distributed and production graphs.

To set the exception stream address (using v3io streams uri):

```
fn_preprocess2.spec.error_stream = err_stream
```

Writing custom steps

The Graph executes built-in task classes, or task classes and functions that you implement. The task parameters include the following:

- `class_name` (str): the relative or absolute class name.
- `handler` (str): the function handler (if class_name is not specified it is the function handler).
- `**class_args`: a set of class `__init__` arguments.

For example, see the following simple `echo` class:

```
import mlrun
```

```
# mlrun: start
```

```
# echo class, custom class example
class Echo:
    def __init__(self, context, name=None, **kw):
        self.context = context
        self.name = name
        self.kw = kw

    def do(self, x):
        print("Echo:", self.name, x)
        return x
```

```
# mlrun: end
```

Test the graph: first convert the code to function, and then add the step to the graph:

```
fn_echo = mlrun.code_to_function("echo_function", kind="serving", image="mlrun/mlrun")

graph_echo = fn_echo.set_topology("flow")

graph_echo.to(class_name="Echo", name="pre-process", some_arg='abc')

graph_echo.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f2d73584c90>
```

Create a mock server to test this locally:

```
echo_server = fn_echo.to_mock_server(current_function="*")

result = echo_server.test("", {"inputs": 123})

print(result)
```

```
{'id': '97397ea412334afdb5e4cb7d7c2e6dd3'}
Echo: pre-process {'inputs': 123}
```

For more information, see the *Advanced Graph Notebook Example*

You can use any Python function by specifying the handler name (e.g. `handler=json.dumps`). The function is triggered with the `event.body` as the first argument, and its result is passed to the next step.

Alternatively, you can use classes that can also store some step/configuration and separate the one time init logic from the per event logic. The classes are initialized with the `class_args`. If the class init args contain `context` or `name`, they are initialized with the *graph context* and the step name.

By default, the `class_name` and handler specify a class/function name in the `globals()` (i.e. this module). Alternatively, those can be full paths to the class (module.submodule.class), e.g. `storey.WriteToParquet`. You can also pass the module as an argument to functions such as `function.to_mock_server(namespace=module)`. In this case the class or handler names are also searched in the provided module.

When using classes the class event handler is invoked on every event with the `event.body`. If the Task step `full_event` parameter is set to `True` the handler is invoked and returns the full `event` object. If the class event

handler is not specified, it invokes the class `do()` method.

If you need to implement async behavior, then subclass `storey.MapClass`.

Built-in steps

MlRun provides you with many built-in stapes that you can use when building your graph.

Click on the step names in the following sections to see the full usage.

- *Base Operators*
- *Data Transformations*
- *External IO and data enrichment*
- *Sources*
- *Targets*
- *Models*
- *Routers*
- *Other*

Base Operators

Class name	Description
storey.transformations.Batch	Batches events. This step emits a batch every `max_events` events, or when `timeout` seconds have passed since the first event in the batch was received.
storey.transformations.Choice	Redirects each input element into one of the multiple downstreams.
storey.Extend	Adds fields to each incoming event.
storey.transformations.Filter	Filters events based on a user-provided function.
storey.transformations.FlatMap	Maps, or transforms, each incoming event into any number of events.
storey.steps.Flatten	Flatten is equivalent to FlatMap(lambda x: x).
storey.transformations.ForEach	Applies the given function on each event in the stream, and passes the original event downstream.
storey.transformations.MapClass	Similar to Map, but instead of a function argument, this class should be extended and its do() method overridden.
storey.transformations.MapWithState	Maps, or transforms, incoming events using a stateful user-provided function, and an initial state, which can be a database table.
storey.transformations.Partition	Partitions events by calling a predicate function on each event. Each processed event results in a Partitioned namedtuple of (left=Optional[Event], right=Optional[Event]).
storey.Reduce	Reduces incoming events into a single value that is returned upon the successful termination of the flow.
storey.transformations.SampleWindow	Emits a single event in a window of `window_size` events, in accordance with `emit_period` and `emit_before_termination`.

Data Transformations

Class name	Description
storey.AggregateByKey	Aggregates the data into the table object provided for later persistence, and outputs an event enriched with the requested aggregation features.
DateExtractor	Extract a date-time component.
mlrun.feature_store.Imputer	Replace None values with default values.
mlrun.feature_store.MapValues	Map column values to new values.
mlrun.feature_store.OneHotEncoder	Create new binary fields, one per category (one hot encoded).
mlrun.feature_store.SetEventMetadata	Set the event metadata (id, key, timestamp) from the event body.

External IO and data enrichment

Class name	Description
BatchHttpRequests	A class for calling remote endpoints in parallel.
mlrun.datastore.DataItem	Data input/output class abstracting access to various local/remote data sources.
storey.transformations.JoinWithTable	Joins each event with data from the given table.
JoinWithV3IOTable	Joins each event with a V3IO table. Used for event augmentation.
QueryByKey	Similar to to AggregateByKey, but this step is for serving only and does not aggregate the event.
RemoteStep	Class for calling remote endpoints.
storey.transformations.SendToHttp	Joins each event with data from any HTTP source. Used for event augmentation.

Sources

Class name	Description
mlrun.datastore.BigQuerySource	Reads Google BigQuery query results as input source for a flow.
mlrun.datastore.CSVSource	Reads a CSV file as input source for a flow.
DataframeSource	Reads data frame as input source for a flow.
mlrun.datastore.HttpSource	
mlrun.datastore.KafkaSource	Sets the kafka source for the flow.
mlrun.datastore.ParquetSource	Reads the Parquet file/dir as the input source for a flow.
mlrun.datastore.StreamSource	Sets the stream source for the flow. If the stream doesn't exist it creates it.

Targets

Class name	Description
mlrun.datastore.CSVTarget	Writes events to a CSV file.
mlrun.datastore.NoSqlTarget	Persists the data in *table* to its associated storage by key.
mlrun.datastore.ParquetTarget	The Parquet target storage driver, used to materialize feature set/vector data into parquet files.
mlrun.datastore.StreamTarget	Writes all incoming events into a V3IO stream.
storey.transformations.ToDataFrame	Create pandas data frame from events. Can appear in the middle of the flow, as opposed to ReduceToDataFrame.
mlrun.datastore.TSBDTarget	

Models

Class name	Description
mlrun.frameworks.onnx.ONNXModelServer	A model serving class for serving ONYX Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.pytorch.PyTorchModelServer	A model serving class for serving PyTorch Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.sklearn.SklearnModelServer	A model serving class for serving Sklearn Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.tf_keras.TFKerasModelServer	A model serving class for serving TFKeras Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.xgboost.XGBModelServer	A model serving class for serving XGB Models. A sub-class of the V2ModelServer class.

Routers

Class name	Description
mlrun.serving.EnrichmentModelRouter	Auto enrich the request with data from the feature store. The router input accepts a list of inference requests (each request can be a dict or a list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).
mlrun.serving.EnrichmentVotingEnsemble	Auto enrich the request with data from the feature store. The router input accepts a list of inference requests (each request can be a dict or a list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).
mlrun.serving.ModelRouter	Basic model router, for calling different models per each model path.
mlrun.serving.VotingEnsemble	An ensemble machine learning model that combines the prediction of several models.

Other

Class name	Description
mlrun.feature_store.FeaturesetValidator	Validate feature values according to the feature set validation policy.
ReduceToDataFrame	Builds a pandas DataFrame from events and returns that DataFrame on flow termination.

Demos and Tutorials

Read these tutorials to get an even better understanding of serving graphs.

Distributed (Multi-function) Pipeline Example

This example demonstrates how to run a pipeline that consists of multiple serverless functions (connected using streams).

In the pipeline example the request contains the a URL of a file. It loads the content of the file and breaks it into paragraphs (using the FlatMap class), and pushes the results to a queue/stream. The second function picks up the paragraphs and runs the NLP flow to extract the entities and push the results to the output stream.

Setting the stream URLs for the internal queue, the final output and error/exceptions stream:

```
streams_prefix = "v3io:///users/admin/"
internal_stream = streams_prefix + "in-stream"
out_stream = streams_prefix + "out-stream"
err_stream = streams_prefix + "err-stream"
```

```
# set up the environment
import mlrun
mlrun.set_environment(project="pipe")
```

```
> 2021-05-03 14:28:39,987 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-05-03 14:28:43,801 [warning] Unable to parse server or client version.␣
↪Assuming compatible: {'server_version': '0.6.3-rc4', 'client_version': 'unstable'}
```

```
('pipe', '/v3io/projects/{{run.project}}/artifacts')
```

```
# uncomment to install spacy requirements locally
# !pip install spacy
# !python -m spacy download en_core_web_sm
```

In this example

- *Create the pipeline*
- *Test the pipeline locally*
- *Deploy to the cluster*

Create the pipeline

The pipeline consists of two functions: data-prep and NLP. Each one has different package dependencies.

Create a file with data-prep graph steps:

```
%%writefile data_prep.py
import mlrun
import json

# load struct from a json file (event points to the url)
def load_url(event):
    url = event["url"]
    data = mlrun.get_object(url).decode("utf-8")
    return {"url": url, "doc": json.loads(data)}

def to_paragraphs(event):
    paragraphs = []
    url = event["url"]
    for i, paragraph in enumerate(event["doc"]):
        paragraphs.append(
            {"url": url, "paragraph_id": i, "paragraph": paragraph}
        )
    return paragraphs
```

```
Overwriting data_prep.py
```

Create a file with NLP graph steps (use spacy):

```
%%writefile nlp.py
import json
import spacy

def myprint(x):
    print(x)
    return x

class ApplyNLP:
    def __init__(self, context=None, spacy_dict="en_core_web_sm"):

        self.nlp = spacy.load(spacy_dict)

    def do(self, paragraph: dict):
        tokenized_paragraphs = []
        if isinstance(paragraph, (str, bytes)):
            paragraph = json.loads(paragraph)
        tokenized = {
            "url": paragraph["url"],
            "paragraph_id": paragraph["paragraph_id"],
            "tokens": self.nlp(paragraph["paragraph"]),
        }
        tokenized_paragraphs.append(tokenized)

        return tokenized_paragraphs

def extract_entities(tokens):
    paragraph_entities = []
```

(continues on next page)

(continued from previous page)

```
    for token in tokens:
        entities = token["tokens"].ents
        for entity in entities:
            paragraph_entities.append(
                {
                    "url": token["url"],
                    "paragraph_id": token["paragraph_id"],
                    "entity": entity.ents,
                }
            )
    return paragraph_entities

def enrich_entities(entities):
    enriched_entities = []
    for entity in entities:
        enriched_entities.append(
            {
                "url": entity["url"],
                "paragraph_id": entity["paragraph_id"],
                "entity_text": entity["entity"][0].text,
                "entity_start_char": entity["entity"][0].start_char,
                "entity_end_char": entity["entity"][0].end_char,
                "entity_label": entity["entity"][0].label_,
            }
        )
    return enriched_entities
```

```
Overwriting nlp.py
```

Build and show the graph:

Create the master function ("multi-func") with the `data_prep.py` source and an async graph topology. Add a pipeline of steps made of custom python handlers, classes and built-in classes (like `storey.FlatMap`).

The pipeline runs across two functions which are connected by a queue/stream (q1). Use the `function=` to specify which function runs the specified step. End the flow with writing to the output stream.

```
# define a new real-time serving function (from code) with an async graph
fn = mlrun.code_to_function("multi-func", filename="./data_prep.py", kind="serving",␣
↪image='mlrun/mlrun')
graph = fn.set_topology("flow", engine="async")

# define the graph steps (DAG)
graph.to(name="load_url", handler="load_url")\
     .to(name="to_paragraphs", handler="to_paragraphs")\
     .to("storey.FlatMap", "flatten_paragraphs", _fn="(event)")\
     .to(">>", "q1", path=internal_stream)\
     .to(name="nlp", class_name="ApplyNLP", function="enrich")\
     .to(name="extract_entities", handler="extract_entities", function="enrich")\
     .to(name="enrich_entities", handler="enrich_entities", function="enrich")\
     .to("storey.FlatMap", "flatten_entities", _fn="(event)", function="enrich")\
     .to(name="printer", handler="myprint", function="enrich")\
     .to(">>", "output_stream", path=out_stream)
```

```
<mlrun.serving.states.QueueState at 0x7f9e618f9910>
```

```
# specify the "enrich" child function, add extra package requirements
child = fn.add_child_function('enrich', './nlp.py', 'mlrun/mlrun')
child.spec.build.commands = ["python -m pip install spacy",
                             "python -m spacy download en_core_web_sm"]
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f9dd5dbed90>
```

Test the pipeline locally

Create an input file:

```
%%writefile in.json
["Born and raised in Queens, New York City, Trump attended Fordham University for two␣
↪years and received a bachelor's degree in economics from the Wharton School of the␣
↪University of Pennsylvania. He became president of his father Fred Trump's real␣
↪estate business in 1971, renamed it The Trump Organization, and expanded its␣
↪operations to building or renovating skyscrapers, hotels, casinos, and golf courses.
↪ Trump later started various side ventures, mostly by licensing his name. Trump and␣
↪his businesses have been involved in more than 4,000 state and federal legal␣
↪actions, including six bankruptcies. He owned the Miss Universe brand of beauty␣
↪pageants from 1996 to 2015, and produced and hosted the reality television series␣
↪The Apprentice from 2004 to 2015.",
 "Trump's political positions have been described as populist, protectionist,␣
↪isolationist, and nationalist. He entered the 2016 presidential race as a␣
↪Republican and was elected in a surprise electoral college victory over Democratic␣
↪nominee Hillary Clinton while losing the popular vote.[a] He became the oldest␣
↪first-term U.S. president[b] and the first without prior military or government␣
↪service. His election and policies have sparked numerous protests. Trump has made␣
↪many false or misleading statements during his campaign and presidency. The␣
↪statements have been documented by fact-checkers, and the media have widely␣
↪described the phenomenon as unprecedented in American politics. Many of his␣
↪comments and actions have been characterized as racially charged or racist."]
```

```
Overwriting in.json
```

Create a mock server (simulator) and test:

```
# tuggle verbosity if needed
fn.verbose = False
```

```
to
# create a mock server (simulator), specify to simulate all the functions in the␣
↪pipeline ("*")
server = fn.to_mock_server(current_function="*")
```

```
# push a sample request into the pipeline and see the results print out (by the␣
↪printer step)
resp = server.test(body={"url": "in.json"})
```

```
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Queens', 'entity_start_char':␣
↪19, 'entity_end_char': 25, 'entity_label': 'GPE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'New York City', 'entity_start_
↪char': 27, 'entity_end_char': 40, 'entity_label': 'GPE'}
```

(continues on next page)

(continued from previous page)

```
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Trump', 'entity_start_char': 42,
↪ 'entity_end_char': 47, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Fordham University', 'entity_
↪start_char': 57, 'entity_end_char': 75, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'two years', 'entity_start_char
↪': 80, 'entity_end_char': 89, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'the Wharton School of the␣
↪University of Pennsylvania', 'entity_start_char': 141, 'entity_end_char': 193,
↪'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Fred Trump', 'entity_start_char
↪': 229, 'entity_end_char': 239, 'entity_label': 'PERSON'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '1971', 'entity_start_char': 266,
↪ 'entity_end_char': 270, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'The Trump Organization',
↪'entity_start_char': 283, 'entity_end_char': 305, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'more than 4,000', 'entity_start_
↪char': 529, 'entity_end_char': 544, 'entity_label': 'CARDINAL'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'six', 'entity_start_char': 588,
↪'entity_end_char': 591, 'entity_label': 'CARDINAL'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Universe', 'entity_start_char':␣
↪624, 'entity_end_char': 632, 'entity_label': 'PERSON'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '1996 to 2015', 'entity_start_
↪char': 663, 'entity_end_char': 675, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'The Apprentice', 'entity_start_
↪char': 731, 'entity_end_char': 745, 'entity_label': 'WORK_OF_ART'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '2004 to 2015', 'entity_start_
↪char': 751, 'entity_end_char': 763, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Trump', 'entity_start_char': 0,
↪'entity_end_char': 5, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': '2016', 'entity_start_char': 122,
↪ 'entity_end_char': 126, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Republican', 'entity_start_char
↪': 150, 'entity_end_char': 160, 'entity_label': 'NORP'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Democratic', 'entity_start_char
↪': 222, 'entity_end_char': 232, 'entity_label': 'NORP'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Hillary Clinton', 'entity_start_
↪char': 241, 'entity_end_char': 256, 'entity_label': 'PERSON'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'first', 'entity_start_char':␣
↪312, 'entity_end_char': 317, 'entity_label': 'ORDINAL'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'U.S.', 'entity_start_char': 323,
↪ 'entity_end_char': 327, 'entity_label': 'GPE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'first', 'entity_start_char':␣
↪349, 'entity_end_char': 354, 'entity_label': 'ORDINAL'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'American', 'entity_start_char':␣
↪671, 'entity_end_char': 679, 'entity_label': 'NORP'}
```

```
server.wait_for_completion()
```

Deploy to the cluster

```
# add credentials to the data/streams
fn.apply(mlrun.platforms.v3io_cred())
child.apply(mlrun.platforms.v3io_cred())

# specify the error stream (to store exceptions from the functions)
fn.spec.error_stream = err_stream

# deploy as a set of serverless functions
fn.deploy()
```

```
> 2021-05-03 14:33:55,400 [info] deploy child function enrich ...
> 2021-05-03 14:33:55,427 [info] Starting remote function deploy
2021-05-03 14:33:55  (info) Deploying function
2021-05-03 14:33:55  (info) Building
2021-05-03 14:33:55  (info) Staging files and preparing base images
2021-05-03 14:33:55  (info) Building processor image
2021-05-03 14:34:02  (info) Build complete
2021-05-03 14:34:08  (info) Function deploy complete
> 2021-05-03 14:34:09,232 [info] function deployed, address=default-tenant.app.yh30.
↪iguazio-c0.com:32356
> 2021-05-03 14:34:09,233 [info] deploy root function multi-func ...
> 2021-05-03 14:34:09,234 [info] Starting remote function deploy
2021-05-03 14:34:09  (info) Deploying function
2021-05-03 14:34:09  (info) Building
2021-05-03 14:34:09  (info) Staging files and preparing base images
2021-05-03 14:34:09  (info) Building processor image
2021-05-03 14:34:16  (info) Build complete
2021-05-03 14:34:22  (info) Function deploy complete
> 2021-05-03 14:34:22,891 [info] function deployed, address=default-tenant.app.yh30.
↪iguazio-c0.com:32046
```

```
'http://default-tenant.app.yh30.iguazio-c0.com:32046'
```

Listen on the output stream

You can use the SDK or CLI to listen on the output stream. Listening should be done in a separate console/notebook. Run:

```
mlrun watch-stream v3io:///users/admin/out-stream -j
```

or use the SDK:

```
from mlrun.platforms import watch_stream
watch_stream("v3io:///users/admin/out-stream", is_json=True)
```

Test the live function:

Note: The url must be a valid path to the input file.

```
fn.invoke('', body={"url": "v3io:///users/admin/pipe/in.json"})
```

```
{'id': '79354e45-a158-405f-811c-976e9cf4ab5e'}
```

Advanced Model Serving Graph - Notebook Example

This example demonstrates how to use MLRun serving graphs and their advanced functionality including:

- Use of flow, task, model, and ensemble router states
- Build tasks from custom handlers, classes and storey components
- Use custom error handlers
- Test graphs locally
- Deploy the graph as a real-time serverless functions

In this example

- *Define functions and classes used in the graph*
- *Create a new serving function and graph*
- *Test the function locally*
- *Deploy the graph as a real-time serverless function*

Define functions and classes used in the graph

```
from cloudpickle import load
from typing import List
from sklearn.datasets import load_iris
import numpy as np

# model serving class example
class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()

# echo class, custom class example
class Echo:
    def __init__(self, context, name=None, **kw):
        self.context = context
        self.name = name
        self.kw = kw

    def do(self, x):
        print("Echo:", self.name, x)
        return x

# error echo function, demo catching error and using custom function
def error_catcher(x):
        x.body = {"body": x.body, "origin_state": x.origin_state, "error": x.error}
        print("EchoError:", x)
        return None
```

```
# mark the end of the code section, DO NOT REMOVE !
# mlrun: end-code
```

Create a new serving function and graph

Use `code_to_function` to convert the above code into a serving function object and initialize a graph with async flow topology.

```
function = mlrun.code_to_function("advanced", kind="serving",
                                  image="mlrun/mlrun",
                                  requirements=['storey'])
graph = function.set_topology("flow", engine="async")
#function.verbose = True
```

Specify the sklearn models that are used in the ensemble.

```
models_path = 'https://s3.wasabisys.com/iguazio/models/iris/model.pkl'
path1 = models_path
path2 = models_path
```

Build and connect the graph (DAG) using the custom function and classes and plot the result. Add states using the `state.to()` method (adds a new state after the current one), or using the `graph.add_step()` method.

Use the `graph.error_handler()` (apply to all states) or `state.error_handler()` (apply to a specific state) if you want the error from the graph or the state to be fed into a specific state (catcher).

You can specify which state is the responder (returns the HTTP response) using the `state.respond()` method. If you don't specify the responder, the graph is non-blocking.

```
# use built-in storey class or our custom Echo class to create and link Task states
graph.to("storey.Extend", name="enrich", _fn='({"tag": "something"})') \
     .to(class_name="Echo", name="pre-process", some_arg='abc').error_handler("catcher
↪")

# add an Ensemble router with two child models (routes). The "*" prefix mark it is a␣
↪router class
router = graph.add_step("*mlrun.serving.VotingEnsemble", name="ensemble", after="pre-
↪process")
router.add_route("m1", class_name="ClassifierModel", model_path=path1)
router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# add the final step (after the router) that handles post processing and responds to␣
↪the client
graph.add_step(class_name="Echo", name="final", after="ensemble").respond()

# add error handling state, run only when/if the "pre-process" state fails (keep␣
↪after="")
graph.add_step(handler="error_catcher", name="catcher", full_event=True, after="")

# plot the graph (using Graphviz) and run a test
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fe03f6941d0>
```

Test the function locally

Create a test set.

```
import random
iris = load_iris()
x = random.sample(iris['data'].tolist(), 5)
```

Create a mock server (simulator) and test the graph with the test data.

> Note: The model and router objects support a common serving protocol API, see the *protocol and API section*.

```
server = function.to_mock_server()
resp = server.test("/v2/models/infer", body={"inputs": x})
server.wait_for_completion()
resp
```

```
> 2021-01-09 22:49:26,365 [info] model m1 was loaded
> 2021-01-09 22:49:26,493 [info] model m2 was loaded
> 2021-01-09 22:49:26,494 [info] Loaded ['m1', 'm2']
Echo: pre-process {'inputs': [[6.9, 3.2, 5.7, 2.3], [6.4, 2.7, 5.3, 1.9], [4.9, 3.1,␣
↪1.5, 0.1], [7.3, 2.9, 6.3, 1.8], [5.4, 3.7, 1.5, 0.2]], 'tag': 'something'}
Echo: final {'model_name': 'ensemble', 'outputs': [2, 2, 0, 2, 0], 'id':
↪'0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

```
{'model_name': 'ensemble',
 'outputs': [2, 2, 0, 2, 0],
 'id': '0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

Deploy the graph as a real-time serverless function

```
function.deploy()
```

```
> 2021-01-09 22:49:40,088 [info] Starting remote function deploy
2021-01-09 22:49:40  (info) Deploying function
2021-01-09 22:49:40  (info) Building
2021-01-09 22:49:40  (info) Staging files and preparing base images
2021-01-09 22:49:40  (info) Building processor image
2021-01-09 22:49:41  (info) Build complete
2021-01-09 22:49:47  (info) Function deploy complete
> 2021-01-09 22:49:48,422 [info] function deployed, address=default-tenant.app.yh55.
↪iguazio-cd0.com:32222
```

```
'http://default-tenant.app.yh55.iguazio-cd0.com:32222'
```

Invoke the remote function using the test data

```
function.invoke("/v2/models/infer", body={"inputs": x})
```

```
{'model_name': 'ensemble',
 'outputs': [1, 2, 0, 0, 0],
 'id': '0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

See the MLRun demos repository for additional use cases and full end-to-end examples, including Fraud Prevention using the Iguazio feature store, a mask detection demo, and converting existing ML code to an MLRun project.

Serving graph high availability configuration

This figure illustrates a simplistic flow of an MLRun serving graph with remote invocation:

As explained in *MLRun Serving Graphs*, the serving graph is based on Nuclio functions.

In this section

- *Using Nuclio with stream triggers*
- *Consumer function configuration*
- *Remote function retry mechanism*
- *Configuration considerations*

Using Nuclio with stream triggers

Nuclio can use different trigger types. When used with stream triggers, such as Kafka and V3IO, it uses a consumer group to continue reading from the last processed offset on function restart. This provides the "at least once" semantics for stateless functions. However, if the function does have state, such as persisting a batch of events to storage (e.g. parquet files, database) or if the function performs additional processing of events after the function handler returns, then the flow can get into situations where events seem to be lost. The mechanism of Window ACK provides a solution for such stateful event processing.

With Window ACK, the consumer group's committed offset is delayed by one window, committing the offset at (processed event num – window). When the function restarts (for any reason including scale-up or scale-down), it starts consuming from this last committed point.

The size of the required Window ACK is based on the number of events that could be in processing when the function terminates. You can define a window ACK per trigger (Kafka, V3IO stream, etc.). When used with a serving graph, the appropriate Window ACK size depends on the graph structure and should be calculated accordingly. The following sections explain the relevant considerations.

Consumer function configuration

A consumer function is essentially a Nuclio function with a stream trigger. As part of the trigger, you can set a consumer group.

When the consumer function is part of a graph then the consumer function's number of replicas is derived from the number of shards and is therefore nonconfigurable. The same applies to the number of workers in each replica, which is set to 1 and is not configurable.

The consumer function has one buffer per worker holding the incoming events that were received by the worker and are waiting to be processed. Once this buffer is full, events need to be processed so that the function is able to receive more events. The buffer size is configurable and is key to the overall configuration.

The buffer should be as small as possible. There is a trade-off between the buffer size and the latency. A larger buffer has lower latency but increases the recovery time after a failure, due to the high number of records that need to be reprocessed. To set the buffer size:

```
function.spec.parameters["source_args"] = {"buffer_size":  1}
```

The default `buffer_size` is 8.

Remote function retry mechanism

The required processing time of a remote function varies, depending on the function. The system assumes a processing time in the order of seconds, which affects the default configurations. However, some functions require a longer processing time. You can configure the timeout on both the caller and on the remote, as appropriate for your functions.

When an event is sent to the remote function, and no response is received by the configured (or default) timeout, or an error 500 (the remote function failed), or error 502, 503, or 504 (the remote function is too busy to handle the request at this time) is received, the caller retries the request, using the platform's exponential retry backoff mechanism. If the number of caller retries reaches the configured maximum number of retries, the event is pushed to the exception stream, indicating that this `event` did not complete successfully. You can look at the exception stream to see the functions that did not complete successfully.

Remote-function caller configuration

In a simplistic flow these are the consumer function defaults:

- Maximum retries: The default is 6, which is equivalent to about 3-4 minutes if all of the related parameters are at their default values. If you expect that some cases will require a higher number, for example, a new node needs to be scaled up depending on your cloud vendor, the instance type, and the zone you are running in, you might want to increase the number of retries.
- Remote step http timeout: The time interval the caller waits for a response from the remote before retrying the request. This value is affected by the remote function processing time.
- Max in flight: The maximum number of requests that each caller worker can send in parallel to the remote function. If the caller has more than one worker, each worker has its own Max in flight.

To set Max in flight, timeout, and retries:

```
RemoteStep(name="remote_scale", ...,  max_in_flight=2, timeout=100,
retries=10)
```

Remote-function configuration

For the remote function, you can configure the following:

- Worker timeout: The maximum time interval, in seconds, an incoming request waits for an available worker. The worker timeout must be shorter than the gateway timeout. The default is 10.
- Gateway timeout: The maximum time interval, in seconds, the gateway waits for a response to a request. This determines when the ingress times out on a request. It must be slightly longer than the expected function processing time. The default is 60.

To set the buffer gateway timeout and worker timeout:

```
my_serving_func.with_http(gateway_timeout=125, worker_timeout=60)
```

Configuration considerations

The following figure zooms in on a single consumer and its workers and illustrates the various concepts and parameters that provide high availability, using a non-default configuration.

- Assume the processing time of the remote function is Pt, in seconds.
- `timeout`: Between <Pt+epsilon> and <Pt+`worker_timeout`>.
- Serving function
  - `gateway_timeout`: Pt+1 second (usually sufficient).
  - `worker_timeout`: The general rule is the greater of Pt/10 or 60 seconds. However, you should adjust the value according to your needs.
- `max_in_flight`: If the processing time is very high then `max_in_flight` should be low. Otherwise, there will be many retries.
- `ack_window_size`:
  - With 1 worker: The consumer `buffer_size`+`max_in_flight`, since it is per each shard and there is a single worker.
  - With >1 worker: The consumer (#workers x `buffer_size`)+`max_in_flight`

Make sure you thoroughly understand your serving graph and its functions before defining the `ack_window_size`. Its value depends on the entire graph flow. You need to understand which steps are parallel (branching) vs. sequential invocation. Another key aspect is that the number of workers affects the window size.

See the ack_window_size API and an example.

For example:

- If a graph includes: consumer -> remote r1 -> remote r2
  - The window should be the sum of: consumer's buffer size + MIF to r1 + MIF to r2.
- If a graph includes: calling to remote r1 and r2 in parallel
  - The window should be set to: consumer's buffer size + max (MIF to r1, MIF to r2).

2.1.18 Model serving pipelines

MLRun serving can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. MLRun serving supports complex and distributed graphs (see the *Distributed (Multi-function) Pipeline Example*), which can involve streaming, data/document/image processing, NLP, and model monitoring, etc. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere.

Simple model serving classes can be written in Python or be taken from a set of pre-developed ML/DL classes. The code can handle complex data, feature preparation, and binary data (such as images and video files). The Nuclio serving engine supports the full model-serving life cycle, including auto-generation of microservices, APIs, load balancing, model logging and monitoring, and configuration management.

In this section

Getting started with model serving

In this section

Serving Functions

To start using a serving graph, you first need a serving function. A serving function contains the serving class code to run the model and all the code necessary to run the tasks. MLRun comes with a wide library of tasks. If you use just those, you don't have to add any special code to the serving function, you just have to provide the code that runs the model. For more information about serving classes see *Creating a custom model serving class*.

For example, the following code is a basic model serving class:

```
# mlrun: start-code
```

```
from cloudpickle import load
from typing import List
import numpy as np

import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model(".pkl")
        self.model = load(open(model_file, "rb"))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body["inputs"])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

```
# mlrun: end-code
```

To obtain the serving function, use the `code_to_function` and specify `kind` to be `serving`.

```
fn = mlrun.code_to_function("serving_example",
                            kind="serving",
                            image="mlrun/mlrun")
```

Topology

Router

Once you have a serving function, you need to choose the graph topology. The default is `router` topology. With the `router` topology you can specify different machine learning models. Each model has a logical name. This name is used to route to the correct model when calling the serving function.

```
from sklearn.datasets import load_iris

# set the topology/router
graph = fn.set_topology("router")

# Add the model
fn.add_model("model1", class_name="ClassifierModel", model_path="https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl")

# Add additional models
#fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:36,925 [info] model model1 was loaded
> 2021-11-02 04:18:36,926 [info] Initializing endpoint records
> 2021-11-02 04:18:36,965 [info] Loaded ['model1']
{'id': '6bd11e864805484ea888f58e478d1f91', 'model_name': 'model1', 'outputs': [0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 2]}
```

Flow

You can use the `flow` topology to specify tasks, which typically manipulates the data. The most common scenario is pre-processing of data prior to the model execution.

Note: Once the topology is set, you cannot change an existing function toplogy.

In this topology, you build and connect the graph (DAG) by adding steps using the `step.to()` method, or by using the `graph.add_step()` method.

> The `step.to()` is typically used to chain steps together. `graph.add_step` can add steps anywhere on the graph and has `before` and `after` parameters to specify the location of the step.

```
fn2 = mlrun.code_to_function("serving_example_flow",
                             kind="serving",
```

(continues on next page)

(continued from previous page)

```
                                image="mlrun/mlrun")

graph2 = fn2.set_topology("flow")

graph2_enrich = graph2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})
↪')

# add an Ensemble router with two child models (routes)
router = graph2.add_step(mlrun.serving.ModelRouter(), name="router", after="enrich")
router.add_route("m1", class_name="ClassifierModel", model_path='https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl')
router.respond()

# Add additional models
#router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# plot the graph (using Graphviz)
graph2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fd46e4dda50>
```

```
fn2_server = fn2.to_mock_server()

result = fn2_server.test("/v2/models/m1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:42,142 [info] model m1 was loaded
> 2021-11-02 04:18:42,142 [info] Initializing endpoint records
> 2021-11-02 04:18:42,183 [info] Loaded ['m1']
{'id': 'f713fd7eedeb431eba101b13c53a15b5'}
```

Remote

You can chain functions together with remote execution. This allows you to:

- Call existing functions from the graph and reuse them from other graphs.
- Scale up and down different components individually.

Calling a remote function can either use HTTP or via a queue (streaming).

HTTP

Calling a function using http uses the special `$remote` class. First deploy the remote function:

```
remote_func_name = "serving-example-flow"
project_name = "graph-basic-concepts"
fn_remote = mlrun.code_to_function(remote_func_name,
                                   project=project_name,
                                   kind="serving",
                                   image="mlrun/mlrun")
```

(continues on next page)

(continued from previous page)

```
fn_remote.add_model("model1", class_name="ClassifierModel", model_path="https://s3.
↪wasabisys.com/iguazio/models/iris/model.pkl")

remote_addr = fn_remote.deploy()
```

```
> 2022-03-17 08:20:40,674 [info] Starting remote function deploy
2022-03-17 08:20:40  (info) Deploying function
2022-03-17 08:20:40  (info) Building
2022-03-17 08:20:40  (info) Staging files and preparing base images
2022-03-17 08:20:40  (info) Building processor image
2022-03-17 08:20:42  (info) Build complete
2022-03-17 08:20:47  (info) Function deploy complete
> 2022-03-17 08:20:48,289 [info] successfully deployed function: {'internal_
↪invocation_urls': ['nuclio-graph-basic-concepts-serving-example-flow.default-tenant.
↪svc.cluster.local:8080'], 'external_invocation_urls': ['graph-basic-concepts-
↪serving-example-flow-graph-basic-concepts.default-tenant.app.maor-gcp2.iguazio-cd0.
↪com/']}
```

Create a new function with a graph and call the remote function above:

```
fn_preprocess = mlrun.new_function("preprocess", kind="serving")
graph_preprocessing = fn_preprocess.set_topology("flow")

graph_preprocessing.to("storey.Extend", name="enrich", _fn='({"tag": "something"})').
↪to(
                       "$remote", "remote_func", url=f'{remote_addr}v2/models/model1/
↪infer', method='put').respond()

graph_preprocessing.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f57dc96a0d0>
```

```
fn3_server = fn_preprocess.to_mock_server()
my_data = '''{"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}'''
result = fn3_server.test("/v2/models/my_model/infer", body=my_data)
print(result)
```

```
> 2022-03-17 08:20:48,374 [warning] run command, file or code were not specified
{'id': '3a1dd36c-e7de-45af-a0c4-72e3163ba92a', 'model_name': 'model1', 'outputs': [0,␣
↪2]}
```

Queue (streaming)

You can use queues to send events from one part of the graph to another and to decouple the processing of those parts. Queues are better suited to deal with bursts of events, since all the events are stored in the queue until they are processed. The example below uses a V3IO stream, which is a fast real-time implementation of a stream that allows processing of events at very low latency.

```
%%writefile echo.py
def echo_handler(x):
    print(x)
    return x
```

```
Overwriting echo.py
```

Configure the streams

```
import os
streams_prefix = f"v3io:///users/{os.getenv('V3IO_USERNAME')}/examples/graph-basic-
↪concepts"

input_stream = streams_prefix + "/in-stream"
out_stream = streams_prefix + "/out-stream"
err_stream = streams_prefix + "/err-stream"
```

Create the graph. Note that in the `to` method the class name is specified to be >> or `$queue` to specify that this is a queue.

```
fn_preprocess2 = mlrun.new_function("preprocess", kind="serving")
fn_preprocess2.add_child_function('echo_func', './echo.py', 'mlrun/mlrun')

graph_preprocess2 = fn_preprocess2.set_topology("flow")

graph_preprocess2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})')\
                 .to(">>", "input_stream", path=input_stream)\
                 .to(name="echo", handler="echo_handler", function="echo_func")\
                 .to(">>", "output_stream", path=out_stream)

graph_preprocess2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f57c7907990>
```

```
from echo import *

fn4_server = fn_preprocess2.to_mock_server(current_function="*")

result = fn4_server.test("/v2/models/my_model/infer", body=my_data)

print(result)
```

```
> 2022-03-17 08:20:55,182 [warning] run command, file or code were not specified
{'id': 'a6efe8217b024ec7a7e02cf0b7850b91'}
{'inputs': [[5.1, 3.5, 1.4, 0.2], [7.7, 3.8, 6.7, 2.2]], 'tag': 'something'}
```

Examples

Simple model serving router

See the Simple model serving router.

NLP processing pipeline with real-time streaming

In some cases it's useful to split your processing to multiple functions and use streaming protocols to connect those functions.

See the example, and also the *full notebook example*, where the data processing is in the first function/container and the NLP processing is in the second function. And the second function contains the GPU.

Currently queues only support iguazio v3io stream. Kafka support will be added in an upcoming release.

Creating a custom model serving class

Model serving classes implement the full model serving functionality which include loading models, pre- and post-processing, prediction, explainability, and model monitoring.

Model serving classes must inherit from `mlrun.serving.V2ModelServer`, and at the minimum implement the `load()` (download the model file(s) and load the model into memory) and `predict()` (accept request payload and return prediction/inference results) methods.

The class is initialized automatically by the model server and can run locally as part of a nuclio serverless function, or as part of a real-time pipeline

You need to implement two mandatory methods:

- **load()** - download the model file(s) and load the model into memory, note this can be done synchronously or asynchronously
- **predict()** - accept request payload and return prediction/inference results

You can override additional methods : `preprocess`, `validate`, `postprocess`, `explain`. You can add a custom api endpoint by adding method `op_xx(event)`. Invoke it by calling the /xx (operation = xx).

In this section

Minimal sklearn serving function example

```
from cloudpickle import load
import numpy as np
import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

Test the function locally using the mock server:

```
import mlrun
from sklearn.datasets import load_iris

fn = mlrun.new_function('my_server', kind='serving')

# set the topology/router and add models
graph = fn.set_topology("router")
fn.add_model("model1", class_name="ClassifierModel", model_path="<path1>")
fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})
```

load() method

In the load method, download the model from external store, run the algorithm/framework `load()` call, and do any other initialization logic.

The load runs synchronously (the deploy is stalled until load completes). This can be an issue for large models and cause a readiness timeout. You can increase the function `spec.readiness_timeout`, or alternatively choose async loading (load () runs in the background) by setting the function `spec.load_mode = "async"`.

The function `self.get_model()` downloads the model metadata object and main file (into `model_file` path). Additional files can be accessed using the returned `extra_data` (dict of dataitem objects).

The model metadata object is stored in `self.model_spec` and provides model parameters, metrics, schema, etc. Parameters can be accessed using `self.get_param(key)`. The parameters can be specified in the model or during the function/model deployment.

predict() method

The predict method is called when you access the `/infer` or `/predict` url suffix (operation). The method accepts the request object (as dict), see *Model server API*. And it should return the specified response object.

explain() method

The explain method provides a hook for model explainability, and is accessed using the `/explain` operation.

pre/post and validate hooks

You can overwrite the `preprocess`, `validate`, and `postprocess` methods for additional control The call flow is:

```
pre-process -> validate -> predict/explain -> post-process
```

Models, routers and graphs

Every serving function can host multiple models and logical steps. Multiple functions can connect in a graph to form complex real-time pipelines.

The basic serving function has a logical `router` with routes to multiple child `models`. The url or the message determines which model is selected, e.g. using the url schema:

```
/v2/models/<model>[/versions/<ver>]/operation
```

> Note: the `model`, `version` and `operation` can also be specified in the message body to support streaming protocols (e.g. Kafka).

More complex routers can be used to support ensembles (send the request to all child models and aggregate the result), multi-armed-bandit, etc.

You can use a pre-defined Router class, or write your own custom router. Routera can route to models on the same function or access models on a separate function.

To specify the topology, router class and class args use `.set_topology()` with your function.

Creating a model serving function (service)

To provision a serving function, you need to create an MLRun function of type `serving`. This can be done by using the `code_to_function()` call from a notebook. You can also import an existing serving function/template from the marketplace.

Example (run inside a notebook): this code converts a notebook to a serving function and adding a model to it:

```
from mlrun import code_to_function
fn = code_to_function('my-function', kind='serving')
fn.add_model('m1', model_path=<model-artifact/dir>, class_name='MyClass', x=100)
```

See `.add_model()` docstring for help and parameters.

See the full Model Server example.

If you want to use multiple versions for the same model, use `:` to separate the name from the version. For example, if the name is `mymodel:v2` it means model name `mymodel` version `v2`.

You should specify the `model_path` (url of the model artifact/dir) and the `class_name` name (or class `module.submodule.class`). Alternatively you can set the `model_url` for calling a model that is served by another function (can be used for ensembles).

The function object(fn) accepts many options. You can specify replicas range (auto-scaling), cpu/gpu/mem resources, add shared volume mounts, secrets, and any other Kubernetes resource through the `fn.spec` object or fn methods.

For example, `fn.gpu(1)` means each replica uses one GPU.

To deploy a model, simply call:

```
fn.deploy()
```

You can also deploy a model from within an ML pipeline (check the various demos for details).

Model monitoring

Model activities can be tracked into a real-time stream and time-series DB. The monitoring data is used to create real-time dashboards and track model accuracy and drift. To set the tracking stream options, specify the following function spec attributes:

```
    fn.set_tracking(stream_path, batch, sample)
```

- **stream_path** - the v3io stream path (e.g. `v3io:///users/..`)
- **sample** - optional, sample every N requests
- **batch** - optional, send micro-batches every N requests

Model Server API

MLRun Serving follows the same REST API defined by Triton and KFServing v2.

Nuclio also supports streaming protocols (Kafka, kinesis, MQTT, etc.). When streaming, the `model` name and `operation` can be encoded inside the message body.

The APIs are:

- *explain*
- *get model health / readiness*
- *get model metadata*
- *get server info*
- *infer / predict*
- *list models*

explain

POST /v2/models/[/versions/{VERSION}]/explain

Request body:

```
{
  "id" : $string #optional,
  "model" : $string #optional
  "parameters" : $parameters #optional,
  "inputs" : [ $request_input, ... ],
  "outputs" : [ $request_output, ... ] #optional
}
```

Response structure:

```
{
  "model_name" : $string,
  "model_version" : $string #optional,
  "id" : $string,
  "outputs" : [ $response_output, ... ]
}
```

get model health / readiness

```
GET v2/models/${MODEL_NAME}[/versions/${VERSION}]/ready
```

Returns 200 for Ok, 40X for not ready.

get model metadata

```
GET v2/models/${MODEL_NAME}[/versions/${VERSION}]
```

Response example: `{"name": "m3", "version": "v2", "inputs": [..], "outputs": [..]}`

get server info

```
GET /
GET /v2/health
```

Response example: `{'name': 'my-server', 'version': 'v2', 'extensions': []}`

infer / predict

```
POST /v2/models/<model>[/versions/{VERSION}]/infer
```

Request body:

```
{
  "id" : $string #optional,
  "model" : $string #optional
  "data_url" : $string #optional
  "parameters" : $parameters #optional,
  "inputs" : [ $request_input, ... ],
  "outputs" : [ $request_output, ... ] #optional
}
```

- **id**: Unique Id of the request, if not provided a random value is provided.
- **model**: Model to select (for streaming protocols without URLs).
- **data_url**: Option to load the `inputs` from an external file/s3/v3io/.. object.
- **parameters**: Optional request parameters.
- **inputs**: List of input elements (numeric values, arrays, or dicts).
- **outputs:** Optional, requested output values.

Note: You can also send binary data to the function, for example, a JPEG image. The serving engine pre-processor detects it based on the HTTP content-type and converts it to the above request structure, placing the image bytes array in the `inputs` field.

Response structure:

```
{
  "model_name" : $string,
  "model_version" : $string #optional,
  "id" : $string,
  "outputs" : [ $response_output, ... ]
}
```

list models

```
GET /v2/models/
```

Response example: `{"models": ["m1", "m2", "m3:v1", "m3:v2"]}`

2.1.19 Model monitoring overview (beta)

Note: Model monitoring is based on Iguazio's streaming technology. Contact Iguazio to enable this feature.

MLRun's model monitoring service tracks the performance of models in production to help identify potential issues with concept drift and prediction accuracy before they impact business goals. Typically, model monitoring is used by devops for tracking model performance, and by data scientists to track model drift. Two monitoring types are supported:

- Model operational performance (latency, requests per second, etc.).
- Drift detection — identifies potential issues with the model. See *Drift Analysis* for more details.

Model monitoring provides warning alerts that can be sent to stakeholders for processing.

The model monitoring data can be viewed using Iguazio's user interface or through Grafana dashboards. Grafana is an interactive web application visualization tool that can be added as a service in the Iguazio platform. See *Model Monitoring Using Grafana Dashboards* for more details.

In this section

Model monitoring overview (beta)

In this section

Architecture

The model monitoring process flow starts with collecting operational data. The operational data are converted to vectors, which are posted to the Model Server. The model server is then wrapped around a machine learning model that uses a function to calculate predictions based on the available vectors. Next, the model server creates a log for the input and output of the vectors, and the entries are written to the production data stream (a *v3io stream*). While the model server is processing the vectors, a Nuclio operation monitors the log of the data stream and is triggered when a new log entry is detected. The Nuclio function examines the log entry, processes it into statistics which are then written to the statistics databases (parquet file, time series database and key value database). The parquet files are written as a feature set under the model monitoring project. The parquet files can be read either using `pandas.read_parquet` or `feature_set.get_offline_features`, like any other feature set. In parallel, a scheduled MLRun job runs reading the parquet files, performing drift analysis. The drift analysis data is stored so that the user can retrieve it in the Iguazio UI or in a Grafana dashboard.

Drift analysis

The model monitoring feature provides drift analysis monitoring. Model drift in machine learning is a situation where the statistical properties of the target variable (what the model is trying to predict) change over time. In other words, the production data has changed significantly over the course of time and no longer matches the input data used to train the model. So, for this new data, accuracy of the model predictions is low. Drift analysis statistics are computed once an hour. For more information see Concept Drift.

Common terminology

The following terms are used in all the model monitoring pages:

- **Total Variation Distance** (TVD) — The statistical difference between the actual predictions and the model's trained predictions.
- **Hellinger Distance** — A type of f-divergence that quantifies the similarity between the actual predictions, and the model's trained predictions.
- **Kullback–Leibler Divergence** (KLD) — The measure of how the probability distribution of actual predictions is different from the second model's trained reference probability distribution.
- **Model Endpoint** — A combination of a deployed Nuclio function and the models themselves. One function can run multiple endpoints; however, statistics are saved per endpoint.

Model monitoring using the Iguazio platform interface

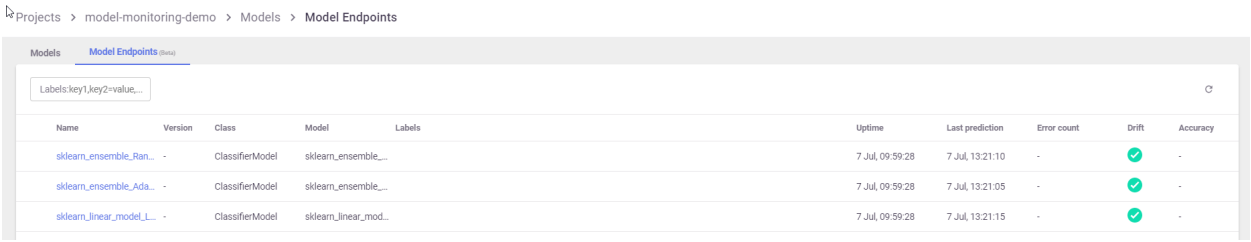
Iguazio's model monitoring data is available for viewing through the regular platform interface. The platform provides four information pages with model monitoring data.

- *Model endpoint summary list*
- *Model endpoint overview*
- *Model drift analysis*
- *Model features analysis*

1. Select a project from the project tiles screen.
2. From the project dashboard, press the **Models** tile to view the models currently deployed .
3. Click **Model Endpoints** from the menu to display a list of monitored endpoints. If the Model Monitoring feature is not enabled, the endpoints list is empty.

Model endpoint summary list

The Model Endpoints summary list provides a quick view of the model monitoring data.



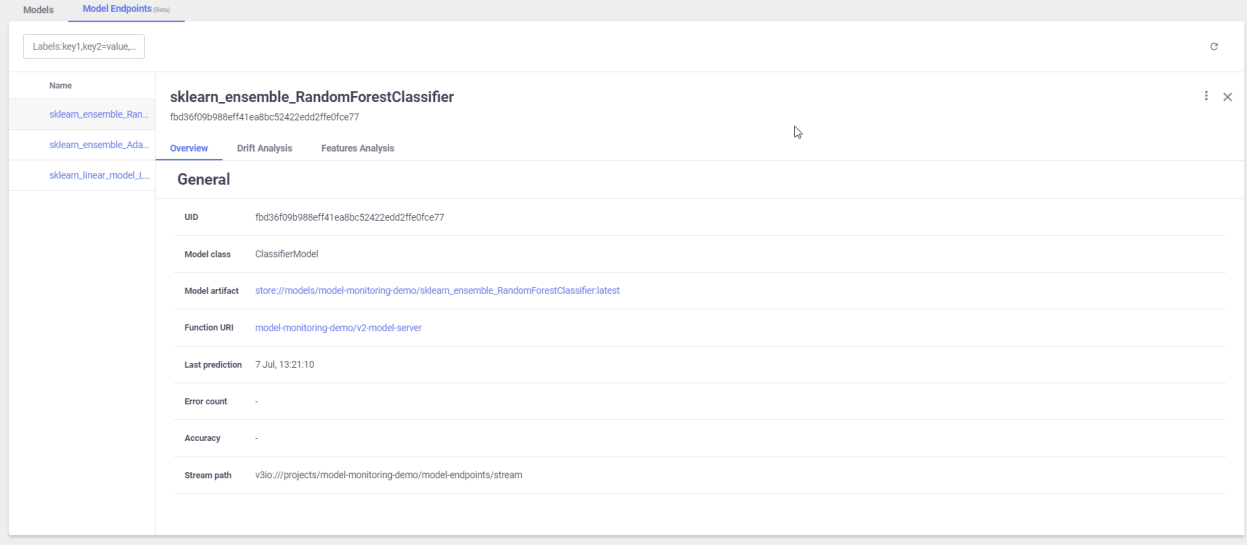
The summary page contains the following fields:

- **Name**—the name of the model endpoint
- **Version**—user configured version taken from model deployment
- **Class**—the implementation class that is used by the endpoint
- **Model**—user defined name for the model
- **Labels**—user configurable tags that are searchable
- **Uptime**—first request for production data
- **Last Prediction**—most recent request for production data
- **Error Count**—includes prediction process errors such as operational issues (For example, a function in a failed state), as well as data processing errors (For example, invalid timestamps, request ids, type mismatches etc.)
- **Drift**—indication of drift status (no drift (green), possible drift (yellow), drift detected (red))
- **Accuracy**—a numeric value representing the accuracy of model predictions (N/A)

Note: Model Accuracy is currently under development.

Model endpoint overview

The Model Endpoints overview pane displays general information about the selected model.



The Overview page contains the following fields:

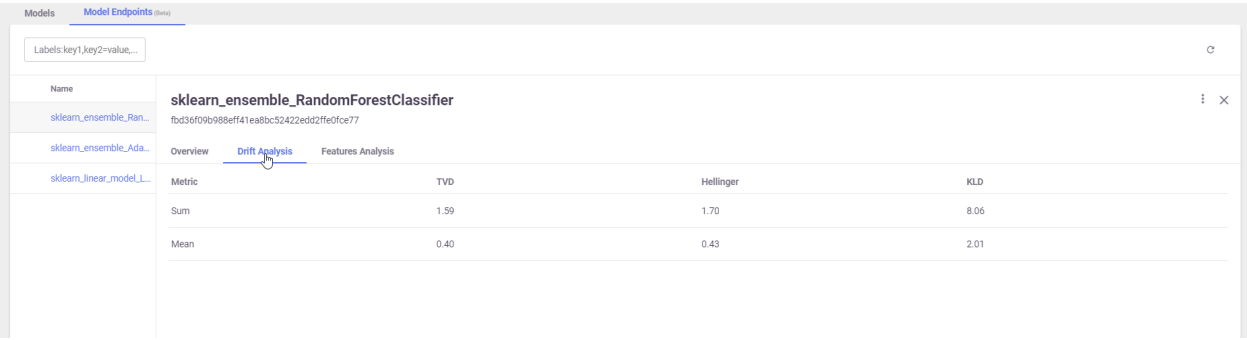
- **UUID** — the ID of the deployed model
- **Model Class** — the implementation class that is used by the endpoint
- **Model Artifact** — reference to the model's file location
- **Function URI** — the MLRun function to access the model
- **Last Prediction** — most recent request for production data

- **Error Count** — includes prediction process errors such as operational issues (For example, a function in a failed state), as well as data processing errors (For example, invalid timestamps, request ids, type mismatches etc.)
- **Accuracy** — a numeric value representing the accuracy of model predictions (N/A)
- **Stream path** — the input and output stream of the selected model

Use the ellipsis to view the YAML resource file for details about the monitored resource.

Model drift analysis

The Drift Analysis pane provides performance statistics for the currently selected model.



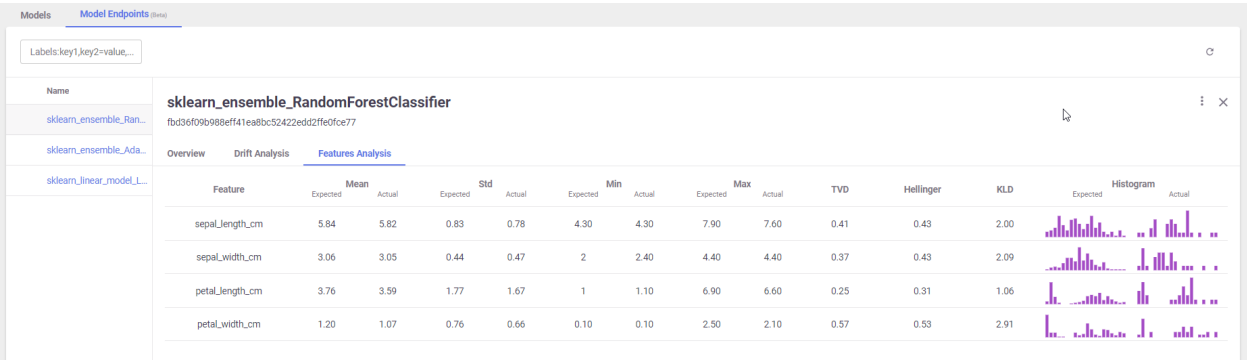
Each of the following fields has both sum and mean numbers displayed. For definitions of the terms see *Common Terminology*.

- **TVD**
- **Hellinger**
- **KLD**

Use the ellipsis to view the YAML resource file for details about the monitored resource.

Model features analysis

The Features Analysis pane provides details of the drift analysis in a table format with each feature in the selected model on its own line.



Each field has a pair of columns. The **Expected** column displays the results from the model training phase, and the **Actual** column displays the results from the live production data. The following fields are available:

- **Mean**

- **STD** (Standard deviation)
- **Min**
- **Max**
- **TVD**
- **Hellinger**
- **KLD**
- **Histograms**—the approximate representation of the distribution of the data. Hover over the bars in the graph for the details.

Use the ellipsis to view the YAML resource file for details about the monitored resource.

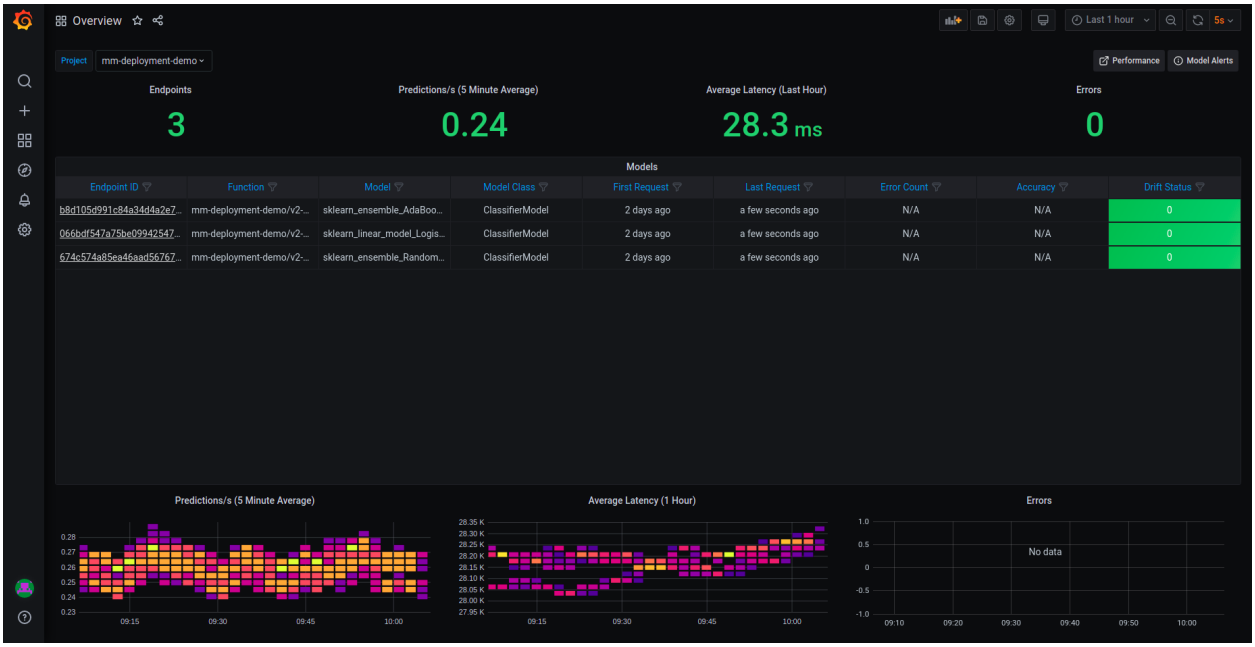
Model monitoring using Grafana dashboards

You can deploy a Grafana service in your Iguazio instance and use Grafana Dashboards to view model monitoring details. There are three dashboards available:

- *Overview Dashboard*
- *Details Dashboard*
- *Performance Dashboard*

Model endpoints overview dashboard

The Overview dashboard displays the model endpoint IDs of a specific project. Only deployed models with Model Monitoring enabled are displayed. Endpoint IDs are URIs used to provide access to performance data and drift detection statistics of a deployed model.



The Overview pane provides details about the performance of all the deployed and monitored models within a project. You can change projects by choosing a new project from the **Project** dropdown. The Overview dashboard displays the

number of endpoints in the project, the average predictions per second (using a 5-minute rolling average), the average latency (using a 1-hour rolling average), and the total error count in the project.

Additional details include:

- **Endpoint ID** — the ID of the deployed model. Use this link to drill down to the model performance and details panes.
- **Function** — the MLRun function to access the model
- **Model** — user defined name for the model
- **Model Class** — the implementation class that is used by the endpoint
- **First Request** — first request for production data
- **Last Request** — most recent request for production data
- **Error Count** — includes prediction process errors such as operational issues (for example, a function in a failed state), as well as data processing errors (for example, invalid timestamps, request ids, type mismatches etc.)
- **Accuracy** — a numeric value representing the accuracy of model predictions (N/A)
- **Drift Status** — no drift (green), possible drift (yellow), drift detected (red)

At the bottom of the dashboard are heat maps for the Predictions per second, Average Latency and Errors. The heat maps display data based on 15 minute intervals. See *How to Read a Heat Map* for more details.

Click an endpoint ID to drill down the performance details of that model.

How to read a heat map

Heat maps are used to analyze trends and to instantly transform and enhance data through visualizations. This helps to quickly identify areas of interest, and empower users to explore the data in order to pinpoint where there may be potential issues. A heat map uses a matrix layout with colour and shading to show the relationship between two categories of values (x and y axes), so the darker the cell, the higher the value. The values presented along each axis correspond to a cell which is color-coded to represent the relationship between the two categories. The Predictions per second heatmap shows the relationship between time, and the predictions per second, and the Average Latency per hour shows the relationship between time and the latency.

To properly read the heap maps, follow the hierarchy of shades from the darkest (the highest values) to the lightest shades (the lowest values).

Note: The exact quantitative values represented by the colors may be difficult to determine. Use the *Performance Dashboard* to see detailed results.

Model endpoint details dashboard

The model endpoint details dashboard displays the real time performance data of the selected model in detail. Model performance data provided is rich and is used to fine tune or diagnose potential performance issues that may affect business goals. The data in this dashboard changes based on the selection of the project and model.

This dashboard has three panes:

1. *Project and model summary*
2. *Analysis panes*
    1. Overall drift analysis

2. Features analysis

3. *Incoming features graph*

Project and model summary

Use the dropdown to change the project and model. The dashboard presents the following information about the project:

- **Endpoint ID** — the ID of the deployed model
- **Model** — user defined name for the model
- **Function URI** — the MLRun function to access the model
- **Model Class** — the implementation class that is used by the endpoint
- **Prediction/s** — the average number of predictions per second over a rolling 5-minute period
- **Average Latency** — the average latency over a rolling 1-hour period
- **First Request** — first request for production data
- **Last Request** — most recent request for production data

Use the *Performance* and *Overview* buttons view those dashboards.

Analysis panes

This pane has two sections: Overall Drift Analysis and Features Analysis. The Overall Drift Analysis pane provides performance statistics for the currently selected model.

- **TVD** (sum and mean)
- **Hellinger** (sum and mean)
- **KLD** (sum and mean)

The Features Analysis pane provides details of the drift analysis for each feature in the selected model. This pane includes five types of statistics:

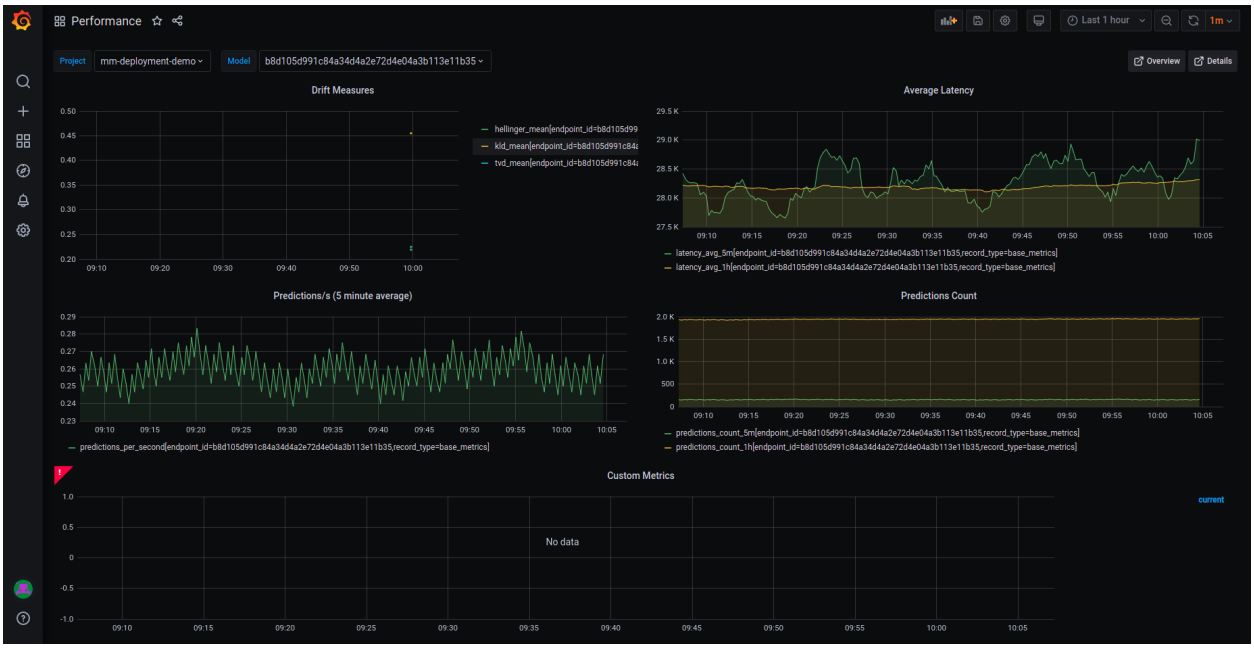
- **Actual** (min, mean and max) — results based on actual live data stream
- **Expected** (min, mean and max) — results based on training data
- **TVD**
- **Hellinger**
- **KLD**

Incoming features graph

This graph displays the performance of the features that are in the selected model based on sampled data points from actual feature production data. The graph displays the values of the features in the model over time.

Model endpoint performance dashboard

Model endpoint performance displays performance details in graphical format.



This dashboard has five graphs:

- **Drift Measures** — the overall drift over time for each of the endpoints in the selected model

- **Average Latency** — the average latency of the model in 5 minute intervals, for 5 minutes and 1 hour rolling windows
- **Predictions/s** — the model predictions per second displayed in 5 second intervals for 5 minutes (rolling)
- **Predictions Count** — the number of predictions the model makes for 5 minutes and 1 hour rolling windows

Configuring Grafana dashboards

Verify that you have a Grafana service running in your Iguazio MLOps Platform. If you do not have a Grafana service running, see Adding Grafana Dashboards to create and configure it. When you create the service: In the **Custom Parameters** tab, **Platform data-access user** parameter, select a user with access to the `/user/pipelines` directory.

For working with Iguazio 3.0.x:

1. Make sure you have the `model-monitoring` as a Grafana data source configured in your Grafana service. If not, add it by:
    1. Open your grafana service.
    2. Navigate to **Configuration | Data Sources**.
    3. Press **Add data source**.
    4. Select the **SimpleJson** datasource and configure the following parameters.

```
URL: http://mlrun-api:8080/api/grafana-proxy/model-endpoints
Access: Server (default)

## Add a custom header of:
    X-V3io-Session-Key: <YOUR ACCESS KEY>
```

    5. Press **Save & Test** for verification. You'll receive a confirmation with either a success or a failure message.
2. Download the following monitoring dashboards:
    - `Model Monitoring - Overview`
    - `Model Monitoring - Details`
    - `Model Monitoring - Performance`
3. Import the downloaded dashboards to your Grafana service. To import that dashboards into your Grafana service:
    1. Navigate to your Grafana service in the Services list and press it.
    2. Press the dashboards icon in left menu.
    3. In the dashboard management screen. press **IMPORT**, and select one file to import. Repeat this step for each dashboard.

For working with Iguazio 3.2.x and later: Add access keys to your model-monitoring data source:

1. Open your Grafana service.
2. Navigate to **Configuration | Data Sources**.
3. Press **mlrun-model-monitoring**.
4. In Custom HTTP Headers, configure the cookie parameter. Set the value to `cookie: session=j:{"sid": "<YOUR ACCESS KEY>"}`

The overview, details, and performance dashboards are in **Dashboards | Manage | private**

Note: You need to train and deploy a model to see results in the dashboards. The dashboards immediately display data if you already have a model trained and running with production data.

Enable model monitoring (beta)

To see tracking results, model monitoring needs to be enabled in each model.

To enable model monitoring, include `serving_fn.set_tracking()` in the model server.

To utilize drift measurement, supply the train set in the training step.

In this section

Model monitoring demo

Use the following code blocks to test and explore model monitoring.

```
# Set project name
project_name = "demo-project"
```

Deploy model servers

Use the following code to deploy a model server in the Iguazio instance.

```
import os
import pandas as pd
from sklearn.datasets import load_iris

from mlrun import import_function, get_dataitem, get_or_create_project
from mlrun.platforms import auto_mount

project = get_or_create_project(project_name, context="./")
project.set_model_monitoring_credentials(os.environ.get("V3IO_ACCESS_KEY"))

# Download the pre-trained Iris model
get_dataitem("https://s3.wasabisys.com/iguazio/models/iris/model.pkl").download(
↪"model.pkl")

iris = load_iris()
train_set = pd.DataFrame(iris['data'],
                         columns=['sepal_length_cm', 'sepal_width_cm',
                                  'petal_length_cm', 'petal_width_cm'])

# Import the serving function from the function hub
serving_fn = import_function('hub://v2_model_server', project=project_name).
↪apply(auto_mount())
```

(continues on next page)

(continued from previous page)

```
model_name = "RandomForestClassifier"

# Log the model through the projects API so that it is available through the feature␣
↪store API
project.log_model(model_name, model_file="model.pkl", training_set=train_set)

# Add the model to the serving function's routing spec
serving_fn.add_model(model_name, model_path=f"store://models/{project_name}/{model_
↪name}:latest")

# Enable model monitoring
serving_fn.set_tracking()

# Deploy the function
serving_fn.deploy()
```

Simulating requests

Use the following code to simulate production data.

```
import json
from time import sleep
from random import choice, uniform

iris_data = iris['data'].tolist()

while True:
    data_point = choice(iris_data)
    serving_fn.invoke(f'v2/models/{model_name}/infer', json.dumps({'inputs': [data_
↪point]}))
    sleep(uniform(0.2, 1.7))
```

2.1.20 CI/CD, rolling upgrades, git

In this section

Github/Gitlab and CI/CD integration

MLRun workflows can run inside the CI system. The most common method is to use the CLI command `mlrun project` to load the project and run a workflow as part of a code update (e.g. pull request, etc.). The pipeline tasks are executed on the Kubernetes cluster, which is orchestrated by MLRun.

When MLRun is executed inside a GitHub Action or GitLab CI/CD pipeline it detects the environment attributes automatically (e.g. repo, commit id, etc.). In addition, a few environment variables and credentials must be set:

- **MLRUN_DBPATH** — url of the MLRun cluster.
- **V3IO_USERNAME** — username in the remote Iguazio cluster.
- **V3IO_ACCESS_KEY** — access key to the remote Iguazio cluster.
- **GIT_TOKEN** or **GITHUB_TOKEN** — Github/Gitlab API token (set automatically in Github Actions).
- **SLACK_WEBHOOK** — optional. Slack API key when using slack notifications.

When the workflow runs inside the Git CI system it reports the pipeline progress and results back into the Git tracking system, similar to:

Contents

Using GitHub Actions

When running using GitHub Actions you need to set the credentials/secrets and add a script under the `.github/workflows/` directory, which is executed when the code is commited/pushed.

Example script that is invoked when you add the comment "/run" to your pull request:

```
name: mlrun-project-workflow
on: [issue_comment]

jobs:
  submit-project:
    if: github.event.issue.pull_request != null && startsWith(github.event.comment.
↪body, '/run')
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.7
      uses: actions/setup-python@v1
      with:
        python-version: '3.7'
        architecture: 'x64'

    - name: Install mlrun
      run: python -m pip install pip install mlrun
    - name: Submit project
      run: python -m mlrun project ./ -w -r main ${CMD:5}
      env:
        V3IO_USERNAME: ${{ secrets.V3IO_USERNAME }}
        V3IO_API: ${{ secrets.V3IO_API }}
        V3IO_ACCESS_KEY: ${{ secrets.V3IO_ACCESS_KEY }}
        MLRUN_DBPATH: ${{ secrets.MLRUN_DBPATH }}
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK: ${{ secrets.SLACK_WEBHOOK }}
        CMD: ${{ github.event.comment.body}}
```

See the full example in **https://github.com/mlrun/project-demo**

Using GitLab CI/CD

When running using GitLab CI/CD you need to set the credentials/secrets and update the script `.gitlab-ci.yml` directory, which is executed when code is commited/pushed.

Example script that is invoked when you create a pull request (merge requests):

```
image: mlrun/mlrun

run:
  script:
    - python -m mlrun project ./ -w -r ci
  only:
    - merge_requests
```

See the full example in **https://gitlab.com/yhaviv/test2**

2.1.21 API Index

2.1.22 API By Module

MLRun is organized into the following modules. The most common functions are exposed in the *mlrun* module, which is the recommended starting point.

mlrun.frameworks

MLRun is providing a quick and easy integration into your code with mlrun.frameworks: a collection of sub-modules for the most commonly used machine and deep learning frameworks, providing features such as automatic logging, model management, and distributed training.

mlrun.frameworks.auto_mlrun

class mlrun.frameworks.auto_mlrun.auto_mlrun.**AutoMLRun**
Bases: object

A library of automatic functions for managing models using MLRun's frameworks package.

static apply_mlrun(*model: Optional[ModelType] = None*, *model_name: Optional[str] = None*, *tag: str = ''*, *model_path: Optional[str] = None*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *context: Optional[*mlrun.execution.MLClientCtx*] = None*, *framework: Optional[str] = None*, *auto_log: bool = True*, ***kwargs*) → mlrun.frameworks._common.model_handler.ModelHandler

Use MLRun's 'apply_mlrun' of the detected given model's framework to wrap the framework relevant methods and gain the framework's features in MLRun. A ModelHandler initialized with the model will be returned.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. If not given will have a default name according to the framework.

- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module␣
↪import func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – A MLRun context.
- **auto_log** – Whether to enable auto-logging capabilities of MLRun or not. Auto logging will add default artifacts and metrics besides the one you can pass here.
- **framework** – The model's framework. If None, AutoMLRun will try to figure out the framework. From the provided model or model path. Defaulted to None.
- **kwargs** – Additional parameters for the specific framework's 'apply_mlrun' function like metrics, callbacks and more (read the docs of the required framework to know more).

Returns The framework's model handler initialized with the given model.

static load_model(*model_path: str, model_name: Optional[str] = None, context: Optional[mlrun.execution.MLClientCtx] = None, modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None, custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None, custom_objects_directory: Optional[str] = None, framework: Optional[str] = None, **kwargs*) → mlrun.frameworks._common.model_handler.ModelHandler

Load a model using MLRun's ModelHandler. The loaded model can be accessed from the model handler returned via model_handler.model. If the model is a store object uri (it is logged in MLRun) then the framework will be read automatically, otherwise (for local path and urls) it must be given. The other parameters will be automatically read in case its a logged model in MLRun.

Parameters

- **model_path** – A store object path of a logged model object in MLRun.
- **model_name** – The model name to use for storing the model artifact. If not given will have a default name according to the framework.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module␣
↪import func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – A MLRun context.

- **framework** – The model's framework. It must be provided for local paths or urls. If None, AutoMLRun will assume the model path is of a store uri model artifact and try to get the framework from it. Defaulted to None.
- **kwargs** – Additional parameters for the specific framework's ModelHandler class.

Returns The model inside a MLRun model handler.

Raises **MLRunInvalidArgumentError** – In case the framework is incorrect or missing.

mlrun.frameworks.auto_mlrun.auto_mlrun.**framework_to_apply_mlrun**(*framework: str*) → Callable[[...], mlrun.frameworks._common.model_hand

Get the 'apply_mlrun' shortcut function of the given framework's name.

Parameters **framework** – The framework's name.

Returns The framework's 'apply_mlrun' shortcut function.

Raises **MLRunInvalidArgumentError** – If the given framework is not supported by AutoMLRun or if it does not have an 'apply_mlrun' yet.

mlrun.frameworks.auto_mlrun.auto_mlrun.**framework_to_model_handler**(*framework: str*) → Type[mlrun.frameworks._common.m

Get the ModelHandler class of the given framework's name.

Parameters **framework** – The framework's name.

Returns The framework's ModelHandler class.

Raises **MLRunInvalidArgumentError** – If the given framework is not supported by AutoMLRun.

mlrun.frameworks.auto_mlrun.auto_mlrun.**get_framework_by_class_name**(*model: ModelType*) → str

Get the framework name of the given model by its class name.

Parameters **model** – The model to get its framework.

Returns The model's framework.

Raises **MLRunInvalidArgumentError** – If the given model's class name is not supported by AutoMLRun or not recognized.

mlrun.frameworks.auto_mlrun.auto_mlrun.**get_framework_by_instance**(*model: ModelType*) → str

Get the framework name of the given model by its instance.

Parameters **model** – The model to get his framework.

Returns The model's framework.

Raises **MLRunInvalidArgumentError** – If the given model type is not supported by AutoMLRun or not recognized.

mlrun.frameworks.tf_keras

mlrun.frameworks.pytorch

mlrun.frameworks.sklearn

mlrun.frameworks.xgboost

mlrun.frameworks.lgbm

mlrun

mlrun.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *code_output: str = ''*, *embed_code: bool = True*, *description: str = ''*, *requirements: Optional[Union[str, List[str]]] = None*, *categories: Optional[List[str]] = None*, *labels: Optional[Dict[str, str]] = None*, *with_doc: bool = True*, *ignored_tags=None*) → Union[mlrun.runtimes.mpijob.v1alpha1.MpiRuntimeV1Alpha1, mlrun.runtimes.mpijob.v1.MpiRuntimeV1, mlrun.runtimes.function.RemoteRuntime, mlrun.runtimes.serving.ServingRuntime, mlrun.runtimes.daskjob.DaskCluster, mlrun.runtimes.kubejob.KubejobRuntime, mlrun.runtimes.local.LocalRuntime, mlrun.runtimes.sparkjob.spark2job.Spark2Runtime, mlrun.runtimes.sparkjob.spark3job.Spark3Runtime, mlrun.runtimes.remotesparkjob.RemoteSparkRuntime]

Convenience function to insert code and configure an mlrun runtime.

Easiest way to construct a runtime type object. Provides the most often used configuration options for all runtimes as parameters.

Instantiated runtimes are considered 'functions' in mlrun, but they are anything from nuclio functions to generic kubernetes pods to spark jobs. Functions are meant to be focused, and as such limited in scope and size. Typically a function can be expressed in a single python module with added support from custom docker images and commands for the environment. The returned runtime object can be further configured if more customization is required.

One of the most important parameters is 'kind'. This is what is used to specify the chosen runtimes. The options are:

- local: execute a local python or shell script
- job: insert the code into a Kubernetes pod and execute it
- nuclio: insert the code into a real-time serverless nuclio function
- serving: insert code into orchestrated nuclio function(s) forming a DAG
- dask: run the specified python code / script as Dask Distributed job
- mpijob: run distributed Horovod jobs over the MPI job operator
- spark: run distributed Spark job using Spark Kubernetes Operator
- remote-spark: run distributed Spark job on remote Spark service

Learn more about function runtimes here: https://docs.mlrun.org/en/latest/runtimes/functions.html#function-runtimes

Parameters

- **name** – function name, typically best to use hyphen-case
- **project** – project used to namespace the function, defaults to 'default'
- **tag** – function tag to track multiple versions of the same function, defaults to 'latest'
- **filename** – path to .py/.ipynb file, defaults to current jupyter notebook
- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to 'main:handler'
- **kind** – function runtime type string - nuclio, job, etc. (see docstring for all options)
- **image** – base docker image to use for building the function container, defaults to None
- **code_output** – specify '.' to generate python module from the current jupyter notebook
- **embed_code** – indicates whether or not to inject the code directly into the function runtime spec, defaults to True
- **description** – short function description, defaults to ''
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **categories** – list of categories for mlrun function marketplace, defaults to None
- **labels** – immutable name/value pairs to tag the function with useful metadata, defaults to None
- **with_doc** – indicates whether to document the function parameters, defaults to True
- **ignored_tags** – notebook cells to ignore when converting notebooks to py code (separated by ';')

Returns pre-configured function object from a mlrun runtime class

example:

```
import mlrun

# create job function object from notebook code and add doc/metadata
fn = mlrun.code_to_function("file_utils", kind="job",
                            handler="open_archive", image="mlrun/mlrun",
                            description = "this function opens a zip archive into␣
↪a local/mounted folder",
                            categories = ["fileutils"],
                            labels = {"author": "me"})
```

example:

```
import mlrun
from pathlib import Path

# create file
Path("mover.py").touch()

# create nuclio function object from python module call mover.py
fn = mlrun.code_to_function("nuclio-mover", kind="nuclio",
                            filename="mover.py", image="python:3.7",
                            description = "this function moves files from one␣
↪system to another",
```

(continues on next page)

(continued from previous page)

```
                                requirements = ["pandas"],
                                labels = {"author": "me"})
```

mlrun.**get_version**()
get current mlrun version

mlrun.**import_function**(*url=''*, *secrets=None*, *db=''*, *project=None*, *new_name=None*)
Create function object from DB or local/remote YAML file

Function can be imported from function repositories (mlrun marketplace or local db), or be read from a remote URL (http(s), s3, git, v3io, ..) containing the function YAML

special URLs:

```
function marketplace: hub://{name}[:{tag}]
local mlrun db:       db://{project-name}/{name}[:{tag}]
```

examples:

```
function = mlrun.import_function("hub://sklearn_classifier")
function = mlrun.import_function("./func.yaml")
function = mlrun.import_function("https://raw.githubusercontent.com/org/repo/func.
↪yaml")
```

Parameters

- **url** – path/url to marketplace, db or function YAML file
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)
- **db** – optional, mlrun api/db path
- **project** – optional, target project for the function
- **new_name** – optional, override the imported function name

Returns function object

mlrun.**set_environment**(*api_path: Optional[str] = None*, *artifact_path: str = ''*, *project: str = ''*, *access_key: Optional[str] = None*, *user_project=False*)
set and test default config for: api path, artifact_path and project

this function will try and read the configuration from the environment/api and merge it with the user provided project name, artifacts path or api path/access_key. it returns the configured artifacts path, this can be used to define sub paths.

Note: the artifact path is an mlrun data uri (e.g. *s3://bucket/path*) and can not be used with file utils.

example:

```
from os import path
artifact_path = set_environment(project='my-project')
data_subpath = path.join(artifact_path, 'data')
```

Parameters

- **api_path** – location/url of mlrun api service
- **artifact_path** – path/url for storing experiment artifacts
- **project** – default project name

- **access_key** – set the remote cluster access key (V3IO_ACCESS_KEY)
- **user_project** – add the current user name to the provided project name (making it unique per user)

Returns default project name actual artifact path/url, can be used to create subpaths per task or group of artifacts

mlrun.artifacts

mlrun.artifacts.**get_model**(*model_dir*, *suffix=''*)

return model file, model spec object, and list of extra data items

this function will get the model file, metadata, and extra data the returned model file is always local, when using remote urls (such as v3io://, s3://, store://, ..) it will be copied locally.

returned extra data dict (of key, DataItem objects) allow reading additional model files/objects e.g. use DataItem.get() or .download(target) .as_df() to read

example:

```
model_file, model_artifact, extra_data = get_model(models_path, suffix='.pkl')
model = load(open(model_file, "rb"))
categories = extra_data['categories'].as_df()
```

Parameters

- **model_dir** – model dir or artifact path (store://..) or DataItem
- **suffix** – model filename suffix (when using a dir)

Returns model filename, model artifact object, extra data dict

mlrun.artifacts.**update_model**(*model_artifact*, *parameters: Optional[dict] = None*, *metrics: Optional[dict] = None*, *extra_data: Optional[dict] = None*, *inputs: Optional[List[mlrun.features.Feature]] = None*, *outputs: Optional[List[mlrun.features.Feature]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *key_prefix: str = ''*, *labels: Optional[dict] = None*, *write_spec_copy=True*, *store_object: bool = True*)

Update model object attributes

this method will edit or add attributes to a model object

example:

```
update_model(model_path, metrics={'speed': 100},
             extra_data={'my_data': b'some text', 'file': 's3://mybucket/..'})
```

Parameters

- **model_artifact** – model artifact object or path (store://..) or DataItem
- **parameters** – parameters dict
- **metrics** – model metrics e.g. accuracy
- **extra_data** – extra data items key, value dict (value can be: path string | bytes | artifact)
- **inputs** – list of input features (feature vector schema)

- **outputs** – list of output features (output vector schema)
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **key_prefix** – key prefix to add to metrics and extra data items
- **labels** – metadata labels
- **write_spec_copy** – write a YAML copy of the spec to the target dir
- **store_object** – Whether to store the model artifact updated.

mlrun.config

Configuration system.

Configuration can be in either a configuration file specified by MLRUN_CONFIG_FILE environment variable or by environment variables.

Environment variables are in the format "MLRUN_httpdb__port=8080". This will be mapped to config.httpdb.port. Values should be in JSON format.

class mlrun.config.**Config**(*cfg=None*)

Bases: object

property dask_kfp_image

See kfp_image property docstring for why we're defining this property

property dbpath

static decode_base64_config_and_load_to_object(*attribute_path: str*, *expected_type=<class 'dict'>*)

decodes and loads the config attribute to expected type :param attribute_path: the path in the default_config e.g preemptible_nodes.node_selector :param expected_type: the object type valid values are : *dict*, *list* etc... :return: the expected type instance

dump_yaml(*stream=None*)

classmethod from_dict(*dict_*)

static get_build_args()

get_default_function_node_selector() → dict

static get_default_function_pod_requirement_resources(*requirement: str*, *with_gpu: bool = True*)

Parameters

- **requirement** – kubernetes requirement resource one of the following : requests, limits
- **with_gpu** – whether to return requirement resources with nvidia.com/gpu field (e.g you cannot specify GPU requests without specifying GPU limits) https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

Returns a dict containing the defaults resources (cpu, memory, nvidia.com/gpu)

get_default_function_pod_resources()

static get_hub_url()

static get_parsed_igz_version() → Optional[semver.VersionInfo]

get_preemptible_node_selector() → dict

get_preemptible_tolerations() → list

static get_storage_auto_mount_params()

static get_valid_function_priority_class_names()

property iguazio_api_url
we want to be able to run with old versions of the service who runs the API (which doesn't configure this value) so we're doing best effort to try and resolve it from other configurations TODO: Remove this hack when 0.6.x is old enough

static is_pip_ca_configured()

is_preemption_nodes_configured()

property kfp_image
When this configuration is not set we want to set it to mlrun/mlrun, but we need to use the enrich_image method. The problem is that the mlrun.utils.helpers module is importing the config (this) module, so we must import the module inside this function (and not on initialization), and then calculate this property value here.

static reload()

resolve_kfp_url(*namespace=None*)

static resolve_ui_url()

to_dict()

update(*cfg*)

property version

mlrun.config.**read_env**(*env=None*, *prefix='MLRUN_'*)
Read configuration from environment

mlrun.datastore

class mlrun.datastore.**BigQuerySource**(*name: str = ''*, *table: Optional[str] = None*, *max_results_for_table: Optional[int] = None*, *query: Optional[str] = None*, *materialization_dataset: Optional[str] = None*, *chunksize: Optional[int] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time=None*, *end_time=None*, *gcp_project: Optional[str] = None*, *spark_options: Optional[dict] = None*)
Bases: mlrun.datastore.sources.BaseSourceDriver

Reads Google BigQuery query results as input source for a flow.

example:

```
# use sql query
query_string = "SELECT * FROM `the-psf.pypi.downloads20210328` LIMIT 5000"
source = BigQuerySource("bq1", query=query_string,
                        gcp_project="my_project",
                        materialization_dataset="dataviews")
```

(continues on next page)

(continued from previous page)

```
# read a table
source = BigQuerySource("bq2", table="the-psf.pypi.downloads20210328", gcp_
↪project="my_project")
```

Parameters

- **name** – source name
- **table** – table name/path, cannot be used together with query
- **query** – sql query string
- **materialization_dataset** – for query with spark, The target dataset for the materialized view. This dataset should be in same location as the view or the queried tables. must be set to a dataset where the GCP user has table creation permission
- **chunksize** – number of rows per chunk (default large single chunk)
- **key_field** – the column to be used as the key for events. Can be a list of keys.
- **time_field** – the column to be parsed as the timestamp for events. Defaults to None
- **schedule** – string to configure scheduling of the ingestion job. For example '*/30 * * * *' will cause the job to run every 30 minutes
- **gcp_project** – google cloud project name
- **spark_options** – additional spart read options

is_iterator()

kind = 'bigquery'

support_spark = True

support_storey = False

to_dataframe()

to_spark_df(*session*, *named_view=False*)

class mlrun.datastore.**CSVSource**(*name: str = ''*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *parse_dates: Optional[Union[List[int], List[str]]] = None*)

Bases: mlrun.datastore.sources.BaseSourceDriver

Reads CSV file as input source for a flow.

Parameters

- **name** – name of the source
- **path** – path to CSV file
- **key_field** – the CSV field to be used as the key for events. May be an int (field index) or string (field name) if with_header is True. Defaults to None (no key). Can be a list of keys.
- **time_field** – the CSV field to be parsed as the timestamp for events. May be an int (field index) or string (field name) if with_header is True. Defaults to None (no timestamp field). The field will be parsed from isoformat (ISO-8601 as defined in datetime.fromisoformat()). In case the format is not isoformat, timestamp_format (as defined in datetime.strptime()) should be passed in attributes.

- **schedule** – string to configure scheduling of the ingestion job.
- **attributes** – additional parameters to pass to storey. For example: attributes={"timestamp_format": '%Y%m%d%H'}
- **parse_dates** – Optional. List of columns (names or integers, other than time_field) that will be attempted to parse as date column.

get_spark_options()

is_iterator()

kind = 'csv'

support_spark = True

support_storey = True

to_dataframe()

to_spark_df(*session*, *named_view=False*)

to_step(*key_field=None*, *time_field=None*, *context=None*)

class mlrun.datastore.**CSVTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, *entities=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)

return the target data as dataframe

get_spark_options(*key_column=None*, *timestamp_key=None*, *overwrite=True*)

is_offline = True

is_single_file()

kind: str = 'csv'

suffix = '.csv'

support_spark = True

support_storey = True

class mlrun.datastore.**DataItem**(*key: str*, *store: mlrun.datastore.base.DataStore*, *subpath: str*, *url: str = ''*, *meta=None*, *artifact_url=None*)

Bases: object

Data input/output class abstracting access to various local/remote data sources

DataItem objects are passed into functions and can be used inside the function, when a function run completes users can access the run data via the run.artifact(key) which returns a DataItem object. users can also convert a data url (e.g. s3://bucket/key.csv) to a DataItem using *mlrun.get_dataitem(url)*.

Example:

```
# using data item inside a function
def my_func(context, data: DataItem):
    df = data.as_df()


# reading run results using DataItem (run.artifact())
train_run = train_iris_func.run(inputs={'dataset': dataset},
                                params={'label_column': 'label'})

train_run.artifact('confusion-matrix').show()
test_set = train_run.artifact('test_set').as_df()

# create and use DataItem from uri
data = mlrun.get_dataitem('http://xyz/data.json').get()
```

property artifact_url
DataItem artifact url (when its an artifact) or url for simple dataitems

as_df(*columns=None*, *df_module=None*, *format=''*, ***kwargs*)
return a dataframe object (generated from the dataitem).

Parameters

- **columns** – optional, list of columns to select
- **df_module** – optional, py module used to create the DataFrame (e.g. pd, dd, cudf, ..)
- **format** – file format, if not specified it will be deducted from the suffix

download(*target_path*)
download to the target dir/path

Parameters **target_path** – local target path for the downloaded item

get(*size=None*, *offset=0*, *encoding=None*)
read all or a byte range and return the content

Parameters

- **size** – number of bytes to get
- **offset** – fetch from offset (in bytes)
- **encoding** – encoding (e.g. “utf-8”) for converting bytes to str

property key
DataItem key

property kind
DataItem store kind (file, s3, v3io, ..)

listdir()
return a list of child file names

local()
get the local path of the file, download to tmp first if its a remote object

ls()
return a list of child file names

property meta
Artifact Metadata, when the DataItem is read from the artifacts store

open(*mode*)
return fsspec file handler, if supported

put(*data*, *append=False*)
write/upload the data, append is only supported by some datastores

Parameters

- **data** – data (bytes/str) to write
- **append** – append data to the end of the object, NOT SUPPORTED BY SOME OBJECT STORES!

show(*format=None*)
show the data object content in Jupyter

Parameters **format** – format to use (when there is no/wrong suffix), e.g. 'png'

stat()
return FileStats class (size, modified, content_type)

property store
DataItem store object

property suffix
DataItem suffix (file extension) e.g. '.png'

upload(*src_path*)
upload the source file (src_path)

Parameters **src_path** – source file path to read from and upload

property url
//bucket/path

Type DataItem url e.g. /dir/path, s3

class mlrun.datastore.**HttpSource**(*name: Optional[str] = None*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *workers: Optional[int] = None*)
Bases: mlrun.datastore.sources.OnlineSource

add_nuclio_trigger(*function*)

kind = 'http'

class mlrun.datastore.**KafkaSource**(*brokers='localhost:9092'*, *topics='topic'*, *group='serving'*, *initial_offset='earliest'*, *partitions=None*, *sasl_user=None*, *sasl_pass=None*, ***kwargs*)
Bases: mlrun.datastore.sources.OnlineSource

Sets kafka source for the flow

Sets kafka source for the flow

Parameters

- **brokers** – list of broker IP addresses
- **topics** – list of topic names on which to listen.
- **group** – consumer group. Default "serving"
- **initial_offset** – from where to consume the stream. Default earliest

- **partitions** – Optional, A list of partitions numbers for which the function receives events.
- **sasl_user** – Optional, user name to use for sasl authentications
- **sasl_pass** – Optional, password to use for sasl authentications

add_nuclio_trigger(*function*)

kind = 'kafka'

class mlrun.datastore.**NoSqlTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, ***kwargs*)
return the target data as dataframe

get_dask_options()

get_spark_options(*key_column=None*, *timestamp_key=None*, *overwrite=True*)

get_table_object()
get storey Table object

is_online = True

is_table = True

kind: str = 'nosql'

prepare_spark_df(*df*)

support_append = True

support_spark = True

support_storey = True

write_dataframe(*df*, *key_column=None*, *timestamp_key=None*, *chunk_id=0*, ***kwargs*)

class mlrun.datastore.**ParquetSource**(*name: str = ''*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time: Optional[Union[datetime.datetime, str]] = None*, *end_time: Optional[Union[datetime.datetime, str]] = None*)

Bases: mlrun.datastore.sources.BaseSourceDriver

Reads Parquet file/dir as input source for a flow.

Parameters

- **name** – name of the source
- **path** – path to Parquet file or directory

- **key_field** – the column to be used as the key for events. Can be a list of keys.
- **time_field** – the column to be parsed as the timestamp for events. Defaults to None
- **start_filter** – datetime. If not None, the results will be filtered by partitions and 'filter_column' > start_filter. Default is None
- **end_filter** – datetime. If not None, the results will be filtered by partitions 'filter_column' <= end_filter. Default is None
- **filter_column** – Optional. if not None, the results will be filtered by this column and start_filter & end_filter
- **schedule** – string to configure scheduling of the ingestion job. For example '*/30 * * * *' will cause the job to run every 30 minutes
- **attributes** – additional parameters to pass to storey.

get_spark_options()

kind = 'parquet'

support_spark = True

support_storey = True

to_dataframe()

to_step(*key_field=None*, *time_field=None*, *start_time=None*, *end_time=None*, *context=None*)

class mlrun.datastore.**ParquetTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: Optional[bool] = None*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = 10000*, *flush_after_seconds: Optional[int] = 900*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

parquet target storage driver, used to materialize feature set/vector data into parquet files

Parameters

- **name** – optional, target name. By default will be called ParquetTarget
- **path** – optional, Output path. Can be either a file or directory. This parameter is forwarded as-is to pandas.DataFrame.to_parquet(). Default location v3io:///projects/{project}/FeatureStore/{name}/parquet/
- **attributes** – optional, extra attributes for storey.ParquetTarget
- **after_step** – optional, after what step in the graph to add the target
- **columns** – optional, which columns from data to write
- **partitioned** – optional, whether to partition the file, False by default, if True without passing any other partition field, the data will be partitioned by /year/month/day/hour
- **key_bucketing_number** – optional, None by default will not partition by key, 0 will partition by the key as is, any other number X will create X partitions and hash the keys to one of them
- **partition_cols** – optional, name of columns from the data to partition by

- **time_partitioning_granularity** – optional. the smallest time unit to partition the data by. For example "hour" will yield partitions of the format /year/month/day/hour
- **max_events** – optional. Maximum number of events to write at a time. All events will be written on flow termination, or after flush_after_seconds (if flush_after_seconds is set). Default 10k events
- **flush_after_seconds** – optional. Maximum number of seconds to hold events before they are written. All events will be written on flow termination, or after max_events are accumulated (if max_events is set).

  Default 15 minutes

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *feature_set_status=None*)

as_df(*columns=None*, *df_module=None*, *entities=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)
return the target data as dataframe

get_dask_options()

get_spark_options(*key_column=None*, *timestamp_key=None*, *overwrite=True*)

is_offline = True

is_single_file()

kind: str = 'parquet'

support_append = True

support_dask = True

support_spark = True

support_storey = True

class mlrun.datastore.**StreamSource**(*name='stream'*, *group='serving'*, *seek_to='earliest'*, *shards=1*, *retention_in_hours=24*, *extra_attributes: Optional[dict] = None*, ***kwargs*)

Bases: mlrun.datastore.sources.OnlineSource

Sets stream source for the flow. If stream doesn't exist it will create it

Sets stream source for the flow. If stream doesn't exist it will create it

Parameters

- **name** – stream name. Default "stream"
- **group** – consumer group. Default "serving"
- **seek_to** – from where to consume the stream. Default earliest
- **shards** – number of shards in the stream. Default 1
- **retention_in_hours** – if stream doesn't exist and it will be created set retention time. Default 24h
- **extra_attributes** – additional nuclio trigger attributes (key/value dict)

add_nuclio_trigger(*function*)

kind = 'v3ioStream'

class mlrun.datastore.**StreamTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, ***kwargs*)

return the target data as dataframe

is_online = False

is_table = False

kind: str = 'stream'

support_append = True

support_spark = False

support_storey = True

mlrun.datastore.**get_store_resource**(*uri*, *db=None*, *secrets=None*, *project=None*)

get store resource object by uri

mlrun.db

class mlrun.db.httpdb.**HTTPRunDB**(*base_url*, *user=''*, *password=''*, *token=''*)

Bases: mlrun.db.base.RunDBInterface

Interface for accessing and manipulating the *mlrun* persistent store, maintaining the full state and catalog of objects that MLRun uses. The *HTTPRunDB* class serves as a client-side proxy to the MLRun API service which maintains the actual data-store, accesses the server through REST APIs.

The class provides functions for accessing and modifying the various objects that are used by MLRun in its operation. The functions provided follow some standard guidelines, which are:

- Every object in MLRun exists in the context of a project (except projects themselves). When referencing an object through any API, a project name must be provided. The default for most APIs is for an empty project name, which will be replaced by the name of the default project (usually `default`). Therefore, if performing an API to list functions, for example, and not providing a project name - the result will not be functions from all projects but rather from the `default` project.
- Many objects can be assigned labels, and listed/queried by label. The label parameter for query APIs allows for listing objects that:
  - Have a specific label, by asking for `label="<label_name>"`. In this case the actual value of the label doesn't matter and every object with that label will be returned
  - Have a label with a specific value. This is done by specifying `label="<label_name>=<label_value>"`. In this case only objects whose label matches the value will be returned

- Most objects have a `create` method as well as a `store` method. Create can only be called when such an does not exist yet, while store allows for either creating a new object or overwriting an existing object.
- Some objects have a `versioned` option, in which case overwriting the same object with a different version of it does not delete the previous version, but rather creates a new version of the object and keeps both versions. Versioned objects usually have a `uid` property which is based on their content and allows to reference a specific version of an object (other than tagging objects, which also allows for easy referencing).
- Many objects have both a `store` function and a `patch` function. These are used in the same way as the corresponding REST verbs - a `store` is passed a full object and will basically perform a PUT operation, replacing the full object (if it exists) while `patch` receives just a dictionary containing the differences to be applied to the object, and will merge those changes to the existing object. The `patch` operation also has a strategy assigned to it which determines how the merge logic should behave. The strategy can be either `replace` or `additive`. For further details on those strategies, refer to https://pypi.org/project/mergedeep/

abort_run(*uid*, *project=''*, *iter=0*)
Abort a running run - will remove the run's runtime resources and mark its state as aborted

api_call(*method*, *path*, *error=None*, *params=None*, *body=None*, *json=None*, *headers=None*, *timeout=45*, *version=None*)
Perform a direct REST API call on the *mlrun* API server.

> **Caution:** For advanced usage - prefer using the various APIs exposed through this class, rather than directly invoking REST calls.

Parameters

- **method** – REST method (POST, GET, PUT...)
- **path** – Path to endpoint executed, for example `"projects"`
- **error** – Error to return if API invocation fails
- **body** – Payload to be passed in the call. If using JSON objects, prefer using the `json` param
- **json** – JSON payload to be passed in the call
- **headers** – REST headers, passed as a dictionary: `{"<header-name>": "<header-value>"}`
- **timeout** – API call timeout
- **version** – API version to use, None (the default) will mean to use the default value from config, for un-versioned api set an empty string.

Returns Python HTTP response object

connect(*secrets=None*)
Connect to the MLRun API server. Must be called prior to executing any other method. The code utilizes the URL for the API server from the configuration - `mlconf.dbpath`.

For example:

```
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'
db = get_run_db().connect()
```

create_feature_set(*feature_set: Union[dict, mlrun.api.schemas.feature_store.FeatureSet]*, *project=''*, *versioned=True*) → dict

Create a new *FeatureSet* and save in the *mlrun* DB. The feature-set must not previously exist in the DB.

Parameters

- **feature_set** – The new *FeatureSet* to create.
- **project** – Name of project this feature-set belongs to.
- **versioned** – Whether to maintain versions for this feature-set. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureSet* object (as dict).

create_feature_vector(*feature_vector: Union[dict, mlrun.api.schemas.feature_store.FeatureVector]*, *project=''*, *versioned=True*) → dict

Create a new *FeatureVector* and save in the *mlrun* DB.

Parameters

- **feature_vector** – The new *FeatureVector* to create.
- **project** – Name of project this feature-vector belongs to.
- **versioned** – Whether to maintain versions for this feature-vector. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureVector* object (as dict).

create_marketplace_source(*source: Union[dict, mlrun.api.schemas.marketplace.IndexedMarketplaceSource]*)

Add a new marketplace source.

MLRun maintains an ordered list of marketplace sources ("sources"). Each source has its details registered and its order within the list. When creating a new source, the special order `-1` can be used to mark this source as last in the list. However, once the source is in the MLRun list, its order will always be `>0`.

The global marketplace source always exists in the list, and is always the last source (`order = -1`). It cannot be modified nor can it be moved to another order in the list.

The source object may contain credentials which are needed to access the datastore where the source is stored. These credentials are not kept in the MLRun DB, but are stored inside a kubernetes secret object maintained by MLRun. They are not returned through any API from MLRun.

Example:

```
import mlrun.api.schemas

# Add a private source as the last one (will be #1 in the list)
private_source = mlrun.api.schemas.IndexedMarketplaceSource(
    order=-1,
    source=mlrun.api.schemas.MarketplaceSource(
        metadata=mlrun.api.schemas.MarketplaceObjectMetadata(name="priv",␣
↪description="a private source"),
        spec=mlrun.api.schemas.MarketplaceSourceSpec(path="/local/path/to/
↪source", channel="development")
    )
)
db.create_marketplace_source(private_source)

# Add another source as 1st in the list - will push previous one to be #2
another_source = mlrun.api.schemas.IndexedMarketplaceSource(
```

(continues on next page)

(continued from previous page)

```
    order=1,
    source=mlrun.api.schemas.MarketplaceSource(
        metadata=mlrun.api.schemas.MarketplaceObjectMetadata(name="priv-2",␣
↪description="another source"),
        spec=mlrun.api.schemas.MarketplaceSourceSpec(
            path="/local/path/to/source/2",
            channel="development",
            credentials={...}
        )
    )
)
db.create_marketplace_source(another_source)
```

Parameters **source** – The source and its order, of type `IndexedMarketplaceSource`, or in dictionary form.

Returns The source object as inserted into the database, with credentials stripped.

create_or_patch_model_endpoint(*project: str*, *endpoint_id: str*, *model_endpoint: mlrun.api.schemas.model_endpoints.ModelEndpoint*, *access_key: Optional[str] = None*)

Creates or updates a KV record with the given model_endpoint record

Parameters

- **project** – The name of the project
- **endpoint_id** – The id of the endpoint
- **model_endpoint** – An object representing the model endpoint
- **access_key** – V3IO access key, when None, will be look for in environ

create_project(*project: Union[dict, mlrun.projects.project.MlrunProject, mlrun.api.schemas.project.Project]*) → mlrun.projects.project.MlrunProject

Create a new project. A project with the same name must not exist prior to creation.

create_project_secrets(*project: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = <SecretProviderName.kubernetes: 'kubernetes'>*, *secrets: Optional[dict] = None*)

Create project-context secrets using either `vault` or `kubernetes` provider. When using with Vault, this will create needed Vault structures for storing secrets in project-context, and store a set of secret values. The method generates Kubernetes service-account and the Vault authentication structures that are required for function Pods to authenticate with Vault and be able to extract secret values passed as part of their context.

Note: This method used with Vault is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

When used with Kubernetes, this will make sure that the project-specific k8s secret is created, and will populate it with the secrets provided, replacing their values if they exist.

Parameters

- **project** – The project context for which to generate the infra and store secrets.
- **provider** – The name of the secrets-provider to work with. Accepts a *SecretProviderName* enum.

- **secrets** – A set of secret values to store. Example:

```
secrets = {'password': 'myPassw0rd', 'aws_key': '111222333'}
db.create_project_secrets(
    "project1",
    provider=mlrun.api.schemas.SecretProviderName.kubernetes,
    secrets=secrets
)
```

create_schedule(*project: str*, *schedule: mlrun.api.schemas.schedule.ScheduleInput*)

Create a new schedule on the given project. The details on the actual object to schedule as well as the schedule itself are within the schedule object provided. The `ScheduleCronTrigger` follows the guidelines in https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html. It also supports a `from_crontab()` function that accepts a crontab-formatted string (see https://en.wikipedia.org/wiki/Cron for more information on the format).

Example:

```
from mlrun.api import schemas

# Execute the get_data_func function every Tuesday at 15:30
schedule = schemas.ScheduleInput(
    name="run_func_on_tuesdays",
    kind="job",
    scheduled_object=get_data_func,
    cron_trigger=schemas.ScheduleCronTrigger(day_of_week='tue', hour=15,␣
↪minute=30),
)
db.create_schedule(project_name, schedule)
```

create_user_secrets(*user: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = <SecretProviderName.vault: 'vault'>*, *secrets: Optional[dict] = None*)

Create user-context secret in Vault. Please refer to *create_project_secrets()* for more details and status of this functionality.

Note: This method is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **user** – The user context for which to generate the infra and store secrets.
- **provider** – The name of the secrets-provider to work with. Currently only `vault` is supported.
- **secrets** – A set of secret values to store within the Vault.

del_artifact(*key*, *tag=None*, *project=''*)

Delete an artifact.

del_artifacts(*name=None*, *project=None*, *tag=None*, *labels=None*, *days_ago=0*)

Delete artifacts referenced by the parameters.

Parameters

- **name** – Name of artifacts to delete. Note that this is a like query, and is case-insensitive. See *list_artifacts()* for more details.

- **project** – Project that artifacts belong to.
- **tag** – Choose artifacts who are assigned this tag.
- **labels** – Choose artifacts which are labeled.
- **days_ago** – This parameter is deprecated and not used.

del_run(*uid*, *project=''*, *iter=0*)
Delete details of a specific run from DB.

Parameters

- **uid** – Unique ID for the specific run to delete.
- **project** – Project that the run belongs to.
- **iter** – Iteration within a specific task.

del_runs(*name=None*, *project=None*, *labels=None*, *state=None*, *days_ago=0*)
Delete a group of runs identified by the parameters of the function.

Example:

```
db.del_runs(state='completed')
```

Parameters

- **name** – Name of the task which the runs belong to.
- **project** – Project to which the runs belong.
- **labels** – Filter runs that are labeled using these specific label values.
- **state** – Filter only runs which are in this state.
- **days_ago** – Filter runs whose start time is newer than this parameter.

delete_feature_set(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureSet* object from the DB. If `tag` or `uid` are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is `name` will be deleted.

delete_feature_vector(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureVector* object from the DB. If `tag` or `uid` are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is `name` will be deleted.

delete_function(*name: str*, *project: str = ''*)
Delete a function belonging to a specific project.

delete_marketplace_source(*source_name: str*)
Delete a marketplace source from the DB. The source will be deleted from the list, and any following sources will be promoted - for example, if the 1st source is deleted, the 2nd source will become #1 in the list. The global marketplace source cannot be deleted.

Parameters **source_name** – Name of the marketplace source to delete.

delete_model_endpoint_record(*project: str*, *endpoint_id: str*, *access_key: Optional[str] = None*)
Deletes the KV record of a given model endpoint, project and endpoint_id are used for lookup

Parameters

- **project** – The name of the project

- **endpoint_id** – The id of the endpoint
- **access_key** – V3IO access key, when None, will be look for in environ

delete_project(*name: str*, *deletion_strategy: Union[str, mlrun.api.schemas.constants.DeletionStrategy] = <DeletionStrategy.restricted: 'restricted'>*)

Delete a project.

Parameters

- **name** – Name of the project to delete.
- **deletion_strategy** – How to treat child objects of the project. Possible values are:
  - `restrict` (default) - Project must not have any child objects when deleted. If using this mode while child objects exist, the operation will fail.
  - `cascade` - Automatically delete all child objects when deleting the project.

delete_project_secrets(*project: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = <SecretProviderName.kubernetes: 'kubernetes'>*, *secrets: Optional[List[str]] = None*)

Delete project-context secrets from Kubernetes.

Parameters

- **project** – The project name.
- **provider** – The name of the secrets-provider to work with. Currently only `kubernetes` is supported.
- **secrets** – A list of secret names to delete. An empty list will delete all secrets assigned to this specific project.

delete_runtime(*kind: str*, *label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)

Deprecated use *delete_runtime_resources()* (with kind filter) instead

delete_runtime_object(*kind: str*, *object_id: str*, *label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)

Deprecated use *delete_runtime_resources()* (with kind and object_id filter) instead

delete_runtime_resources(*project: Optional[str] = None*, *label_selector: Optional[str] = None*, *kind: Optional[str] = None*, *object_id: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*) → Dict[str, Dict[str, mlrun.api.schemas.runtime_resource.RuntimeResources]]

Delete all runtime resources which are in terminal state.

Parameters **project** – Delete only runtime resources of a specific project, by default None, which will delete only

from the projects you're authorized to delete from. :param label_selector: Delete only runtime resources matching the label selector. :param kind: The kind of runtime to delete. May be one of `['dask', 'job', 'spark', 'remote-spark', 'mpijob']` :param object_id: The identifier of the mlrun object to delete its runtime resources. for most function runtimes, runtime resources are per Run, for which the identifier is the Run's UID. For dask runtime, the runtime resources are per Function, for which the identifier is the Function's name. :param force: Force deletion - delete the runtime resource even if it's not in terminal state or if the grace period didn't pass. :param grace_period: Grace period given to the runtime resource before they are actually removed, counted from the moment they moved to terminal state.

Returns `GroupedByProjectRuntimeResourcesOutput` listing the runtime resources that were removed.

delete_runtimes(*label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)
Deprecated use *delete_runtime_resources()* instead

delete_schedule(*project: str*, *name: str*)
Delete a specific schedule by name.

static get_api_path_prefix(*version: Optional[str] = None*) → str

Parameters **version** – API version to use, None (the default) will mean to use the default value from mlconf, for un-versioned api set an empty string.

get_background_task(*name: str*) → mlrun.api.schemas.background_task.BackgroundTask
Retrieve updated information on a background task being executed.

get_base_api_url(*path: str*, *version: Optional[str] = None*) → str

get_builder_status(*func: mlrun.runtimes.base.BaseRuntime*, *offset=0*, *logs=True*, *last_log_timestamp=0*, *verbose=False*)
Retrieve the status of a build operation currently in progress.

Parameters

- **func** – Function object that is being built.
- **offset** – Offset into the build logs to retrieve logs from.
- **logs** – Should build logs be retrieved.
- **last_log_timestamp** – Last timestamp of logs that were already retrieved. Function will return only logs later than this parameter.
- **verbose** – Add verbose logs into the output.

Returns

The following parameters:

- Text of builder logs.
- Timestamp of last log retrieved, to be used in subsequent calls to this function.

The function also updates internal members of the `func` object to reflect build process info.

get_feature_set(*name: str*, *project: str = ''*, *tag: Optional[str] = None*, *uid: Optional[str] = None*) → mlrun.feature_store.feature_set.FeatureSet
Retrieve a ~mlrun.feature_store.FeatureSet` object. If both `tag` and `uid` are not specified, then the object tagged `latest` will be retrieved.

Parameters

- **name** – Name of object to retrieve.
- **project** – Project the FeatureSet belongs to.
- **tag** – Tag of the specific object version to retrieve.
- **uid** – uid of the object to retrieve (can only be used for versioned objects).

get_feature_vector(*name: str*, *project: str = ''*, *tag: Optional[str] = None*, *uid: Optional[str] = None*) → mlrun.feature_store.feature_vector.FeatureVector
Return a specific feature-vector referenced by its tag or uid. If none are provided, `latest` tag will be used.

get_function(*name*, *project=''*, *tag=None*, *hash_key=''*)
Retrieve details of a specific function, identified by its name and potentially a tag or function hash.

get_log(*uid*, *project=''*, *offset=0*, *size=- 1*)
Retrieve a log.

Parameters

- **uid** – Log unique ID
- **project** – Project name for which the log belongs
- **offset** – Retrieve partial log, get up to `size` bytes starting at offset `offset` from beginning of log
- **size** – See `offset`. If set to `-1` (the default) will retrieve all data to end of log.

Returns

The following objects:

- state - The state of the runtime object which generates this log, if it exists. In case no known state exists, this will be `unknown`.
- content - The actual log content.

get_marketplace_catalog(*source_name: str*, *channel: Optional[str] = None*, *version: Optional[str] = None*, *tag: Optional[str] = None*, *force_refresh: bool = False*)
Retrieve the item catalog for a specified marketplace source. The list of items can be filtered according to various filters, using item's metadata to filter.

Parameters

- **source_name** – Name of the source.
- **channel** – Filter items according to their channel. For example `development`.
- **version** – Filter items according to their version.
- **tag** – Filter items based on tag.
- **force_refresh** – Make the server fetch the catalog from the actual marketplace source, rather than rely on cached information which may exist from previous get requests. For example, if the source was re-built, this will make the server get the updated information. Default is `False`.

Returns `MarketplaceCatalog` object, which is essentially a list of `MarketplaceItem` entries.

get_marketplace_item(*source_name: str*, *item_name: str*, *channel: str = 'development'*, *version: Optional[str] = None*, *tag: str = 'latest'*, *force_refresh: bool = False*)
Retrieve a specific marketplace item.

Parameters

- **source_name** – Name of source.
- **item_name** – Name of the item to retrieve, as it appears in the catalog.
- **channel** – Get the item from the specified channel. Default is `development`.
- **version** – Get a specific version of the item. Default is `None`.
- **tag** – Get a specific version of the item identified by tag. Default is `latest`.

- **force_refresh** – Make the server fetch the information from the actual marketplace source, rather than rely on cached information. Default is `False`.

Returns `MarketplaceItem`.

get_marketplace_source(*source_name: str*)
Retrieve a marketplace source from the DB.

Parameters **source_name** – Name of the marketplace source to retrieve.

get_model_endpoint(*project: str*, *endpoint_id: str*, *start: Optional[str] = None*, *end: Optional[str] = None*, *metrics: Optional[List[str]] = None*, *feature_analysis: bool = False*, *access_key: Optional[str] = None*) → mlrun.api.schemas.model_endpoints.ModelEndpoint
Returns a ModelEndpoint object with additional metrics and feature related data.

Parameters

- **project** – The name of the project
- **endpoint_id** – The id of the model endpoint
- **metrics** – A list of metrics to return for each endpoint, read more in 'TimeMetric'
- **start** – The start time of the metrics
- **end** – The end time of the metrics
- **feature_analysis** – When True, the base feature statistics and current feature statistics will be added to

the output of the resulting object :param access_key: V3IO access key, when None, will be look for in environ

get_pipeline(*run_id: str*, *namespace: Optional[str] = None*, *timeout: int = 10*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = <PipelinesFormat.summary: 'summary'>*, *project: Optional[str] = None*)
Retrieve details of a specific pipeline using its run ID (as provided when the pipeline was executed).

get_project(*name: str*) → mlrun.projects.project.MlrunProject
Get details for a specific project.

get_project_background_task(*project: str*, *name: str*) → mlrun.api.schemas.background_task.BackgroundTask
Retrieve updated information on a project background task being executed.

get_runtime(*kind: str*, *label_selector: Optional[str] = None*) → Dict
Deprecated use *list_runtime_resources()* (with kind filter) instead

get_schedule(*project: str*, *name: str*, *include_last_run: bool = False*) → mlrun.api.schemas.schedule.ScheduleOutput
Retrieve details of the schedule in question. Besides returning the details of the schedule object itself, this function also returns the next scheduled run for this specific schedule, as well as potentially the results of the last run executed through this schedule.

Parameters

- **project** – Project name.
- **name** – Name of the schedule object to query.
- **include_last_run** – Whether to include the results of the schedule's last run in the response.

invoke_schedule(*project: str*, *name: str*)
Execute the object referenced by the schedule immediately.

kind = 'http'

list_artifact_tags(*project=None*) → List[str]
Return a list of all the tags assigned to artifacts in the scope of the given project.

list_artifacts(*name=None*, *project=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*, *kind: Optional[str] = None*, *category: Optional[Union[str, mlrun.api.schemas.artifact.ArtifactCategories]] = None*) → mlrun.lists.ArtifactList
List artifacts filtered by various parameters.

Examples:

```
# Show latest version of all artifacts in project
latest_artifacts = db.list_artifacts('', tag='latest', project='iris')
# check different artifact versions for a specific artifact
result_versions = db.list_artifacts('results', tag='*', project='iris')
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **project** – Project name.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in *HTTPRunDB*.
- **until** – Not in use in *HTTPRunDB*.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.
- **kind** – Return artifacts of the requested kind.
- **category** – Return artifacts of the requested category.

list_entities(*project: str*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *labels: Optional[List[str]] = None*) → List[dict]
Retrieve a list of entities and their mapping to the containing feature-sets. This function is similar to the *list_features()* function, and uses the same logic. However, the entities are matched against the name rather than the features.

list_feature_sets(*project: str = ''*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *state: Optional[str] = None*, *entities: Optional[List[str]] = None*, *features: Optional[List[str]] = None*, *labels: Optional[List[str]] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.FeatureStorePartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = <OrderType.desc: 'desc'>*) → List[mlrun.feature_store.feature_set.FeatureSet]
Retrieve a list of feature-sets matching the criteria provided.

Parameters

- **project** – Project name.
- **name** – Name of feature-set to match. This is a like query, and is case-insensitive.
- **tag** – Match feature-sets with specific tag.
- **state** – Match feature-sets with a specific state.
- **entities** – Match feature-sets which contain entities whose name is in this list.
- **features** – Match feature-sets which contain features whose name is in this list.
- **labels** – Match feature-sets which have these labels.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed value are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.

Returns List of matching *FeatureSet* objects.

list_feature_vectors(*project: str = ''*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *state: Optional[str] = None*, *labels: Optional[List[str]] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.FeatureStorePartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = <OrderType.desc: 'desc'>*) → List[mlrun.feature_store.feature_vector.FeatureVector]

Retrieve a list of feature-vectors matching the criteria provided.

Parameters

- **project** – Project name.
- **name** – Name of feature-vector to match. This is a like query, and is case-insensitive.
- **tag** – Match feature-vectors with specific tag.
- **state** – Match feature-vectors with a specific state.
- **labels** – Match feature-vectors which have these labels.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed values are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.

Returns List of matching *FeatureVector* objects.

list_features(*project: str*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *entities: Optional[List[str]] = None*, *labels: Optional[List[str]] = None*) → List[dict]

List feature-sets which contain specific features. This function may return multiple versions of the same feature-set if a specific tag is not requested. Note that the various filters of this function actually refer to the feature-set object containing the features, not to the features themselves.

Parameters

- **project** – Project which contains these features.
- **name** – Name of the feature to look for. The name is used in a like query, and is not case-sensitive. For example, looking for `feat` will return features which are named `MyFeature` as well as `defeat`.
- **tag** – Return feature-sets which contain the features looked for, and are tagged with the specific tag.
- **entities** – Return only feature-sets which contain an entity whose name is contained in this list.
- **labels** – Return only feature-sets which are labeled as requested.

Returns A list of mapping from feature to a digest of the feature-set, which contains the feature-set meta-data. Multiple entries may be returned for any specific feature due to multiple tags or versions of the feature-set.

list_functions(*name=None*, *project=None*, *tag=None*, *labels=None*)

Retrieve a list of functions, filtered by specific criteria.

Parameters

- **name** – Return only functions with a specific name.
- **project** – Return functions belonging to this project. If not specified, the default project is used.
- **tag** – Return function versions with specific tags.
- **labels** – Return functions that have specific labels assigned to them.

Returns List of function objects (as dictionary).

list_marketplace_sources()

List marketplace sources in the MLRun DB.

list_model_endpoints(*project: str*, *model: Optional[str] = None*, *function: Optional[str] = None*, *labels: Optional[List[str]] = None*, *start: str = 'now-1h'*, *end: str = 'now'*, *metrics: Optional[List[str]] = None*, *access_key: Optional[str] = None*, *top_level: bool = False*, *uids: Optional[List[str]] = None*) → mlrun.api.schemas.model_endpoints.ModelEndpointList

Returns a list of ModelEndpointState objects. Each object represents the current state of a model endpoint. This functions supports filtering by the following parameters: 1) model 2) function 3) labels By default, when no filters are applied, all available endpoints for the given project will be listed.

In addition, this functions provides a facade for listing endpoint related metrics. This facade is time-based and depends on the 'start' and 'end' parameters. By default, when the metrics parameter is None, no metrics are added to the output of this function.

Parameters

- **project** – The name of the project
- **model** – The name of the model to filter by
- **function** – The name of the function to filter by

- **labels** – A list of labels to filter by. Label filters work by either filtering a specific value of a label

(i.e. list("key==value")) or by looking for the existence of a given key (i.e. "key") :param metrics: A list of metrics to return for each endpoint, read more in 'TimeMetric' :param start: The start time of the metrics :param end: The end time of the metrics :param access_key: V3IO access key, when None, will be look for in environ :param top_level: if true will return only routers and endpoint that are NOT children of any router :param uids: if passed will return ModelEndpointList of endpoints with uid in uids

list_pipelines(*project: str*, *namespace: Optional[str] = None*, *sort_by: str = ''*, *page_token: str = ''*, *filter_: str = ''*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = <PipelinesFormat.metadata_only: 'metadata_only'>*, *page_size: Optional[int] = None*) → mlrun.api.schemas.pipeline.PipelinesOutput

Retrieve a list of KFP pipelines. This function can be invoked to get all pipelines from all projects, by specifying `project=*`, in which case pagination can be used and the various sorting and pagination properties can be applied. If a specific project is requested, then the pagination options cannot be used and pagination is not applied.

Parameters

- **project** – Project name. Can be * for query across all projects.
- **namespace** – Kubernetes namespace in which the pipelines are executing.
- **sort_by** – Field to sort the results by.
- **page_token** – Use for pagination, to retrieve next page.
- **filter** – Kubernetes filter to apply to the query, can be used to filter on specific object fields.
- **format** – Result format. Can be one of:
  - `full` - return the full objects.
  - `metadata_only` (default) - return just metadata of the pipelines objects.
  - `name_only` - return just the names of the pipeline objects.
- **page_size** – Size of a single page when applying pagination.

list_project_secret_keys(*project: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = <SecretProviderName.kubernetes: 'kubernetes'>*, *token: Optional[str] = None*) → mlrun.api.schemas.secret.SecretKeysData

Retrieve project-context secret keys from Vault or Kubernetes.

Note: This method for Vault functionality is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **project** – The project name.
- **provider** – The name of the secrets-provider to work with. Accepts a *SecretProviderName* enum.
- **token** – Vault token to use for retrieving secrets. Only in use if `provider` is `vault`. Must be a valid Vault token, with permissions to retrieve secrets of the project in question.

list_project_secrets(*project: str*, *token: Optional[str] = None*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = <SecretProviderName.kubernetes: 'kubernetes'>*, *secrets: Optional[List[str]] = None*) → mlrun.api.schemas.secret.SecretsData

Retrieve project-context secrets from Vault.

Note: This method for Vault functionality is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **project** – The project name.
- **token** – Vault token to use for retrieving secrets. Must be a valid Vault token, with permissions to retrieve secrets of the project in question.
- **provider** – The name of the secrets-provider to work with. Currently only `vault` is accepted.
- **secrets** – A list of secret names to retrieve. An empty list `[]` will retrieve all secrets assigned to this specific project. `kubernetes` provider only supports an empty list.

list_projects(*owner: Optional[str] = None*, *format_: Union[str, mlrun.api.schemas.project.ProjectsFormat] = <ProjectsFormat.full: 'full'>*, *labels: Optional[List[str]] = None*, *state: Optional[Union[str, mlrun.api.schemas.project.ProjectState]] = None*) → List[Union[mlrun.projects.project.MlrunProject, str]]

Return a list of the existing projects, potentially filtered by specific criteria.

Parameters

- **owner** – List only projects belonging to this specific owner.
- **format** – Format of the results. Possible values are:
  - `full` (default value) - Return full project objects.
  - `name_only` - Return just the names of the projects.
- **labels** – Filter by labels attached to the project.
- **state** – Filter by project's state. Can be either `online` or `archived`.

list_runs(*name=None*, *uid=None*, *project=None*, *labels=None*, *state=None*, *sort=True*, *last=0*, *iter=False*, *start_time_from: Optional[datetime.datetime] = None*, *start_time_to: Optional[datetime.datetime] = None*, *last_update_time_from: Optional[datetime.datetime] = None*, *last_update_time_to: Optional[datetime.datetime] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.RunPartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = <OrderType.desc: 'desc'>*, *max_partitions: int = 0*) → mlrun.lists.RunList

Retrieve a list of runs, filtered by various options. Example:

```
runs = db.list_runs(name='download', project='iris', labels='owner=admin')
# If running in Jupyter, can use the .show() function to display the results
db.list_runs(name='', project=project_name).show()
```

Parameters

- **name** – Name of the run to retrieve.
- **uid** – Unique ID of the run.
- **project** – Project that the runs belongs to.
- **labels** – List runs that have a specific label assigned. Currently only a single label filter can be applied, otherwise result will be empty.
- **state** – List only runs whose state is specified.
- **sort** – Whether to sort the result according to their start time. Otherwise results will be returned by their internal order in the DB (order will not be guaranteed).
- **last** – Deprecated - currently not used.
- **iter** – If `True` return runs from all iterations. Otherwise, return only runs whose `iter` is 0.
- **start_time_from** – Filter by run start time in `[start_time_from, start_time_to]`.
- **start_time_to** – Filter by run start time in `[start_time_from, start_time_to]`.
- **last_update_time_from** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.
- **last_update_time_to** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed values are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.
- **max_partitions** – Maximal number of partitions to include in the result. Default is *0* which means no limit.

list_runtime_resources(*project: Optional[str] = None*, *label_selector: Optional[str] = None*, *kind: Optional[str] = None*, *object_id: Optional[str] = None*, *group_by: Optional[mlrun.api.schemas.runtime_resource.ListRuntimeResourcesGroupByField] = None*) → Union[List[mlrun.api.schemas.runtime_resource.KindRuntimeResources], Dict[str, Dict[str, mlrun.api.schemas.runtime_resource.RuntimeResources]]]

List current runtime resources, which are usually (but not limited to) Kubernetes pods or CRDs. Function applies for runs of type `['dask', 'job', 'spark', 'remote-spark', 'mpijob']`, and will return per runtime kind a list of the runtime resources (which may have already completed their execution).

Parameters **project** – Get only runtime resources of a specific project, by default None, which will return only the

projects you're authorized to see. :param label_selector: A label filter that will be passed to Kubernetes for filtering the results according

to their labels.

Parameters

- **kind** – The kind of runtime to query. May be one of `['dask', 'job', 'spark', 'remote-spark', 'mpijob']`
- **object_id** – The identifier of the mlrun object to query its runtime resources. for most function runtimes,

runtime resources are per Run, for which the identifier is the Run's UID. For dask runtime, the runtime resources are per Function, for which the identifier is the Function's name. :param group_by: Object to group results by. Allowed values are *job* and *project*.

list_runtimes(*label_selector: Optional[str] = None*) → List
Deprecated use *list_runtime_resources()* instead

list_schedules(*project: str*, *name: Optional[str] = None*, *kind: Optional[mlrun.api.schemas.schedule.ScheduleKinds] = None*, *include_last_run: bool = False*) → mlrun.api.schemas.schedule.SchedulesOutput
Retrieve list of schedules of specific name or kind.

Parameters

- **project** – Project name.
- **name** – Name of schedule to retrieve. Can be omitted to list all schedules.
- **kind** – Kind of schedule objects to retrieve, can be either `job` or `pipeline`.
- **include_last_run** – Whether to return for each schedule returned also the results of the last run of that schedule.

patch_feature_set(*name*, *feature_set_update: dict*, *project=''*, *tag=None*, *uid=None*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = <PatchMode.replace: 'replace'>*)
Modify (patch) an existing *FeatureSet* object. The object is identified by its name (and project it belongs to), as well as optionally a `tag` or its `uid` (for versioned object). If both `tag` and `uid` are omitted then the object with tag `latest` is modified.

Parameters

- **name** – Name of the object to patch.
- **feature_set_update** – The modifications needed in the object. This parameter only has the changes in it, not a full object. Example:

```
feature_set_update = {"status": {"processed" : True}}
```

  Will apply the field `status.processed` to the existing object.
- **project** – Project which contains the modified object.
- **tag** – The tag of the object to modify.
- **uid** – uid of the object to modify.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

patch_feature_vector(*name*, *feature_vector_update: dict*, *project=''*, *tag=None*, *uid=None*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = <PatchMode.replace: 'replace'>*)
Modify (patch) an existing *FeatureVector* object. The object is identified by its name (and project it belongs to), as well as optionally a `tag` or its `uid` (for versioned object). If both `tag` and `uid` are omitted then the object with tag `latest` is modified.

Parameters

- **name** – Name of the object to patch.
- **feature_vector_update** – The modifications needed in the object. This parameter only has the changes in it, not a full object.
- **project** – Project which contains the modified object.
- **tag** – The tag of the object to modify.
- **uid** – uid of the object to modify.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

patch_project(*name: str*, *project: dict*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = <PatchMode.replace: 'replace'>*) → mlrun.projects.project.MlrunProject

Patch an existing project object.

Parameters

- **name** – Name of project to patch.
- **project** – The actual changes to the project object.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

read_artifact(*key*, *tag=None*, *iter=None*, *project=''*)

Read an artifact, identified by its key, tag and iteration.

read_run(*uid*, *project=''*, *iter=0*)

Read the details of a stored run from the DB.

Parameters

- **uid** – The run's unique ID.
- **project** – Project name.
- **iter** – Iteration within a specific execution.

remote_builder(*func*, *with_mlrun*, *mlrun_version_specifier=None*, *skip_deployed=False*, *builder_env=None*)

Build the pod image for a function, for execution on a remote cluster. This is executed by the MLRun API server, and creates a Docker image out of the function provided and any specific build instructions provided within. This is a pre-requisite for remotely executing a function, unless using a pre-deployed image.

Parameters

- **func** – Function to build.
- **with_mlrun** – Whether to add MLRun package to the built package. This is not required if using a base image that already has MLRun in it.
- **mlrun_version_specifier** – Version of MLRun to include in the built image.
- **skip_deployed** – Skip the build if we already have an image for the function.
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials)

remote_start(*func_url*) → mlrun.api.schemas.background_task.BackgroundTask

Execute a function remotely, Used for `dask` functions.

Parameters **func_url** – URL to the function to be executed.

Returns A BackgroundTask object, with details on execution process and its status.

remote_status(*project*, *name*, *kind*, *selector*)
Retrieve status of a function being executed remotely (relevant to `dask` functions).

Parameters

- **project** – The project of the function
- **name** – The name of the function
- **kind** – The kind of the function, currently `dask` is supported.
- **selector** – Selector clause to be applied to the Kubernetes status query to filter the results.

store_artifact(*key*, *artifact*, *uid*, *iter=None*, *tag=None*, *project=''*)
Store an artifact in the DB.

Parameters

- **key** – Identifying key of the artifact.
- **artifact** – The actual artifact to store.
- **uid** – A unique ID for this specific version of the artifact.
- **iter** – The task iteration which generated this artifact. If `iter` is not `None` the iteration will be added to the key provided to generate a unique key for the artifact of the specific iteration.
- **tag** – Tag of the artifact.
- **project** – Project that the artifact belongs to.

store_feature_set(*feature_set: Union[dict, mlrun.api.schemas.feature_store.FeatureSet]*, *name=None*, *project=''*, *tag=None*, *uid=None*, *versioned=True*) → dict
Save a *FeatureSet* object in the *mlrun* DB. The feature-set can be either a new object or a modification to existing object referenced by the params of the function.

Parameters

- **feature_set** – The *FeatureSet* to store.
- **project** – Name of project this feature-set belongs to.
- **tag** – The `tag` of the object to replace in the DB, for example `latest`.
- **uid** – The `uid` of the object to replace in the DB. If using this parameter, the modified object must have the same `uid` of the previously-existing object. This cannot be used for non-versioned objects.
- **versioned** – Whether to maintain versions for this feature-set. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureSet* object (as dict).

store_feature_vector(*feature_vector: Union[dict, mlrun.api.schemas.feature_store.FeatureVector]*, *name=None*, *project=''*, *tag=None*, *uid=None*, *versioned=True*) → dict
Store a *FeatureVector* object in the *mlrun* DB. The feature-vector can be either a new object or a modification to existing object referenced by the params of the function.

Parameters

- **feature_vector** – The *FeatureVector* to store.

- **project** – Name of project this feature-vector belongs to.
- **tag** – The `tag` of the object to replace in the DB, for example `latest`.
- **uid** – The `uid` of the object to replace in the DB. If using this parameter, the modified object must have the same `uid` of the previously-existing object. This cannot be used for non-versioned objects.
- **versioned** – Whether to maintain versions for this feature-vector. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureVector* object (as dict).

store_function(*function*, *name*, *project=''*, *tag=None*, *versioned=False*)
Store a function object. Function is identified by its name and tag, and can be versioned.

store_log(*uid*, *project=''*, *body=None*, *append=False*)
Save a log persistently.

Parameters

- **uid** – Log unique ID
- **project** – Project name for which this log belongs
- **body** – The actual log to store
- **append** – Whether to append the log provided in `body` to an existing log with the same `uid` or to create a new log. If set to `False`, an existing log with same `uid` will be overwritten

store_marketplace_source(*source_name: str*, *source: Union[dict, mlrun.api.schemas.marketplace.IndexedMarketplaceSource]*)
Create or replace a marketplace source. For an example of the source format and explanation of the source order logic, please see *create_marketplace_source()*. This method can be used to modify the source itself or its order in the list of sources.

Parameters

- **source_name** – Name of the source object to modify/create. It must match the `source.metadata.name` parameter in the source itself.
- **source** – Source object to store in the database.

Returns The source object as stored in the DB.

store_project(*name: str*, *project: Union[dict, mlrun.projects.project.MlrunProject, mlrun.api.schemas.project.Project]*) → mlrun.projects.project.MlrunProject
Store a project in the DB. This operation will overwrite existing project of the same name if exists.

store_run(*struct*, *uid*, *project=''*, *iter=0*)
Store run details in the DB. This method is usually called from within other *mlrun* flows and not called directly by the user.

submit_job(*runspec*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*)
Submit a job for remote execution.

Parameters

- **runspec** – The runtime object spec (Task) to execute.
- **schedule** – Whether to schedule this job using a Cron trigger. If not specified, the job will be submitted immediately.

submit_pipeline(*project*, *pipeline*, *arguments=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *ttl=None*)
Submit a KFP pipeline for execution.

Parameters

- **project** – The project of the pipeline
- **pipeline** – Pipeline function or path to .yaml/.zip pipeline file.
- **arguments** – A dictionary of arguments to pass to the pipeline.
- **experiment** – A name to assign for the specific experiment.
- **run** – A name for this specific run.
- **namespace** – Kubernetes namespace to execute the pipeline in.
- **artifact_path** – A path to artifacts used by this pipeline.
- **ops** – Transformers to apply on all ops in the pipeline.
- **ttl** – Set the TTL for the pipeline after its completion.

trigger_migrations() → Optional[mlrun.api.schemas.background_task.BackgroundTask]
Trigger migrations (will do nothing if no migrations are needed) and wait for them to finish if actually triggered :returns: `BackgroundTask`.

update_run(*updates: dict*, *uid*, *project=''*, *iter=0*)
Update the details of a stored run in the DB.

update_schedule(*project: str*, *name: str*, *schedule: mlrun.api.schemas.schedule.ScheduleUpdate*)
Update an existing schedule, replace it with the details contained in the schedule object.

verify_authorization(*authorization_verification_input: mlrun.api.schemas.auth.AuthorizationVerificationInput*)
Verifies authorization for the provided action on the provided resource.

Parameters **authorization_verification_input** – Instance of `AuthorizationVerificationInput` that includes all the needed parameters for the auth verification

watch_log(*uid*, *project=''*, *watch=True*, *offset=0*)
Retrieve logs of a running process, and watch the progress of the execution until it completes. This method will print out the logs and continue to periodically poll for, and print, new logs as long as the state of the runtime which generates this log is either `pending` or `running`.

Parameters

- **uid** – The uid of the log object to watch.
- **project** – Project that the log belongs to.
- **watch** – If set to `True` will continue tracking the log as described above. Otherwise this function is practically equivalent to the *get_log()* function.
- **offset** – Minimal offset in the log to watch.

Returns The final state of the log being watched.

class mlrun.api.schemas.secret.**SecretProviderName**(*value*)
Bases: `str`, `enum.Enum`

Enum containing names of valid providers for secrets.

kubernetes = 'kubernetes'

vault = 'vault'

mlrun.execution

class mlrun.execution.**MLClientCtx**(*autocommit=False*, *tmp=''*, *log_stream=None*)
Bases: object

ML Execution Client Context

the context is generated and injected to the function using the function.run() or manually using the *get_or_create_ctx()* call and provides an interface to use run params, metadata, inputs, and outputs

base metadata include: uid, name, project, and iteration (for hyper params) users can set labels and annotations using *set_label()*, *set_annotation()* access parameters and secrets using *get_param()*, *get_secret()* access input data objects using *get_input()* store results, artifacts, and real-time metrics using the *log_result()*, *log_artifact()*, *log_dataset()* and *log_model()* methods

see doc for the individual params and methods

property annotations
dictionary with annotations (read-only)

artifact_subpath(**subpaths*)
subpaths under output path artifacts path

example:

```
data_path=context.artifact_subpath('data')
```

property artifacts
dictionary of artifacts (read-only)

commit(*message: str = ''*, *completed=True*)
save run state and optionally add a commit message

Parameters

- **message** – commit message to save in the run
- **completed** – mark run as completed

classmethod from_dict(*attrs: dict*, *rundb=''*, *autocommit=False*, *tmp=''*, *host=None*, *log_stream=None*, *is_api=False*)
create execution context from dict

get_cached_artifact(*key*)
return an logged artifact from cache (for potential updates)

get_child_context(*with_parent_params=False*, ***params*)
get child context (iteration)

allow sub experiments (epochs, hyper-param, ..) under a parent will create a new iteration, log_xx will update the child only use commit_children() to save all the children and specify the best run

example:

```
def handler(context: mlrun.MLClientCtx, data: mlrun.DataItem):
    df = data.as_df()
    best_accuracy = accuracy_sum = 0
    for param in param_list:
        with context.get_child_context(myparam=param) as child:
            accuracy = child_handler(child, df, **child.parameters)
            accuracy_sum += accuracy
            child.log_result('accuracy', accuracy)
```

(continues on next page)

(continued from previous page)

```
            if accuracy > best_accuracy:
                child.mark_as_best()
                best_accuracy = accuracy

    context.log_result('avg_accuracy', accuracy_sum / len(param_list))
```

Parameters

- **params** – extra (or override) params to parent context
- **with_parent_params** – child will copy the parent parameters and add to them

Returns child context

get_dataitem(*url*)
get mlrun dataitem from url

example:

```
data = context.get_dataitem("s3://my-bucket/file.csv").as_df()
```

get_input(*key: str*, *url: str = ''*)
get an input `DataItem` object, data objects have methods such as .get(), .download(), .url, .. to access the actual data

example:

```
data = context.get_input("my_data").get()
```

get_meta()
Reserved for internal use

get_param(*key: str*, *default=None*)
get a run parameter, or use the provided default if not set

example:

```
p1 = context.get_param("p1", 0)
```

get_project_param(*key: str*, *default=None*)
get a parameter from the run's project's parameters

get_secret(*key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through vault, files, env, ..

example:

```
access_key = context.get_secret("ACCESS_KEY")
```

get_store_resource(*url*)
get mlrun data resource (feature set/vector, artifact, item) from url

example:

```
feature_vector = context.get_store_resource("store://feature-vectors/default/
↪myvec")
dataset = context.get_store_resource("store://artifacts/default/mydata")
```

Parameters **url** – store resource uri/path, store://<type>/<project>/<name>:<version> types: artifacts | feature-sets | feature-vectors

property in_path
default input path for data objects

property inputs
dictionary of input data items (read-only)

property iteration
child iteration index, for hyper parameters

kind = 'run'

property labels
dictionary with labels (read-only)

log_artifact(*item*, *body=None*, *local_path=None*, *artifact_path=None*, *tag=''*, *viewer=None*, *target_path=''*, *src_path=None*, *upload=None*, *labels=None*, *format=None*, *db_key=None*, ***kwargs*)
log an output artifact and optionally upload it to datastore

example:

```
context.log_artifact(
    "some-data",
    body=b"abc is 123",
    local_path="model.txt",
    labels={"framework": "xgboost"},
)
```

Parameters

- **item** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **viewer** – kubeflow viewer type
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **src_path** – deprecated, use local_path
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_dataset(*key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=True*, *labels=None*, *format=''*, *preview=None*, *stats=False*, *db_key=None*, *target_path=''*, *extra_data=None*, ***kwargs*)

log a dataset artifact and optionally upload it to datastore

example:

```
raw_data = {
    "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
    "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
    "age": [42, 52, 36, 24, 73],
    "testScore": [25, 94, 57, 62, 70],
}
df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age",
↪"testScore"])
context.log_dataset("mydf", df=df, stats=True)
```

Parameters

- **key** – artifact key
- **df** – dataframe object
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_iteration_results(*best*, *summary: list*, *task: dict*, *commit=False*)

Reserved for internal use

property log_level

get the logging level, e.g. 'debug', 'info', 'error'

log_metric(*key: str*, *value*, *timestamp=None*, *labels=None*)

TBD, log a real-time time-series metric

log_metrics(*keyvals: dict*, *timestamp=None*, *labels=None*)

TBD, log a set of real-time time-series metrics

log_model(*key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *algorithm=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=True*, *labels=None*, *inputs: Optional[List[mlrun.features.Feature]] = None*, *outputs: Optional[List[mlrun.features.Feature]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *training_set=None*, *label_column: Optional[Union[str, list]] = None*, *extra_data=None*, *db_key=None*, ***kwargs*)

log a model artifact and optionally upload it to datastore

example:

```
context.log_model("model", body=dumps(model),
                  model_file="model.pkl",
                  metrics=context.results,
                  training_set=training_df,
                  label_column='label',
                  feature_vector=feature_vector_uri,
                  labels={"app": "fraud"})
```

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **model_file** – path to the local model file we upload (see also model_dir)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **framework** – name of the ML framework
- **algorithm** – training algorithm name
- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **training_set** – training set dataframe, used to infer inputs & outputs
- **label_column** – which columns in the training set are the label (target) columns
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be abs/relative path string | bytes | artifact object
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_result (*key: str*, *value*, *commit=False*)
log a scalar result value

example:

```
context.log_result('accuracy', 0.85)
```

Parameters

- **key** – result key
- **value** – result value
- **commit** – commit (write to DB now vs wait for the end of the run)

log_results (*results: dict*, *commit=False*)
log a set of scalar result values

example:

```
context.log_results({'accuracy': 0.85, 'loss': 0.2})
```

Parameters

- **results** – key/value dict or results
- **commit** – commit (write to DB now vs wait for the end of the run)

property logger
built-in logger interface

example:

```
context.logger.info("started experiment..", param=5)
```

mark_as_best ()
mark a child as the best iteration result, see .get_child_context()

property out_path
default output path for artifacts

property parameters
dictionary of run parameters (read-only)

property project
project name, runs can be categorized by projects

property results
dictionary of results (read-only)

set_annotation (*key: str*, *value*, *replace: bool = True*)
set/record a specific annotation

example:

```
context.set_annotation("comment", "some text")
```

set_hostname (*host: str*)
update the hostname, for internal use

set_label(*key: str*, *value*, *replace: bool = True*)
set/record a specific label

example:

```
context.set_label("framework", "sklearn")
```

set_logger_stream(*stream*)

set_state(*state: Optional[str] = None*, *error: Optional[str] = None*, *commit=True*)
modify and store the run state or mark an error

Parameters

- **state** – set run state
- **error** – error message (if exist will set the state to error)
- **commit** – will immediately update the state in the DB

property tag
run tag (uid or workflow id if exists)

to_dict()
convert the run context to a dictionary

to_json()
convert the run context to a json buffer

to_yaml()
convert the run context to a yaml buffer

property uid
Unique run id

update_artifact(*artifact_object*)
update an artifact object in the cache and the DB

update_child_iterations(*best_run=0*, *commit_children=False*, *completed=True*)
update children results in the parent, and optionally mark the best

Parameters

- **best_run** – marks the child iteration number (starts from 1)
- **commit_children** – commit all child runs to the db
- **completed** – mark children as completed

mlrun.feature_store

class mlrun.feature_store.**Entity**(*name: Optional[str] = None*, *value_type: Optional[mlrun.data_types.data_types.ValueType] = None*, *description: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data entity (index)

data entity (index key)

Parameters

- **name** – entity name

- **value_type** – type of the entity, e.g. ValueType.STRING, ValueType.INT
- **description** – test description of the entity
- **labels** – a set of key/value labels (tags)

class mlrun.feature_store.**Feature**(*value_type: Optional[str] = None*, *dims: Optional[List[int]] = None*, *description: Optional[str] = None*, *aggregate: Optional[bool] = None*, *name: Optional[str] = None*, *validator=None*, *default: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data feature

data feature

Features can be specified manually or inferred automatically (during ingest/preview)

Parameters

- **value_type** – type of the feature. Use the ValueType constants library e.g. ValueType.STRING, ValueType.INT
- **dims** – list of dimensions for vectors/tensors, e.g. [2, 2]
- **description** – text description of the feature
- **aggregate** – is it an aggregated value
- **name** – name of the feature
- **validator** – feature validation policy
- **default** – default value
- **labels** – a set of key/value labels (tags)

property validator

class mlrun.feature_store.**FeatureSet**(*name: Optional[str] = None*, *description: Optional[str] = None*, *entities: Optional[List[Union[mlrun.features.Entity, str]]] = None*, *timestamp_key: Optional[str] = None*, *engine: Optional[str] = None*)

Bases: mlrun.model.ModelObj

Feature set object, defines a set of features and their data pipeline

Feature set object, defines a set of features and their data pipeline

example:

```
import mlrun.feature_store as fstore
ticks = fstore.FeatureSet("ticks", entities=["stock"], timestamp_key="timestamp")
fstore.ingest(ticks, df)
```

Parameters

- **name** – name of the feature set
- **description** – text description
- **entities** – list of entity (index key) names or Entity
- **timestamp_key** – timestamp column name

- **engine** – name of the processing engine (storey, pandas, or spark), defaults to storey

add_aggregation(*column*, *operations*, *windows*, *period=None*, *name=None*, *step_name=None*, *after=None*, *before=None*, *state_name=None*, *emit_policy: Optional[storey.dtypes.EmitPolicy] = None*)

add feature aggregation rule

example:

```
myset.add_aggregation("ask", ["sum", "max"], "1h", "10m", name="asks")
```

Parameters

- **column** – name of column/field aggregate. Do not name columns starting with either *t_* or *aggr_*. They are reserved for internal use, and the data does not ingest correctly. When using the pandas engine, do not use spaces (` ) *or periods (.*`) in the column names; they cause errors in the ingestion.
- **operations** – aggregation operations, e.g. [‘sum’, ‘std’]
- **windows** – time windows, can be a single window, e.g. ‘1h’, ‘1d’, or a list of same unit windows e.g [‘1h’, ‘6h’] windows are transformed to fixed windows or sliding windows depending whether period parameter provided.
  - Sliding window is fixed-size overlapping windows that slides with time. The window size determines the size of the sliding window and the period determines the step size to slide. Period must be integral divisor of the window size. If the period is not provided then fixed windows is used.
  - Fixed window is fixed-size, non-overlapping, gap-less window. The window is referred to as a tumbling window. In this case, each record on an in-application stream belongs to a specific window. It is processed only once (when the query processes the window to which the record belongs).
- **period** – optional, sliding window granularity, e.g. ’20s’ ‘10m’ ‘3h’ ‘7d’
- **name** – optional, aggregation name/prefix. Must be unique per feature set.If not passed, the column will be used as name.
- **step_name** – optional, graph step name
- **state_name** – *Deprecated* - use step_name instead
- **after** – optional, after which graph step it runs
- **before** – optional, comes before graph step
- **emit_policy** – optional, which emit policy to use when performing the aggregations. Use the derived classes of `storey.EmitPolicy`. The default is to emit every period for Spark engine and emit every event for storey. Currently the only other supported option is to use `emit_policy=storey.EmitEveryEvent()` when using the Spark engine to emit every event

add_entity(*name: str*, *value_type: Optional[mlrun.data_types.data_types.ValueType] = None*, *description: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

add/set an entity (dataset index)

example:

```
import mlrun.feature_store as fstore

ticks = fstore.FeatureSet("ticks",
                entities=["stock"],
                timestamp_key="timestamp")
ticks.add_entity("country",
                mlrun.data_types.ValueType.STRING,
                description="stock country")
ticks.add_entity("year", mlrun.data_types.ValueType.INT16)
ticks.save()
```

Parameters

- **name** – entity name
- **value_type** – type of the entity (default to ValueType.STRING)
- **description** – description of the entity
- **labels** – label tags dict

add_feature(*feature: mlrun.features.Feature*, *name=None*)
add/set a feature

example:

```
import mlrun.feature_store as fstore
from mlrun.features import Feature

ticks = fstore.FeatureSet("ticks",
                entities=["stock"],
                timestamp_key="timestamp")
ticks.add_feature(Feature(value_type=mlrun.data_types.ValueType.STRING,
                description="client consistency"),"ABC01")
ticks.add_feature(Feature(value_type=mlrun.data_types.ValueType.FLOAT,
                description="client volatility"),"SAB")
ticks.save()
```

Parameters

- **feature** – setting of Feature
- **name** – feature name

property fullname
{tag}]

Type full name in the form {project}/{name}[

get_stats_table()
get feature statistics table (as dataframe)

get_target_path(*name=None*)
get the url/path for an offline or specified data target

property graph
feature set transformation graph/DAG

has_valid_source()
check if object's spec has a valid (non empty) source definition

kind = 'FeatureSet'

link_analysis(*name*, *uri*)
add a linked file/artifact (chart, data, ..)

property metadata

plot(*filename=None*, *format=None*, *with_targets=False*, ***kw*)
generate graphviz plot

purge_targets(*target_names: Optional[List[str]] = None*, *silent: bool = False*)
Delete data of specific targets :param target_names: List of names of targets to delete (default: delete all ingested targets) :param silent: Fail silently if target doesn't exist in featureset status

reload(*update_spec=True*)
reload/sync the feature vector status and spec from the DB

save(*tag=''*, *versioned=False*)
save to mlrun db

set_targets(*targets=None*, *with_defaults=True*, *default_final_step=None*, *default_final_state=None*)
set the desired target list or defaults

Parameters

- **targets** – list of target type names ('csv', 'nosql', ..) or target objects CSVTarget(), ParquetTarget(), NoSqlTarget(), StreamTarget(), ..
- **with_defaults** – add the default targets (as defined in the central config)
- **default_final_step** – the final graph step after which we add the target writers, used when the graph branches and the end cant be determined automatically
- **default_final_state** – *Deprecated* - use default_final_step instead

property spec

property status

to_dataframe(*columns=None*, *df_module=None*, *target_name=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)
return featureset (offline) data as dataframe

Parameters

- **columns** – list of columns to select (if not all)
- **df_module** – py module used to create the DataFrame (pd for Pandas, dd for Dask, ..)
- **target_name** – select a specific target (material view)
- **start_time** – filter by start time
- **end_time** – filter by end time
- **time_column** – specify the time column name in the file
- **kwargs** – additional reader (csv, parquet, ..) args

Returns DataFrame

update_targets_for_ingest(*targets: List[*mlrun.model.DataTargetBase*]*, *overwrite: Optional[bool] = None*)

property uri
fully qualified feature set uri

class mlrun.feature_store.**FeatureVector**(*name=None*, *features=None*, *label_feature=None*, *description=None*, *with_indexes=None*)

Bases: mlrun.model.ModelObj

Feature vector, specify selected features, their metadata and material views

Feature vector, specify selected features, their metadata and material views

example:

```
import mlrun.feature_store as fstore
features = ["quotes.bid", "quotes.asks_sum_5h as asks_5h", "stocks.*"]
vector = fstore.FeatureVector("my-vec", features)

# get the vector as a dataframe
df = fstore.get_offline_features(vector).to_dataframe()

# return an online/real-time feature service
svc = fs.get_online_feature_service(vector, impute_policy={"*": "$mean"})
resp = svc.get([{"stock": "GOOG"}])
```

Parameters

- **name** – List of names of targets to delete (default: delete all ingested targets)
- **features** – list of feature to collect to this vector. format [<project>/]<feature_set>.<feature_name or *> [as <alias>]
- **label_feature** – feature name to be used as label data
- **description** – text description of the vector
- **with_indexes** – whether to keep the entity and timestamp columns in the response

get_feature_aliases()

get_stats_table()
get feature statistics table (as dataframe)

get_target_path(*name=None*)

kind = 'FeatureVector'

link_analysis(*name*, *uri*)
add a linked file/artifact (chart, data, ..)

property metadata

parse_features(*offline=True*, *update_stats=False*)
parse and validate feature list (from vector) and add metadata from feature sets

:returns feature_set_objects: cache of used feature set objects feature_set_fields: list of field (name, alias) per featureset

reload(*update_spec=True*)
reload/sync the feature set status and spec from the DB

save(*tag=''*, *versioned=False*)
save to mlrun db

property spec

property status

to_dataframe(*df_module=None*, *target_name=None*)
return feature vector (offline) data as dataframe

property uri
fully qualified feature vector uri

class mlrun.feature_store.**FixedWindowType**(*value*)
Bases: enum.Enum

An enumeration.

CurrentOpenWindow = 1

LastClosedWindow = 2

to_qbk_fixed_window_type()

class mlrun.feature_store.**OfflineVectorResponse**(*merger*)
Bases: object

get_offline_features response object

property status
vector prep job status (ready, running, error)

to_csv(*target_path*, ***kw*)
return results as csv file

to_dataframe(*to_pandas=True*)
return result as dataframe

to_parquet(*target_path*, ***kw*)
return results as parquet file

class mlrun.feature_store.**OnlineVectorService**(*vector*, *graph*, *index_columns*, *impute_policy: Optional[dict] = None*)
Bases: object

get_online_feature_service response object

close()
terminate the async loop

get(*entity_rows: List[Union[dict, list]]*, *as_list=False*)
get feature vector given the provided entity inputs

take a list of input vectors/rows and return a list of enriched feature vectors each input and/or output vector can be a list of values or a dictionary of field names and values, to return the vector as a list of values set the *as_list* to True.

if the input is a list of list (vs a list of dict), the values in the list will correspond to the index/entity values, i.e. [[“GOOG”], [“MSFT”]] means “GOOG” and “MSFT” are the index/entity fields.

example:

```
# accept list of dict, return list of dict
svc = fs.get_online_feature_service(vector)
resp = svc.get([{"name": "joe"}, {"name": "mike"}])

# accept list of list, return list of list
svc = fs.get_online_feature_service(vector, as_list=True)
resp = svc.get([["joe"], ["mike"]])
```

Parameters

- **entity_rows** – list of list/dict with input entity data/rows
- **as_list** – return a list of list (list input is required by many ML frameworks)

initialize()
internal, init the feature service and prep the imputing logic

property status
vector merger function status (ready, running, error)

class mlrun.feature_store.**RunConfig**(*function: Optional[Union[str, mlrun.runtimes.function_reference.FunctionReference, mlrun.runtimes.base.BaseRuntime]] = None*, *local: Optional[bool] = None*, *image: Optional[str] = None*, *kind: Optional[str] = None*, *handler: Optional[str] = None*, *parameters: Optional[dict] = None*, *watch: Optional[bool] = None*, *owner=None*, *credentials: Optional[mlrun.model.Credentials] = None*, *code: Optional[str] = None*, *requirements: Optional[Union[str, List[str]]] = None*, *extra_spec: Optional[dict] = None*)

Bases: object

class for holding function and run specs for jobs and serving functions

class for holding function and run specs for jobs and serving functions

when running feature ingestion or merging tasks we use the RunConfig class to pass the desired function and job configuration. the apply() method is used to set resources like volumes, the with_secret() method adds secrets

Most attributes are optional, if not specified a proper default value will be set

examples:

```
# config for local run emulation
config = RunConfig(local=True)

# config for using empty/default code
config = RunConfig()

# config for using .py/.ipynb file with image and extra package requirements
config = RunConfig("mycode.py", image="mlrun/mlrun", requirements=["spacy"])

# config for using function object
function = mlrun.import_function("hub://some_function")
config = RunConfig(function)
```

Parameters

- **function** – this can be function uri or function object or path to function code (.py/.ipynb) or a FunctionReference the function define the code, dependencies, and resources
- **local** – use True to simulate local job run or mock service
- **image** – function container image
- **kind** – function runtime kind (job, serving, spark, ..), required when function points to code
- **handler** – the function handler to execute (for jobs or nuclio)
- **parameters** – job parameters

- **watch** – in batch jobs will wait for the job completion and print job logs to the console
- **owner** – job owner
- **credentials** – job credentials
- **code** – function source code (as string)
- **requirements** – python requirements file path or list of packages
- **extra_spec** – additional dict with function spec fields/values to add to the function

apply(*modifier*)
apply a modifier to add/set function resources like volumes

example:

```
run_config.apply(mlrun.platforms.auto_mount())
```

copy()

property function

to_function(*default_kind=None*, *default_image=None*)
internal, generate function object

with_secret(*kind*, *source*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in jobs, example:

```
run_config.with_secrets('file', 'file.txt')
run_config.with_secrets('inline', {'key': 'val'})
run_config.with_secrets('env', 'ENV1,ENV2')
run_config.with_secrets('vault', ['secret1', 'secret2'...])
```

Parameters

- **kind** – secret type (file, inline, env, vault)
- **source** – secret data or link (see example)

Returns This (self) object

mlrun.feature_store.**delete_feature_set**(*name*, *project=''*, *tag=None*, *uid=None*, *force=False*)
Delete a *FeatureSet* object from the DB. :param name: Name of the object to delete :param project: Name of the object's project :param tag: Specific object's version tag :param uid: Specific object's uid :param force: Delete feature set without purging its targets

If tag or uid are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is name will be deleted.

mlrun.feature_store.**delete_feature_vector**(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureVector* object from the DB. :param name: Name of the object to delete :param project: Name of the object's project :param tag: Specific object's version tag :param uid: Specific object's uid

If tag or uid are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is name will be deleted.

mlrun.feature_store.**deploy_ingestion_service**(*featureset: Union[mlrun.feature_store.feature_set.FeatureSet, str]*, *source: Optional[mlrun.model.DataSource]* = *None*, *targets: Optional[List[mlrun.model.DataTargetBase]]* = *None*, *name: Optional[str]* = *None*, *run_config: Optional[mlrun.feature_store.common.RunConfig]* = *None*, *verbose=False*)

Start real-time ingestion service using nuclio function

Deploy a real-time function implementing feature ingestion pipeline the source maps to Nuclio event triggers (http, kafka, v3io stream, etc.)

the *run_config* parameter allow specifying the function and job configuration, see: *RunConfig*

example:

```
source = HTTPSource()
func = mlrun.code_to_function("ingest", kind="serving").apply(mount_v3io())
config = RunConfig(function=func)
fs.deploy_ingestion_service(my_set, source, run_config=config)
```

Parameters

- **featureset** – feature set object or uri
- **source** – data source object describing the online or offline source
- **targets** – list of data target objects
- **name** – name for the job/function
- **run_config** – service runtime configuration (function object/uri, resources, etc..)
- **verbose** – verbose log

mlrun.feature_store.**get_feature_set**(*uri*, *project=None*)

get feature set object from the db

Parameters

- **uri** – a feature set uri({project}/{name}[:version])
- **project** – project name if not specified in uri or not using the current/default

mlrun.feature_store.**get_feature_vector**(*uri*, *project=None*)

get feature vector object from the db

Parameters

- **uri** – a feature vector uri({project}/{name}[:version])
- **project** – project name if not specified in uri or not using the current/default

mlrun.feature_store.**get_offline_features**(*feature_vector: Union[str, mlrun.feature_store.feature_vector.FeatureVector]*, *entity_rows=None*, *entity_timestamp_column: Optional[str] = None*, *target: Optional[mlrun.model.DataTargetBase] = None*, *run_config: Optional[mlrun.feature_store.common.RunConfig] = None*, *drop_columns: Optional[List[str]] = None*, *start_time: Optional[Union[str, pandas._libs.tslibs.timestamps.Timestamp]] = None*, *end_time: Optional[Union[str, pandas._libs.tslibs.timestamps.Timestamp]] = None*, *with_indexes: bool = False*, *update_stats: bool = False*, *engine: Optional[str] = None*, *engine_args: Optional[dict] = None*) → mlrun.feature_store.feature_vector.OfflineVectorResponse

retrieve offline feature vector results

specify a feature vector object/uri and retrieve the desired features, their metadata and statistics. returns *OfflineVectorResponse*, results can be returned as a dataframe or written to a target

The start_time and end_time attributes allow filtering the data to a given time range, they accept string values or pandas *Timestamp* objects, string values can also be relative, for example: "now", "now - 1d2h", "now+5m", where a valid pandas Timedelta string follows the verb "now", for time alignment you can use the verb "floor" e.g. "now -1d floor 1H" will align the time to the last hour (the floor string is passed to pandas.Timestamp.floor(), can use D, H, T, S for day, hour, min, sec alignment).

example:

```
features = [
    "stock-quotes.bid",
    "stock-quotes.asks_sum_5h",
    "stock-quotes.ask as mycol",
    "stocks.*",
]
vector = FeatureVector(features=features)
resp = get_offline_features(
    vector, entity_rows=trades, entity_timestamp_column="time"
)
print(resp.to_dataframe())
print(vector.get_stats_table())
resp.to_parquet("./out.parquet")
```

Parameters

- **feature_vector** – feature vector uri or FeatureVector object. passing feature vector obj requires update permissions
- **entity_rows** – dataframe with entity rows to join with
- **target** – where to write the results to
- **drop_columns** – list of columns to drop from the final result
- **entity_timestamp_column** – timestamp column name in the entity rows dataframe
- **run_config** – function and/or run configuration see *RunConfig*
- **start_time** – datetime, low limit of time needed to be filtered. Optional. entity_timestamp_column must be passed when using time filtering.

- **end_time** – datetime, high limit of time needed to be filtered. Optional. entity_timestamp_column must be passed when using time filtering.
- **with_indexes** – return vector with index columns and timestamp_key from the feature sets (default False)
- **update_stats** – update features statistics from the requested feature sets on the vector. Default is False.
- **engine** – processing engine kind ("local", "dask", or "spark")
- **engine_args** – kwargs for the processing engine

mlrun.feature_store.**get_online_feature_service**(*feature_vector: Union[str, mlrun.feature_store.feature_vector.FeatureVector], run_config: Optional[mlrun.feature_store.common.RunConfig] = None, fixed_window_type: mlrun.feature_store.feature_vector.FixedWindowType = <FixedWindowType.LastClosedWindow: 2>, impute_policy: Optional[dict] = None, update_stats: bool = False*) → mlrun.feature_store.feature_vector.OnlineVectorService

initialize and return online feature vector service api, returns *OnlineVectorService*

There are two ways to use the function 1. As context manager

example:

```
    with get_online_feature_service(vector_uri) as svc:
        resp = svc.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
        print(resp)
        resp = svc.get([{"ticker": "AAPL"}], as_list=True)
        print(resp)

example with imputing::

    with get_online_feature_service(vector_uri, impute_policy={"*": "$mean
↪", "amount": 0)) as svc:
        resp = svc.get([{"id": "C123487"}])
```

2. **as simple function, note that in that option you need to close the session.** example:

```
svc = get_online_feature_service(vector_uri)
try:
    resp = svc.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
    print(resp)
    resp = svc.get([{"ticker": "AAPL"}], as_list=True)
    print(resp)

finally:
    svc.close()
```

example with imputing:

```
svc = get_online_feature_service(vector_uri, impute_policy={"*": "$mean",
↪"amount": 0))
try:
    resp = svc.get([{"id": "C123487"}])
except Exception as e:
    handling exception...
finally:
    svc.close()
```

Parameters

- **feature_vector** – feature vector uri or FeatureVector object. passing feature vector obj requires update permissions
- **run_config** – function and/or run configuration for remote jobs/services
- **impute_policy** – a dict with *impute_policy* per feature, the dict key is the feature name and the dict value indicate which value will be used in case the feature is NaN/empty, the replaced value can be fixed number for constants or $mean, $max, $min, $std, $count for statistical values. "*" is used to specify the default for all features, example: *{"*": "$mean"}*
- **fixed_window_type** – determines how to query the fixed window values which were previously inserted by ingest
- **update_stats** – update features statistics from the requested feature sets on the vector. Default is False.

mlrun.feature_store.**ingest**(*featureset: Optional[Union[mlrun.feature_store.feature_set.FeatureSet, str]] = None*, *source=None*, *targets: Optional[List[mlrun.model.DataTargetBase]] = None*, *namespace=None*, *return_df: bool = True*, *infer_options: mlrun.data_types.data_types.InferOptions = 63*, *run_config: Optional[mlrun.feature_store.common.RunConfig] = None*, *mlrun_context=None*, *spark_context=None*, *overwrite=None*) → pandas.core.frame.DataFrame

Read local DataFrame, file, URL, or source into the feature store Ingest reads from the source, run the graph transformations, infers metadata and stats and writes the results to the default of specified targets

when targets are not specified data is stored in the configured default targets (will usually be NoSQL for real-time and Parquet for offline).

the *run_config* parameter allow specifying the function and job configuration, see: *RunConfig*

example:

```
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
stocks = pd.read_csv("stocks.csv")
df = ingest(stocks_set, stocks, infer_options=fstore.InferOptions.default())

# for running as remote job
config = RunConfig(image='mlrun/mlrun')
df = ingest(stocks_set, stocks, run_config=config)

# specify source and targets
source = CSVSource("mycsv", path="measurements.csv")
targets = [CSVTarget("mycsv", path="./mycsv.csv")]
ingest(measurements, source, targets)
```

Parameters

- **featureset** – feature set object or featureset.uri. (uri must be of a feature set that is in the DB, call *.save()* if it's not)
- **source** – source dataframe or other sources (e.g. parquet source see: *ParquetSource* and other classes in mlrun.datastore with suffix Source)
- **targets** – optional list of data target objects
- **namespace** – namespace or module containing graph classes
- **return_df** – indicate if to return a dataframe with the graph results
- **infer_options** – schema and stats infer options
- **run_config** – function and/or run configuration for remote jobs, see *RunConfig*
- **mlrun_context** – mlrun context (when running as a job), for internal use !
- **spark_context** – local spark session for spark ingestion, example for creating the spark context: *spark = SparkSession.builder.appName("Spark function").getOrCreate()* For remote spark ingestion, this should contain the remote spark service name
- **overwrite** – delete the targets' data prior to ingestion (default: True for non scheduled ingest - deletes the targets that are about to be ingested.

  False for scheduled ingest - does not delete the target)

mlrun.feature_store.**preview**(*featureset: mlrun.feature_store.feature_set.FeatureSet*, *source*, *entity_columns: Optional[list] = None*, *timestamp_key: Optional[str] = None*, *namespace=None*, *options: Optional[mlrun.data_types.data_types.InferOptions] = None*, *verbose: bool = False*, *sample_size: Optional[int] = None*) → pandas.core.frame.DataFrame

run the ingestion pipeline with local DataFrame/file data and infer features schema and stats

example:

```
quotes_set = FeatureSet("stock-quotes", entities=[Entity("ticker")])
quotes_set.add_aggregation("ask", ["sum", "max"], ["1h", "5h"], "10m")
quotes_set.add_aggregation("bid", ["min", "max"], ["1h"], "10m")
df = preview(
    quotes_set,
    quotes_df,
    entity_columns=["ticker"],
    timestamp_key="time",
)
```

Parameters

- **featureset** – feature set object or uri
- **source** – source dataframe or csv/parquet file path
- **entity_columns** – list of entity (index) column names
- **timestamp_key** – timestamp column name
- **namespace** – namespace or module containing graph classes
- **options** – schema and stats infer options (`InferOptions`)
- **verbose** – verbose log

- **sample_size** – num of rows to sample from the dataset (for large datasets)

class mlrun.feature_store.steps.**DateExtractor**(*parts: Union[Dict[str, str], List[str]]*, *timestamp_col: Optional[str] = None*, ***kwargs*)

Date Extractor allows you to extract a date-time component

Date Extractor extract a date-time component into new columns

The extracted date part will appear as <*timestamp_col*>_<*date_part*> feature.

Supports part values:

- asm8: Return numpy datetime64 format in nanoseconds.
- day_of_week: Return day of the week.
- day_of_year: Return the day of the year.
- dayofweek: Return day of the week.
- dayofyear: Return the day of the year.
- days_in_month: Return the number of days in the month.
- daysinmonth: Return the number of days in the month.
- freqstr: Return the total number of days in the month.
- is_leap_year: Return True if year is a leap year.
- is_month_end: Return True if date is last day of month.
- is_month_start: Return True if date is first day of month.
- is_quarter_end: Return True if date is last day of the quarter.
- is_quarter_start: Return True if date is first day of the quarter.
- is_year_end: Return True if date is last day of the year.
- is_year_start: Return True if date is first day of the year.
- quarter: Return the quarter of the year.
- tz: Alias for tzinfo.
- week: Return the week number of the year.
- weekofyear: Return the week number of the year.

example:

```
# (taken from the fraud-detection end-to-end feature store demo)
# Define the Transactions FeatureSet
transaction_set = fs.FeatureSet("transactions",
                                entities=[fs.Entity("source")],
                                timestamp_key='timestamp',
                                description="transactions feature set")

# Get FeatureSet computation graph
transaction_graph = transaction_set.graph

# Add the custom `DateExtractor` step
# to the computation graph
transaction_graph.to(
```

(continues on next page)

(continued from previous page)

```
        class_name='DateExtractor',
        name='Extract Dates',
        parts = ['hour', 'day_of_week'],
        timestamp_col = 'timestamp',
    )
```

Parameters

- **parts** – list of pandas style date-time parts you want to extract.
- **timestamp_col** – The name of the column containing the timestamps to extract from, by default "timestamp"

__init__(*parts: Union[Dict[str, str], List[str]]*, *timestamp_col: Optional[str] = None*, ***kwargs*)

Date Extractor extract a date-time component into new columns

The extracted date part will appear as <*timestamp_col*>_<*date_part*> feature.

Supports part values:

- asm8: Return numpy datetime64 format in nanoseconds.
- day_of_week: Return day of the week.
- day_of_year: Return the day of the year.
- dayofweek: Return day of the week.
- dayofyear: Return the day of the year.
- days_in_month: Return the number of days in the month.
- daysinmonth: Return the number of days in the month.
- freqstr: Return the total number of days in the month.
- is_leap_year: Return True if year is a leap year.
- is_month_end: Return True if date is last day of month.
- is_month_start: Return True if date is first day of month.
- is_quarter_end: Return True if date is last day of the quarter.
- is_quarter_start: Return True if date is first day of the quarter.
- is_year_end: Return True if date is last day of the year.
- is_year_start: Return True if date is first day of the year.
- quarter: Return the quarter of the year.
- tz: Alias for tzinfo.
- week: Return the week number of the year.
- weekofyear: Return the week number of the year.

example:

```
# (taken from the fraud-detection end-to-end feature store demo)
# Define the Transactions FeatureSet
transaction_set = fs.FeatureSet("transactions",
                                entities=[fs.Entity("source")],
```

(continues on next page)

(continued from previous page)

```
                                    timestamp_key='timestamp',
                                    description="transactions feature set")

# Get FeatureSet computation graph
transaction_graph = transaction_set.graph

# Add the custom `DateExtractor` step
# to the computation graph
transaction_graph.to(
        class_name='DateExtractor',
        name='Extract Dates',
        parts = ['hour', 'day_of_week'],
        timestamp_col = 'timestamp',
    )
```

Parameters

- **parts** – list of pandas style date-time parts you want to extract.
- **timestamp_col** – The name of the column containing the timestamps to extract from, by default "timestamp"

class mlrun.feature_store.steps.**FeaturesetValidator**(*featureset=None*, *columns=None*, *name=None*, ***kwargs*)

Validate feature values according to the feature set validation policy

Validate feature values according to the feature set validation policy

Parameters

- **featureset** – feature set uri (or "." for current feature set pipeline)
- **columns** – names of the columns/fields to validate
- **name** – step name
- **kwargs** – optional kwargs (for storey)

__init__(*featureset=None*, *columns=None*, *name=None*, ***kwargs*)

Validate feature values according to the feature set validation policy

Parameters

- **featureset** – feature set uri (or "." for current feature set pipeline)
- **columns** – names of the columns/fields to validate
- **name** – step name
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**Imputer**(*method: str = 'avg'*, *default_value=None*, *mapping: Optional[Dict[str, Dict[str, Any]]] = None*, ***kwargs*)

Replace None values with default values

Parameters

- **method** – for future use
- **default_value** – default value if not specified per column

- **mapping** – a dict of per column default value
- **kwargs** – optional kwargs (for storey)

__init__(*method: str = 'avg'*, *default_value=None*, *mapping: Optional[Dict[str, Dict[str, Any]]] = None*, ***kwargs*)

Replace None values with default values

Parameters

- **method** – for future use
- **default_value** – default value if not specified per column
- **mapping** – a dict of per column default value
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**MapValues**(*mapping: Dict[str, Dict[str, Any]]*, *with_original_features: bool = False*, *suffix: str = 'mapped'*, ***kwargs*)

Map column values to new values

Map column values to new values

example:

```
# replace the value "U" with '0' in the age column
graph.to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))

# replace integers, example
graph.to(MapValues(mapping={'not': {0: 1, 1: 0}}))

# replace by range, use -inf and inf for extended range
graph.to(MapValues(mapping={'numbers': {'ranges': {'negative': [-inf, 0],
↪'positive': [0, inf]}}}))
```

Parameters

- **mapping** – a dict with entry per column and the associated old/new values map
- **with_original_features** – set to True to keep the original features
- **suffix** – the suffix added to the column name <column>_<suffix> (default is “mapped”)
- **kwargs** – optional kwargs (for storey)

__init__(*mapping: Dict[str, Dict[str, Any]]*, *with_original_features: bool = False*, *suffix: str = 'mapped'*, ***kwargs*)

Map column values to new values

example:

```
# replace the value "U" with '0' in the age column
graph.to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))

# replace integers, example
graph.to(MapValues(mapping={'not': {0: 1, 1: 0}}))

# replace by range, use -inf and inf for extended range
graph.to(MapValues(mapping={'numbers': {'ranges': {'negative': [-inf, 0],
↪'positive': [0, inf]}}}))
```

Parameters

- **mapping** – a dict with entry per column and the associated old/new values map
- **with_original_features** – set to True to keep the original features
- **suffix** – the suffix added to the column name <column>_<suffix> (default is "mapped")
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**OneHotEncoder**(*mapping: Dict[str, Dict[str, Any]]*, ***kwargs*)

Create new binary fields, one per category (one hot encoded)

example:

```
mapping = {'category': ['food', 'health', 'transportation'],
           'gender': ['male', 'female']}
graph.to(OneHotEncoder(mapping=one_hot_encoder_mapping))
```

Parameters

- **mapping** – a dict of per column categories (to map to binary fields)
- **kwargs** – optional kwargs (for storey)

__init__(*mapping: Dict[str, Dict[str, Any]]*, ***kwargs*)

Create new binary fields, one per category (one hot encoded)

example:

```
mapping = {'category': ['food', 'health', 'transportation'],
           'gender': ['male', 'female']}
graph.to(OneHotEncoder(mapping=one_hot_encoder_mapping))
```

Parameters

- **mapping** – a dict of per column categories (to map to binary fields)
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**SetEventMetadata**(*id_path: Optional[str] = None*, *key_path: Optional[str] = None*, *time_path: Optional[str] = None*, *random_id: Optional[bool] = None*, ***kwargs*)

Set the event metadata (id, key, timestamp) from the event body

Set the event metadata (id, key, timestamp) from the event body

set the event metadata fields (id, key, and time) from the event body data structure the xx_path attribute defines the key or path to the value in the body dict, "." in the path string indicate the value is in a nested dict e.g. *"x.y"* means *{"x": {"y": value}}*

example:

```
flow = function.set_topology("flow")
# build a graph and use the SetEventMetadata step to extract the id, key and path␣
↪from the event body
# ("myid", "mykey" and "mytime" fields), the metadata will be used for following␣
↪data processing steps
```

(continues on next page)

(continued from previous page)

```
# (e.g. feature store ops, time/key aggregations, write to databases/streams, etc.
↪)
flow.to(SetEventMetadata(id_path="myid", key_path="mykey", time_path="mytime"))
    .to(...)  # additional steps

server = function.to_mock_server()
event = {"myid": "34", "mykey": "123", "mytime": "2022-01-18 15:01"}
resp = server.test(body=event)
```

Parameters

- **id_path** – path to the id value
- **key_path** – path to the key value
- **time_path** – path to the time value (value should be of type str or datetime)
- **random_id** – if True will set the event.id to a random value

__init__(*id_path: Optional[str] = None*, *key_path: Optional[str] = None*, *time_path: Optional[str] = None*, *random_id: Optional[bool] = None*, ***kwargs*)

Set the event metadata (id, key, timestamp) from the event body

set the event metadata fields (id, key, and time) from the event body data structure the xx_path attribute defines the key or path to the value in the body dict, "." in the path string indicate the value is in a nested dict e.g. *"x.y"* means *{"x": {"y": value}}*

example:

```
flow = function.set_topology("flow")
# build a graph and use the SetEventMetadata step to extract the id, key and␣
↪path from the event body
# ("myid", "mykey" and "mytime" fields), the metadata will be used for␣
↪following data processing steps
# (e.g. feature store ops, time/key aggregations, write to databases/streams,␣
↪etc.)
flow.to(SetEventMetadata(id_path="myid", key_path="mykey", time_path="mytime
↪"))
    .to(...)  # additional steps

server = function.to_mock_server()
event = {"myid": "34", "mykey": "123", "mytime": "2022-01-18 15:01"}
resp = server.test(body=event)
```

Parameters

- **id_path** – path to the id value
- **key_path** – path to the key value
- **time_path** – path to the time value (value should be of type str or datetime)
- **random_id** – if True will set the event.id to a random value

mlrun.model

class mlrun.model.**DataSource**(*name: Optional[str] = None*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time: Optional[Union[datetime.datetime, str]] = None*, *end_time: Optional[Union[datetime.datetime, str]] = None*)

Bases: mlrun.model.ModelObj

online or offline data source spec

class mlrun.model.**DataTarget**(*kind: Optional[str] = None*, *name: str = ''*, *path=None*, *online=None*)

Bases: *mlrun.model.DataTargetBase*

data target with extra status information (used in the feature-set/vector status)

class mlrun.model.**DataTargetBase**(*kind: Optional[str] = None*, *name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *after_state=None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data target spec, specify a destination for the feature set data

classmethod from_dict(*struct=None*, *fields=None*)

create an object from a python dictionary

class mlrun.model.**FeatureSetProducer**(*kind=None*, *name=None*, *uri=None*, *owner=None*, *sources=None*)

Bases: mlrun.model.ModelObj

information about the task/job which produced the feature set data

class mlrun.model.**HyperParamOptions**(*param_file=None*, *strategy=None*, *selector: Optional[mlrun.model.HyperParamStrategies] = None*, *stop_condition=None*, *parallel_runs=None*, *dask_cluster_uri=None*, *max_iterations=None*, *max_errors=None*, *teardown_dask=None*)

Bases: mlrun.model.ModelObj

Hyper Parameter Options

Parameters

- **param_file** (*str*) – hyper params input file path/url, instead of inline
- **strategy** (*str*) – hyper param strategy - grid, list or random
- **selector** (*str*) – selection criteria for best result ([min|max.]<result>), e.g. max.accuracy
- **stop_condition** (*str*) – early stop condition e.g. "accuracy > 0.9"
- **parallel_runs** (*int*) – number of param combinations to run in parallel (over Dask)
- **dask_cluster_uri** (*str*) – db uri for a deployed dask cluster function, e.g. db://myproject/dask
- **max_iterations** (*int*) – max number of runs (in random strategy)

- **max_errors** (*int*) – max number of child runs errors for the overall job to fail
- **teardown_dask** (*bool*) – kill the dask cluster pods after the runs

mlrun.model.**NewTask**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)

Creates a new task - see new_task

class mlrun.model.**RunMetadata**(*uid=None*, *name=None*, *project=None*, *labels=None*, *annotations=None*, *iteration=None*)

Bases: mlrun.model.ModelObj

Run metadata

class mlrun.model.**RunObject**(*spec: Optional[*mlrun.model.RunSpec*] = None*, *metadata: Optional[*mlrun.model.RunMetadata*] = None*, *status: Optional[*mlrun.model.RunStatus*] = None*)

Bases: *mlrun.model.RunTemplate*

A run

artifact(*key*) → mlrun.datastore.base.DataItem
return artifact DataItem by key

logs(*watch=True*, *db=None*)
return or watch on the run logs

output(*key*)
return the value of a specific result or artifact by key

property outputs
return a dict of outputs, result values and artifact uris

refresh()
refresh run state from the db

show()
show the current status widget, in jupyter notebook

state()
current run state

property ui_url
UI URL (for relevant runtimes)

uid()
run unique id

wait_for_completion(*sleep=3*, *timeout=0*, *raise_on_failure=True*)
wait for async run to complete

class mlrun.model.**RunSpec**(*parameters=None*, *hyperparams=None*, *param_file=None*, *selector=None*, *handler=None*, *inputs=None*, *outputs=None*, *input_path=None*, *output_path=None*, *function=None*, *secret_sources=None*, *data_stores=None*, *strategy=None*, *verbose=None*, *scrape_metrics=None*, *hyper_param_options=None*, *allow_empty_resources=None*)

Bases: mlrun.model.ModelObj

Run specification

to_dict (*fields=None*, *exclude=None*)
convert the object to a python dictionary

class mlrun.model.**RunStatus**(*state=None*, *error=None*, *host=None*, *commit=None*, *status_text=None*, *results=None*, *artifacts=None*, *start_time=None*, *last_update=None*, *iterations=None*, *ui_url=None*)
Bases: mlrun.model.ModelObj

Run status

class mlrun.model.**RunTemplate**(*spec: Optional[mlrun.model.RunSpec] = None*, *metadata: Optional[mlrun.model.RunMetadata] = None*)
Bases: mlrun.model.ModelObj

Run template

set_label(*key*, *value*)
set a key/value label for the task

with_hyper_params(*hyperparams*, *selector=None*, *strategy: Optional[mlrun.model.HyperParamStrategies] = None*, ***options*)
set hyper param values and configurations, see parameters in: *HyperParamOptions*

example:

```
grid_params = {"p1": [2,4,1], "p2": [10,20]}
task = mlrun.new_task("grid-search")
task.with_hyper_params(grid_params, selector="max.accuracy")
```

with_input(*key*, *path*)
set task data input, path is an Mlrun global DataItem uri

examples:

```
task.with_input("data", "/file-dir/path/to/file")
task.with_input("data", "s3://<bucket>/path/to/file")
task.with_input("data", "v3io://[<remote-host>]/<data-container>/path/to/file
↪")
```

with_param_file(*param_file*, *selector=None*, *strategy: Optional[mlrun.model.HyperParamStrategies] = None*, ***options*)
set hyper param values (from a file url) and configurations, see parameters in: *HyperParamOptions*

example:

```
grid_params = "s3://<my-bucket>/path/to/params.json"
task = mlrun.new_task("grid-search")
task.with_param_file(grid_params, selector="max.accuracy")
```

with_params(***kwargs*)
set task parameters using key=value, key2=value2, ..

with_secrets(*kind*, *source*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, example:

```
task.with_secrets('file', 'file.txt')
task.with_secrets('inline', {'key': 'val'})
task.with_secrets('env', 'ENV1,ENV2')
```

(continues on next page)

(continued from previous page)

```
task.with_secrets('vault', ['secret1', 'secret2'...])

# If using with k8s secrets, the k8s secret is managed by MLRun, through the␣
↪project-secrets
# mechanism. The secrets will be attached to the running pod as environment␣
↪variables.
task.with_secrets('kubernetes', ['secret1', 'secret2'])

# If using an empty secrets list [] then all accessible secrets will be␣
↪available.
task.with_secrets('vault', [])

# To use with Azure key vault, a k8s secret must be created with the␣
↪following keys:
# kubectl -n <namespace> create secret generic azure-key-vault-secret \
#     --from-literal=tenant_id=<service principal tenant ID> \
#     --from-literal=client_id=<service principal client ID> \
#     --from-literal=secret=<service principal secret key>

task.with_secrets('azure_vault', {
    'name': 'my-vault-name',
    'k8s_secret': 'azure-key-vault-secret',
    # An empty secrets list may be passed ('secrets': []) to access all vault␣
↪secrets.
    'secrets': ['secret1', 'secret2'...]
})
```

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns The RunTemplate object

class mlrun.model.**TargetPathObject**(*base_path=None*, *run_id=None*, *is_single_file=False*)

Bases: object

Class configuring the target path This class will take consideration of a few parameters to create the correct end result path: * run_id - if run_id is provided target will be considered as run_id mode

which require to contain a {run_id} place holder in the path.

- **is_single_file - if true then run_id must be the directory containing the output file** or generated before the file name (run_id/output.file).
- **base_path - if contains the place holder for run_id, run_id must not be None.** if run_id passed and place holder doesn't exist the place holder will be generated in the correct place.

mlrun.model.**new_task**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *hyper_param_options=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*) → *mlrun.model.RunTemplate*

Creates a new task

Parameters

- **name** – task name
- **project** – task project
- **handler** – code entry-point/handler name
- **params** – input parameters (dict)
- **hyper_params** – dictionary of hyper parameters and list values, each hyper param holds a list of values, the run will be executed for every parameter combination (GridSearch)
- **param_file** – a csv file with parameter combinations, first row hold the parameter names, following rows hold param values
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper parameter options, see: *HyperParamOptions*
- **inputs** – dictionary of input objects + optional paths (if path is omitted the path will be the in_path/key)
- **outputs** – dictionary of input objects + optional paths (if path is omitted the path will be the out_path/key)
- **in_path** – default input path/url (prefix) for inputs
- **out_path** – default output path/url (prefix) for artifacts
- **artifact_path** – default artifact output path
- **secrets** – extra secrets specs, will be injected into the runtime e.g. ['file=<filename>', 'env=ENV_KEY1,ENV_KEY2']
- **base** – task instance to use as a base instead of a fresh new task instance

mlrun.platforms

mlrun.platforms.**VolumeMount**
alias of mlrun.platforms.iguazio.Mount

mlrun.platforms.**auto_mount**(*pvc_name=''*, *volume_mount_path=''*, *volume_name=None*)
choose the mount based on env variables and params

volume will be selected by the following order: - k8s PVC volume when both pvc_name and volume_mount_path are set - k8s PVC volume when env var is set: MLRUN_PVC_MOUNT=<pvc-name>:<mount-path> - k8s PVC volume if it's configured as the auto mount type - iguazio v3io volume when V3IO_ACCESS_KEY and V3IO_USERNAME env vars are set

mlrun.platforms.**mount_configmap**(*configmap_name*, *mount_path*, *volume_name='configmap'*, *items=None*)
Modifier function to mount kubernetes configmap as files(s)

Parameters

- **configmap_name** – k8s configmap name
- **mount_path** – path to mount inside the container
- **volume_name** – unique volume name
- **items** – If unspecified, each key-value pair in the Data field of the referenced Configmap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present.

mlrun.platforms.**mount_hostpath**(*host_path*, *mount_path*, *volume_name='hostpath'*)
Modifier function to mount kubernetes configmap as files(s)

Parameters

- **host_path** – host path
- **mount_path** – path to mount inside the container
- **volume_name** – unique volume name

mlrun.platforms.**mount_pvc**(*pvc_name=None*, *volume_name='pipeline'*, *volume_mount_path='/mnt/pipeline'*)
Modifier function to apply to a Container Op to simplify volume, volume mount addition and enable better reuse of volumes, volume claims across container ops.

Usage:

```
train = train_op(...)
train.apply(mount_pvc('claim-name', 'pipeline', '/mnt/pipeline'))
```

mlrun.platforms.**mount_v3io**(*name='v3io'*, *remote=''*, *mount_path=''*, *access_key=''*, *user=''*, *secret=None*, *volume_mounts=None*)
Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mount_path** – the volume mount path (deprecated, exists for backwards compatibility, prefer to use mounts instead)
- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used
- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.
- **volume_mounts** – list of VolumeMount. empty volume mounts & remote will default to mount /v3io & /User.

mlrun.platforms.**mount_v3io_extended**(*name='v3io'*, *remote=''*, *mounts=None*, *access_key=''*, *user=''*, *secret=None*)
Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mounts** – list of mount & volume sub paths (type Mount). empty mounts & remote mount /v3io & /User
- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used

- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.platforms.**mount_v3io_legacy**(*name='v3io'*, *remote='~/'*, *mount_path='/User'*, *access_key=''*, *user=''*, *secret=None*)

Modifier function to apply to a Container Op to volume mount a v3io path :param name: the volume name :param remote: the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/ :param mount_path: the volume mount path :param access_key: the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used :param user: the username used to auth against v3io. if not given V3IO_USERNAME env var will be used :param secret: k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.platforms.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*)

Pretty-print a Python object to a stream [default is sys.stdout].

mlrun.platforms.**sleep**(*seconds*)

Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

mlrun.platforms.**v3io_cred**(*api=''*, *user=''*, *access_key=''*)

Modifier function to copy local v3io env vars to container

Usage:

```
train = train_op(...)
train.apply(use_v3io_cred())
```

mlrun.platforms.**watch_stream**(*url*, *shard_ids: Optional[list] = None*, *seek_to: Optional[str] = None*, *interval=None*, *is_json=False*, ***kwargs*)

watch on a v3io stream and print data every interval

example:: watch_stream('v3io:///users/admin/mystream')

Parameters

- **url** – stream url
- **shard_ids** – range or list of shard IDs
- **seek_to** – where to start/seek ('EARLIEST', 'LATEST', 'TIME', 'SEQUENCE')

:param interval watch interval time in seconds, 0 to run once and return :param is_json: indicate the payload is json (will be deserialized)

mlrun.projects

class mlrun.projects.**MlrunProject**(*name=None*, *description=None*, *params=None*, *functions=None*, *workflows=None*, *artifacts=None*, *artifact_path=None*, *conda=None*, *metadata=None*, *spec=None*, *default_requirements: Optional[Union[str, List[str]]] = None*)

Bases: mlrun.model.ModelObj

property artifact_path

This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property artifacts
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

build_function(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *with_mlrun: Optional[bool] = None*, *skip_deployed: bool = False*, *image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*)
deploy ML function, build container with its dependencies

Parameters

- **function** – name of the function (in the project) or function object
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **image** – target image name/path
- **base_image** – base image name/path (commands and source code will be added to it)
- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret_name** – k8s secret for accessing the docker registry
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP

clear_context()
delete all files and clear the context dir

property context
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

create_remote(*url*, *name='origin'*)
create remote for the project git

Parameters

- **url** – remote git url
- **name** – name for the remote (default is 'origin')

create_vault_secrets(*secrets*)

deploy_function(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *dashboard: str = ''*, *models: Optional[list] = None*, *env: Optional[dict] = None*, *tag: Optional[str] = None*, *verbose: Optional[bool] = None*, *builder_env: Optional[dict] = None*)
deploy real-time (nuclio based) functions

Parameters

- **function** – name of the function (in the project) or function object
- **dashboard** – url of the remote Nuclio dashboard (when not local)
- **models** – list of model items
- **env** – dict of extra environment variables
- **tag** – extra version tag

:param verbose add verbose prints/logs :param builder_env: env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}

property description
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

export (*filepath=None*)
save the project object into a file (default to project.yaml)

func (*key*, *sync=False*) → mlrun.runtimes.base.BaseRuntime
get function object by name

Parameters **sync** – will reload/reinit the function

Returns function object

property functions
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

get_artifact_uri (*key: str*, *category: str = 'artifact'*, *tag: Optional[str] = None*) → str
return the project artifact uri (store://..) from the artifact key

example:

```
uri = project.get_artifact_uri("my_model", category="model", tag="prod")
```

Parameters

- **key** – artifact key/name
- **category** – artifact category (artifact, model, feature-vector, ..)
- **tag** – artifact version tag, default to latest version

get_function (*key*, *sync=False*, *enrich=False*, *ignore_cache=False*) → mlrun.runtimes.base.BaseRuntime
get function object by name

Parameters

- **key** – name of key for search
- **sync** – will reload/reinit the function
- **enrich** – add project info/config/source info to the function object
- **ignore_cache** – read the function object from the DB (ignore the local cache)

Returns function object

get_function_objects () → Dict[str, mlrun.runtimes.base.BaseRuntime]
"get a virtual dict with all the project functions ready for use in a pipeline

get_param (*key: str*, *default=None*)
get project param by key

get_run_status (*run*, *timeout=3600*, *expected_statuses=None*, *notifiers: Optional[mlrun.utils.helpers.RunNotifications] = None*)

get_secret (*key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through files, env, ..

get_store_resource(*uri*)
get store resource object by uri

get_vault_secrets(*secrets=None*, *local=False*)

kind = 'project'

list_artifacts(*name=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*, *kind: Optional[str] = None*, *category: Optional[Union[str, mlrun.api.schemas.artifact.ArtifactCategories]] = None*) → mlrun.lists.ArtifactList
List artifacts filtered by various parameters.

The returned result is an *ArtifactList* (list of dict), use *.to_objects()* to convert it to a list of RunObjects, *.show()* to view graphically in Jupyter, and *.to_df()* to convert to a DataFrame.

Examples:

```
# Get latest version of all artifacts in project
latest_artifacts = project.list_artifacts('', tag='latest')
# check different artifact versions for a specific artifact, return as␣
↪objects list
result_versions = project.list_artifacts('results', tag='*').to_objects()
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in `HTTPRunDB`.
- **until** – Not in use in `HTTPRunDB`.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.
- **kind** – Return artifacts of the requested kind.
- **category** – Return artifacts of the requested category.

list_functions(*name=None*, *tag=None*, *labels=None*)
Retrieve a list of functions, filtered by specific criteria.

example:

```
functions = project.list_functions(tag="latest")
```

Parameters

- **name** – Return only functions with a specific name.
- **tag** – Return function versions with specific tags.

- **labels** – Return functions that have specific labels assigned to them.

Returns List of function objects.

list_models(*name=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*)

List models in project, filtered by various parameters.

Examples:

```
# Get latest version of all models in project
latest_models = project.list_models('', tag='latest')
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in `HTTPRunDB`.
- **until** – Not in use in `HTTPRunDB`.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.

list_runs(*name=None*, *uid=None*, *labels=None*, *state=None*, *sort=True*, *last=0*, *iter=False*, *start_time_from: Optional[datetime.datetime] = None*, *start_time_to: Optional[datetime.datetime] = None*, *last_update_time_from: Optional[datetime.datetime] = None*, *last_update_time_to: Optional[datetime.datetime] = None*, ***kwargs*) → mlrun.lists.RunList

Retrieve a list of runs, filtered by various options.

The returned result is a `` (list of dict), use *.to_objects()* to convert it to a list of RunObjects, *.show()* to view graphically in Jupyter, *.to_df()* to convert to a DataFrame, and *compare()* to generate comparison table and PCP plot.

Example:

```
# return a list of runs matching the name and label and compare
runs = project.list_runs(name='download', labels='owner=admin')
runs.compare()
# If running in Jupyter, can use the .show() function to display the results
project.list_runs(name='').show()
```

Parameters

- **name** – Name of the run to retrieve.
- **uid** – Unique ID of the run.
- **project** – Project that the runs belongs to.

- **labels** – List runs that have a specific label assigned. Currently only a single label filter can be applied, otherwise result will be empty.
- **state** – List only runs whose state is specified.
- **sort** – Whether to sort the result according to their start time. Otherwise results will be returned by their internal order in the DB (order will not be guaranteed).
- **last** – Deprecated - currently not used.
- **iter** – If `True` return runs from all iterations. Otherwise, return only runs whose `iter` is 0.
- **start_time_from** – Filter by run start time in `[start_time_from, start_time_to]`.
- **start_time_to** – Filter by run start time in `[start_time_from, start_time_to]`.
- **last_update_time_from** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.
- **last_update_time_to** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.

log_artifact(*item*, *body=None*, *tag=''*, *local_path=''*, *artifact_path=None*, *format=None*, *upload=None*, *labels=None*, *target_path=None*)

log an output artifact and optionally upload it to datastore

example:

```
project.log_artifact(
    "some-data",
    body=b"abc is 123",
    local_path="model.txt",
    labels={"framework": "xgboost"},
)
```

Parameters

- **item** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **format** – artifact file format: csv, png, ..
- **tag** – version tag
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with

Returns artifact object

log_dataset(*key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=None*, *labels=None*, *format=''*, *preview=None*, *stats=False*, *target_path=''*, *extra_data=None*, ***kwargs*) → mlrun.artifacts.dataset.DatasetArtifact

log a dataset artifact and optionally upload it to datastore

example:

```
raw_data = {
    "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
    "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
    "age": [42, 52, 36, 24, 73],
    "testScore": [25, 94, 57, 62, 70],
}
df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age",
↪"testScore"])
project.log_dataset("mydf", df=df, stats=True)
```

Parameters

- **key** – artifact key
- **df** – dataframe object
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with

Returns artifact object

log_model(*key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *algorithm=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=None*, *labels=None*, *inputs: Optional[List[mlrun.features.Feature]] = None*, *outputs: Optional[List[mlrun.features.Feature]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *training_set=None*, *label_column=None*, *extra_data=None*, ***kwargs*)

log a model artifact and optionally upload it to datastore

example:

```
project.log_model("model", body=dumps(model),
                  model_file="model.pkl",
                  metrics=context.results,
                  training_set=training_df,
                  label_column='label',
```

(continues on next page)

(continued from previous page)

```
                    feature_vector=feature_vector_uri,
                    labels={"app": "fraud"})
```

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **model_file** – path to the local model file we upload (see also model_dir)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **framework** – name of the ML framework
- **algorithm** – training algorithm name
- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **training_set** – training set dataframe, used to infer inputs & outputs
- **label_column** – which columns in the training set are the label (target) columns
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be abs/relative path string | bytes | artifact object

Returns artifact object

property metadata

property mountdir
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property name
Project name, this is a property of the project metadata

property notifiers

property params
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

pull(*branch=None*, *remote=None*)
pull/update sources from git or tar into the context dir

Parameters

- **branch** – git branch, if not the current one
- **remote** – git remote, if other than origin

push(*branch*, *message=None*, *update=True*, *remote=None*, *add: Optional[list] = None*)
update spec and push updates to remote git repo

Parameters

- **branch** – target git branch
- **message** – git commit message
- **update** – update files (git add update=True)
- **remote** – git remote, default to origin
- **add** – list of files to add

register_artifacts()
register the artifacts in the MLRun DB (under this project)

reload(*sync=False*, *context=None*) → mlrun.projects.project.MlrunProject
reload the project and function objects from the project yaml/specs

Parameters

- **sync** – set to True to load functions objects
- **context** – context directory (where the yaml and code exist)

Returns project object

remove_function(*name*)
remove a function from a project

Parameters **name** – name of the function (under the project)

run(*name=None*, *workflow_path=None*, *arguments=None*, *artifact_path=None*, *workflow_handler=None*, *namespace=None*, *sync=False*, *watch=False*, *dirty=False*, *ttl=None*, *engine=None*, *local=False*) → mlrun.projects.pipelines._PipelineRunStatus
run a workflow using kubeflow pipelines

Parameters

- **name** – name of the workflow
- **workflow_path** – url to a workflow file, if not a project workflow
- **arguments** – kubeflow pipelines arguments (parameters)
- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id
- **workflow_handler** – workflow function handler (for running workflow function directly)
- **namespace** – kubernetes namespace if other than default
- **sync** – force functions sync before run
- **watch** – wait for pipeline completion
- **dirty** – allow running the workflow when the git repo is dirty
- **ttl** – pipeline ttl in secs (after that the pods will be removed)

- **engine** – workflow engine running the workflow. supported values are 'kfp' (default) or 'local'
- **local** – run local pipeline with local functions (set local=True in function.run())

Returns run id

run_function(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *handler: Optional[str] = None*, *name: str = ''*, *params: Optional[dict] = None*, *hyperparams: Optional[dict] = None*, *hyper_param_options: Optional[*mlrun.model.HyperParamOptions*] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[List[str]] = None*, *workdir: str = ''*, *labels: Optional[dict] = None*, *base_task: Optional[*mlrun.model.RunTemplate*] = None*, *watch: bool = True*, *local: bool = False*, *verbose: Optional[bool] = None*, *selector: Optional[str] = None*, *auto_build: Optional[bool] = None*) → Union[*mlrun.model.RunObject*, kfp.dsl._container_op.ContainerOp]

Run a local or remote task as part of a local/kubeflow pipeline

example (use with project):

```
# create a project with two functions (local and from marketplace)
project = mlrun.new_project(project_name, "./proj")
project.set_function("mycode.py", "myfunc", image="mlrun/mlrun")
project.set_function("hub://sklearn_classifier", "train")

# run functions (refer to them by name)
run1 = project.run_function("myfunc", params={"x": 7})
run2 = project.run_function("train", params={"data": run1.outputs["data"]})
```

Parameters

- **function** – name of the function (in the project) or function object
- **handler** – name of the function handler
- **name** – execution name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **workdir** – default input artifacts path
- **labels** – labels to tag the job/run with ({key:val, ..})
- **base_task** – task object to use as base
- **watch** – watch/follow run log, True by default
- **local** – run the function locally vs on the runtime/cluster
- **verbose** – add verbose prints/logs
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs

Returns MLRun RunObject or KubeFlow containerOp

save (*filepath=None*)

save_to_db ()

save_workflow (*name*, *target*, *artifact_path=None*, *ttl=None*)
create and save a workflow as a yaml or archive file

Parameters

- **name** – workflow name
- **target** – target file path (can end with .yaml or .zip)
- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id
- **ttl** – pipeline ttl (time to live) in secs (after that the pods will be removed)

set_artifact (*key*, *artifact=None*, *target_path=None*, *tag=None*)
add/set an artifact in the project spec (will be registered on load)

example:

```
# register a simple file artifact
project.set_artifact('data', target_path=data_url)
# register a model artifact
project.set_artifact('model', ModelArtifact(model_file="model.pkl"), target_
↪path=model_dir_url)
```

Parameters

- **key** – artifact key/name
- **artifact** – mlrun Artifact object (or its subclasses)
- **target_path** – absolute target path url (point to the artifact content location)
- **tag** – artifact tag

set_function (*func: Optional[Union[str, mlrun.runtimes.base.BaseRuntime]] = None*, *name: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *handler=None*, *with_repo: Optional[bool] = None*, *requirements: Optional[Union[str, List[str]]] = None*) → mlrun.runtimes.base.BaseRuntime
update or add a function object to the project

function can be provided as an object (func) or a .py/.ipynb/.yaml url support url prefixes:

```
object (s3://, v3io://, ..)
MLRun DB e.g. db://project/func:ver
functions hub/market: e.g. hub://sklearn_classifier:master
```

examples:

```
proj.set_function(func_object)
proj.set_function('./src/mycode.py', 'ingest',
                  image='myrepo/ing:latest', with_repo=True)
proj.set_function('http://.../mynb.ipynb', 'train')
proj.set_function('./func.yaml')
proj.set_function('hub://get_toy_data', 'getdata')
```

Parameters

- **func** – function object or spec/code url, None refers to current Notebook

- **name** – name of the function (under the project)
- **kind** – runtime kind e.g. job, nuclio, spark, dask, mpijob default: job
- **image** – docker image to be used, can also be specified in the function object/yaml
- **handler** – default function handler to invoke (can only be set with .py/.ipynb files)
- **with_repo** – add (clone) the current repo to the build source
- **requirements** – list of python packages or pip requirements file path

Returns project object

set_model_monitoring_credentials(*access_key: str*)
Set the credentials that will be used by the project's model monitoring infrastructure functions. The supplied credentials must have data access

Parameters **access_key** – Model Monitoring access key for managing user permissions.

set_secrets(*secrets: Optional[dict] = None*, *file_path: Optional[str] = None*, *provider: Optional[Union[str, mlrun.api.schemas.secret.SecretProviderName]] = None*)
set project secrets from dict or secrets env file when using a secrets file it should have lines in the form KEY=VALUE, comment line start with "#" V3IO paths/credentials and MLrun service API address are dropped from the secrets

example secrets file:

```
# this is an env file
AWS_ACCESS_KEY_ID-XXXX
AWS_SECRET_ACCESS_KEY=YYYY
```

usage:

```
# read env vars from dict or file and set as project secrets
project.set_secrets({"SECRET1": "value"})
project.set_secrets(file_path="secrets.env")
```

Parameters

- **secrets** – dict with secrets key/value
- **file_path** – path to secrets file
- **provider** – MLRun secrets provider

set_source(*source*, *pull_at_runtime=False*)
set the project source code path(can be git/tar/zip archive)

Parameters

- **source** – valid path to git, zip, or tar file, (or None for current) e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **pull_at_runtime** – load the archive into the container at job runtime vs on build/deploy

set_workflow(*name*, *workflow_path: str*, *embed=False*, *engine=None*, *args_schema: Optional[List[mlrun.model.EntrypointParam]] = None*, *handler=None*, ***args*)
add or update a workflow, specify a name and the code path

Parameters

- **name** – name of the workflow

- **workflow_path** – url/path for the workflow file
- **embed** – add the workflow code into the project.yaml
- **engine** – workflow processing engine ("kfp" or "local")
- **args_schema** – list of arg schema definitions (:py:class`~mlrun.model.EntrypointParam`)
- **handler** – workflow function handler
- **args** – argument values (key=value, ..)

property source
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property spec

property status

sync_functions(*names: Optional[list] = None*, *always=True*, *save=False*)
reload function objects from specs and files

with_secrets(*kind*, *source*, *prefix=''*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows,example:

```
proj.with_secrets('file', 'file.txt')
proj.with_secrets('inline', {'key': 'val'})
proj.with_secrets('env', 'ENV1,ENV2', prefix='PFX_')
```

Vault secret source has several options:

```
proj.with_secrets('vault', {'user': <user name>, 'secrets': ['secret1',
↪'secret2' ...]})
proj.with_secrets('vault', {'project': <proj.name>, 'secrets': ['secret1',
↪'secret2' ...]})
proj.with_secrets('vault', ['secret1', 'secret2' ...])
```

The 2nd option uses the current project name as context. Can also use empty secret list:

```
proj.with_secrets('vault', [])
```

This will enable access to all secrets in vault registered to the current project.

Parameters

- **kind** – secret type (file, inline, env, vault)
- **source** – secret data or link (see example)
- **prefix** – add a prefix to the keys in this source

Returns project object

property workflows
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

class mlrun.projects.**ProjectMetadata**(*name=None*, *created=None*, *labels=None*, *annotations=None*)
Bases: mlrun.model.ModelObj

property name
Project name

static validate_project_name(*name: str*, *raise_on_failure: bool = True*) → bool

class mlrun.projects.**ProjectSpec**(*description=None*, *params=None*, *functions=None*, *workflows=None*, *artifacts=None*, *artifact_path=None*, *conda=None*, *source=None*, *subpath=None*, *origin_url=None*, *goals=None*, *load_source_on_run=None*, *default_requirements: Optional[Union[str, List[str]]] = None*, *desired_state='online'*, *owner=None*, *disable_auto_mount=False*)
Bases: mlrun.model.ModelObj

property artifacts
list of artifacts used in this project

property functions
list of function object/specs used in this project

property mountdir
specify to mount the context dir inside the function container use '.' to use the same path as in the client e.g. Jupyter

remove_artifact(*key*)

remove_function(*name*)

remove_workflow(*name*)

set_artifact(*key*, *artifact*)

set_function(*name*, *function_object*, *function_dict*)

set_workflow(*name*, *workflow*)

property source
source url or git repo

property workflows
list of workflows specs used in this project

class mlrun.projects.**ProjectStatus**(*state=None*)
Bases: mlrun.model.ModelObj

mlrun.projects.**build_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *with_mlrun: Optional[bool] = None*, *skip_deployed: bool = False*, *image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *requirements: Optional[Union[str, List[str]]] = None*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*, *project_object=None*)
deploy ML function, build container with its dependencies

Parameters

- **function** – name of the function (in the project) or function object
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **image** – target image name/path
- **base_image** – base image name/path (commands and source code will be added to it)

- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret_name** – k8s secret for accessing the docker registry
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP
- **project_object** – override the project object to use, will default to the project set in the runtime context.
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP

mlrun.projects.**deploy_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *dashboard: str = ''*, *models: Optional[list] = None*, *env: Optional[dict] = None*, *tag: Optional[str] = None*, *verbose: Optional[bool] = None*, *builder_env: Optional[dict] = None*, *project_object=None*)

deploy real-time (nuclio based) functions

Parameters

- **function** – name of the function (in the project) or function object
- **dashboard** – url of the remote Nuclio dashboard (when not local)
- **models** – list of model items
- **env** – dict of extra environment variables
- **tag** – extra version tag

:param verbose add verbose prints/logs :param builder_env: env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token} :param project_object: override the project object to use, will default to the project set in the runtime context.

mlrun.projects.**get_or_create_project**(*name*, *context*, *url=None*, *secrets=None*, *init_git=False*, *subpath=''*, *clone=False*, *user_project=False*, *from_template=None*) → mlrun.projects.project.MlrunProject

Load a project from MLRun DB, or create/import if doesnt exist

example:

```
# load project from the DB (if exist) or the source repo
project = get_or_create_project("myproj", "./", "git://github.com/mlrun/demo-xgb-
↪project.git")
project.pull("development")  # pull the latest code from git
project.run("main", arguments={'data': data_url})  # run the workflow "main"
```

Parameters

- **context** – project local directory path
- **url** – name (in DB) or git or tar.gz or .zip sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git http://mysite/archived-project.zip
- **name** – project name
- **secrets** – key:secret dict or SecretsStore used to download sources

- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive)
- **clone** – if True, always clone (delete any existing content)
- **user_project** – add the current user name to the project name (for db:// prefixes)
- **from_template** – path to project YAML file that will be used as from_template (for new projects)

Returns project object

mlrun.projects.**load_project**(*context*, *url=None*, *name=None*, *secrets=None*, *init_git=False*, *subpath=''*, *clone=False*, *user_project=False*) → mlrun.projects.project.MlrunProject

Load an MLRun project from git or tar or dir

example:

```
# load the project and run the 'main' workflow
project = load_project("./", "git://github.com/mlrun/project-demo.git")
project.run("main", arguments={'data': data_url})
```

Parameters

- **context** – project local directory path
- **url** – name (in DB) or git or tar.gz or .zip sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git http://mysite/archived-project.zip <project-name>
- **name** – project name
- **secrets** – key:secret dict or SecretsStore used to download sources
- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive)
- **clone** – if True, always clone (delete any existing content)
- **user_project** – add the current user name to the project name (for db:// prefixes)

Returns project object

mlrun.projects.**new_project**(*name*, *context=None*, *init_git=False*, *user_project=False*, *remote=None*, *from_template=None*, *secrets=None*, *description=None*) → mlrun.projects.project.MlrunProject

Create a new MLRun project, optionally load it from a yaml/zip/git template

example:

```
# create a project with local and marketplace functions, a workflow, and an␣
↪artifact
project = mlrun.new_project("myproj", "./", init_git=True, description="my new␣
↪project")
project.set_function('prep_data.py', 'prep-data', image='mlrun/mlrun', handler=
↪'prep_data')
project.set_function('hub://sklearn_classifier', 'train')
project.set_artifact('data', Artifact(target_path=data_url))
project.set_workflow('main', "./myflow.py")
```

(continues on next page)

(continued from previous page)

```
project.save()

# run the "main" workflow (watch=True to wait for run completion)
project.run("main", watch=True)
```

example (load from template):

```
# create a new project from a zip template (can also use yaml/git templates)
# initialize a local git, and register the git remote path
project = mlrun.new_project("myproj", "./", init_git=True,
                            remote="git://github.com/mlrun/project-demo.git",
                            from_template="http://mysite/proj.zip")
project.run("main", watch=True)
```

Parameters

- **name** – project name
- **context** – project local directory path
- **init_git** – if True, will git init the context dir
- **user_project** – add the current user name to the provided project name (making it unique per user)
- **remote** – remote Git url
- **from_template** – path to project YAML/zip file that will be used as a template
- **secrets** – key:secret dict or SecretsStore used to download sources
- **description** – text describing the project

Returns project object

mlrun.projects.**run_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *handler: Optional[str] = None*, *name: str = ''*, *params: Optional[dict] = None*, *hyperparams: Optional[dict] = None*, *hyper_param_options: Optional[mlrun.model.HyperParamOptions] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[List[str]] = None*, *workdir: str = ''*, *labels: Optional[dict] = None*, *base_task: Optional[mlrun.model.RunTemplate] = None*, *watch: bool = True*, *local: bool = False*, *verbose: Optional[bool] = None*, *selector: Optional[str] = None*, *project_object=None*, *auto_build: Optional[bool] = None*) → Union[*mlrun.model.RunObject*, kfp.dsl._container_op.ContainerOp]

Run a local or remote task as part of a local/kubeflow pipeline

run_function() allow you to execute a function locally, on a remote cluster, or as part of an automated workflow function can be specified as an object or by name (str), when the function is specified by name it is looked up in the current project eliminating the need to redefine/edit functions.

when functions run as part of a workflow/pipeline (project.run()) some attributes can be set at the run level, e.g. local=True will run all the functions locally, setting artifact_path will direct all outputs to the same path. project runs provide additional notifications/reporting and exception handling. inside a Kubeflow pipeline (KFP) run_function() generates KFP "ContainerOps" which are used to form a DAG some behavior may differ between regular runs and deferred KFP runs.

example (use with function object):

```
function = mlrun.import_function("hub://sklearn_classifier")
run1 = run_function(function, params={"data": url})
```

example (use with project):

```
# create a project with two functions (local and from marketplace)
project = mlrun.new_project(project_name, "./proj)
project.set_function("mycode.py", "myfunc", image="mlrun/mlrun")
project.set_function("hub://sklearn_classifier", "train")

# run functions (refer to them by name)
run1 = run_function("myfunc", params={"x": 7})
run2 = run_function("train", params={"data": run1.outputs["data"]})
```

example (use in pipeline):

```
@dsl.pipeline(name="test pipeline", description="test")
def my_pipe(url=""):
    run1 = run_function("loaddata", params={"url": url})
    run2 = run_function("train", params={"data": run1.outputs["data"]})

project.run(workflow_handler=my_pipe, arguments={"param1": 7})
```

Parameters

- **function** – name of the function (in the project) or function object
- **handler** – name of the function handler
- **name** – execution name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **workdir** – default input artifacts path
- **labels** – labels to tag the job/run with ({key:val, ..})
- **base_task** – task object to use as base
- **watch** – watch/follow run log, True by default
- **local** – run the function locally vs on the runtime/cluster
- **verbose** – add verbose prints/logs
- **project_object** – override the project object to use, will default to the project set in the runtime context.
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs

Returns MLRun RunObject or KubeFlow containerOp

mlrun.run

class mlrun.run.**RunStatuses**
Bases: object

static all()

error = 'Error'

failed = 'Failed'

running = 'Running'

skipped = 'Skipped'

static stable_statuses()

succeeded = 'Succeeded'

static transient_statuses()

mlrun.run.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *code_output: str = ''*, *embed_code: bool = True*, *description: str = ''*, *requirements: Optional[Union[str, List[str]]] = None*, *categories: Optional[List[str]] = None*, *labels: Optional[Dict[str, str]] = None*, *with_doc: bool = True*, *ignored_tags=None*) → Union[mlrun.runtimes.mpijob.v1alpha1.MpiRuntimeV1Alpha1, mlrun.runtimes.mpijob.v1.MpiRuntimeV1, mlrun.runtimes.function.RemoteRuntime, mlrun.runtimes.serving.ServingRuntime, mlrun.runtimes.daskjob.DaskCluster, mlrun.runtimes.kubejob.KubejobRuntime, mlrun.runtimes.local.LocalRuntime, mlrun.runtimes.sparkjob.spark2job.Spark2Runtime, mlrun.runtimes.sparkjob.spark3job.Spark3Runtime, mlrun.runtimes.remotesparkjob.RemoteSparkRuntime]

Convenience function to insert code and configure an mlrun runtime.

Easiest way to construct a runtime type object. Provides the most often used configuration options for all runtimes as parameters.

Instantiated runtimes are considered 'functions' in mlrun, but they are anything from nuclio functions to generic kubernetes pods to spark jobs. Functions are meant to be focused, and as such limited in scope and size. Typically a function can be expressed in a single python module with added support from custom docker images and commands for the environment. The returned runtime object can be further configured if more customization is required.

One of the most important parameters is 'kind'. This is what is used to specify the chosen runtimes. The options are:

- local: execute a local python or shell script
- job: insert the code into a Kubernetes pod and execute it
- nuclio: insert the code into a real-time serverless nuclio function
- serving: insert code into orchestrated nuclio function(s) forming a DAG
- dask: run the specified python code / script as Dask Distributed job
- mpijob: run distributed Horovod jobs over the MPI job operator
- spark: run distributed Spark job using Spark Kubernetes Operator

- remote-spark: run distributed Spark job on remote Spark service

Learn more about function runtimes here: https://docs.mlrun.org/en/latest/runtimes/functions.html#function-runtimes

Parameters

- **name** – function name, typically best to use hyphen-case
- **project** – project used to namespace the function, defaults to 'default'
- **tag** – function tag to track multiple versions of the same function, defaults to 'latest'
- **filename** – path to .py/.ipynb file, defaults to current jupyter notebook
- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to 'main:handler'
- **kind** – function runtime type string - nuclio, job, etc. (see docstring for all options)
- **image** – base docker image to use for building the function container, defaults to None
- **code_output** – specify '.' to generate python module from the current jupyter notebook
- **embed_code** – indicates whether or not to inject the code directly into the function runtime spec, defaults to True
- **description** – short function description, defaults to ''
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **categories** – list of categories for mlrun function marketplace, defaults to None
- **labels** – immutable name/value pairs to tag the function with useful metadata, defaults to None
- **with_doc** – indicates whether to document the function parameters, defaults to True
- **ignored_tags** – notebook cells to ignore when converting notebooks to py code (separated by ';')

Returns pre-configured function object from a mlrun runtime class

example:

```
import mlrun

# create job function object from notebook code and add doc/metadata
fn = mlrun.code_to_function("file_utils", kind="job",
                            handler="open_archive", image="mlrun/mlrun",
                            description = "this function opens a zip archive into␣
↪a local/mounted folder",
                            categories = ["fileutils"],
                            labels = {"author": "me"})
```

example:

```
import mlrun
from pathlib import Path

# create file
Path("mover.py").touch()
```

(continues on next page)

(continued from previous page)

```
# create nuclio function object from python module call mover.py
fn = mlrun.code_to_function("nuclio-mover", kind="nuclio",
                            filename="mover.py", image="python:3.7",
                            description = "this function moves files from one␣
↪system to another",
                            requirements = ["pandas"],
                            labels = {"author": "me"})
```

mlrun.run.**download_object**(*url*, *target*, *secrets=None*)
download mlrun dataitem (from path/url to target path)

mlrun.run.**function_to_module**(*code=''*, *workdir=None*, *secrets=None*, *silent=False*)
Load code, notebook or mlrun function as .py module this function can import a local/remote py file or notebook or load an mlrun function object as a module, you can use this from your code, notebook, or another function (for common libs)

Note: the function may have package requirements which must be satisfied

example:

```
mod = mlrun.function_to_module('./examples/training.py')
task = mlrun.new_task(inputs={'infile.txt': '../examples/infile.txt'})
context = mlrun.get_or_create_ctx('myfunc', spec=task)
mod.my_job(context, p1=1, p2='x')
print(context.to_yaml())

fn = mlrun.import_function('hub://open_archive')
mod = mlrun.function_to_module(fn)
data = mlrun.run.get_dataitem("https://fpsignals-public.s3.amazonaws.com/
↪catsndogs.tar.gz")
context = mlrun.get_or_create_ctx('myfunc')
mod.open_archive(context, archive_url=data)
print(context.to_yaml())
```

Parameters

- **code** – path/url to function (.py or .ipynb or .yaml) OR function object
- **workdir** – code workdir
- **secrets** – secrets needed to access the URL (e.g.s3, v3io, ..)
- **silent** – do not raise on errors

Returns python module

mlrun.run.**get_dataitem**(*url*, *secrets=None*, *db=None*)
get mlrun dataitem object (from path/url)

mlrun.run.**get_object**(*url*, *secrets=None*, *size=None*, *offset=0*, *db=None*)
get mlrun dataitem body (from path/url)

mlrun.run.**get_or_create_ctx**(*name: str*, *event=None*, *spec=None*, *with_env: bool = True*, *rundb: str = ''*, *project: str = ''*, *upload_artifacts=False*)
called from within the user program to obtain a run context

the run context is an interface for receiving parameters, data and logging run results, the run context is read from the event, spec, or environment (in that order), user can also work without a context (local defaults mode)

all results are automatically stored in the "rundb" or artifact store, the path to the rundb can be specified in the call or obtained from env.

Parameters

- **name** – run name (will be overridden by context)
- **event** – function (nuclio Event object)
- **spec** – dictionary holding run spec
- **with_env** – look for context in environment vars, default True
- **rundb** – path/url to the metadata and artifact database
- **project** – project to initiate the context in (by default mlrun.mlctx.default_project)
- **upload_artifacts** – when using local context (not as part of a job/run), upload artifacts to the system default artifact path location

Returns execution context

Examples:

```
# load MLRUN runtime context (will be set by the runtime framework e.g. KubeFlow)
context = get_or_create_ctx('train')

# get parameters from the runtime context (or use defaults)
p1 = context.get_param('p1', 1)
p2 = context.get_param('p2', 'a-string')

# access input metadata, values, files, and secrets (passwords)
print(f'Run: {context.name} (uid={context.uid})')
print(f'Params: p1={p1}, p2={p2}')
print(f'accesskey = {context.get_secret("ACCESS_KEY")}')
input_str = context.get_input('infile.txt').get()
print(f'file: {input_str}')

# RUN some useful code e.g. ML training, data prep, etc.

# log scalar result values (job result metrics)
context.log_result('accuracy', p1 * 2)
context.log_result('loss', p1 * 3)
context.set_label('framework', 'sklearn')

# log various types of artifacts (file, web page, table), will be versioned and␣
↪visible in the UI
context.log_artifact('model.txt', body=b'abc is 123', labels={'framework':
↪'xgboost'})
context.log_artifact('results.html', body=b'<b> Some HTML <b>', viewer='web-app')
```

mlrun.run.**get_pipeline**(*run_id*, *namespace=None*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = <PipelinesFormat.summary: 'summary'>*, *project: Optional[str] = None*, *remote: bool = True*)

Get Pipeline status

Parameters

- **run_id** – id of pipelines run
- **namespace** – k8s namespace if not default

- **format** – Format of the results. Possible values are: - `summary` (default value) - Return summary of the object data. - `full` - Return full pipeline object.
- **project** – the project of the pipeline run
- **remote** – read kfp data from mlrun service (default=True)

Returns kfp run dict

`mlrun.run.`**`import_function`**(*url=''*, *secrets=None*, *db=''*, *project=None*, *new_name=None*)

Create function object from DB or local/remote YAML file

Function can be imported from function repositories (mlrun marketplace or local db), or be read from a remote URL (http(s), s3, git, v3io, ..) containing the function YAML

special URLs:

```
function marketplace: hub://{name}[:{tag}]
local mlrun db:       db://{project-name}/{name}[:{tag}]
```

examples:

```
function = mlrun.import_function("hub://sklearn_classifier")
function = mlrun.import_function("./func.yaml")
function = mlrun.import_function("https://raw.githubusercontent.com/org/repo/func.
↪yaml")
```

Parameters

- **url** – path/url to marketplace, db or function YAML file
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)
- **db** – optional, mlrun api/db path
- **project** – optional, target project for the function
- **new_name** – optional, override the imported function name

Returns function object

`mlrun.run.`**`import_function_to_dict`**(*url*, *secrets=None*)

Load function spec from local/remote YAML file

`mlrun.run.`**`list_pipelines`**(*full=False*, *page_token=''*, *page_size=None*, *sort_by=''*, *filter_=''*, *namespace=None*, *project='*'*, *format_: mlrun.api.schemas.pipeline.PipelinesFormat = <PipelinesFormat.metadata_only: 'metadata_only'>*) → Tuple[int, Optional[int], List[dict]]

List pipelines

Parameters

- **full** – Deprecated, use **format_** instead. if True will set **format_** to full, otherwise **format_** will be used
- **page_token** – A page token to request the next page of results. The token is acquired from the nextPageToken field of the response from the previous call or can be omitted when fetching the first page.
- **page_size** – The number of pipelines to be listed per page. If there are more pipelines than this number, the response message will contain a nextPageToken field you can use to fetch the next page.

- **sort_by** – Can be format of "field_name", "field_name asc" or "field_name desc" (Example, "name asc" or "id desc"). Ascending by default.
- **filter** – A url-encoded, JSON-serialized Filter protocol buffer, see: [filter.proto](https://github.com/kubeflow/pipelines/ blob/master/backend/api/filter.proto).
- **namespace** – Kubernetes namespace if other than default
- **project** – Can be used to retrieve only specific project pipelines. "*" for all projects. Note that filtering by project can't be used together with pagination, sorting, or custom filter.
- **format** – Control what will be returned (full/metadata_only/name_only)

mlrun.run.**load_func_code**(*command=''*, *workdir=None*, *secrets=None*, *name='name'*)

mlrun.run.**new_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *kind: str = ''*, *command: str = ''*, *image: str = ''*, *args: Optional[list] = None*, *runtime=None*, *mode=None*, *handler: Optional[str] = None*, *source: Optional[str] = None*, *requirements: Optional[Union[str, List[str]]] = None*, *kfp=None*)

Create a new ML function from base properties

example:

```
# define a container based function (the `training.py` must exist in the␣
↪container workdir)
f = new_function(command='training.py -x {x}', image='myrepo/image:latest', kind=
↪'job')
f.run(params={"x": 5})

# define a container based function which reads its source from a git archive
f = new_function(command='training.py -x {x}', image='myrepo/image:latest', kind=
↪'job',
                 source='git://github.com/mlrun/something.git')
f.run(params={"x": 5})

# define a local handler function (execute a local function handler)
f = new_function().run(task, handler=myfunction)
```

Parameters

- **name** – function name
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **kind** – runtime type (local, job, nuclio, spark, mpijob, dask, ..)
- **command** – command/url + args (e.g.: training.py –verbose)
- **image** – container image (start with '.' for default registry)
- **args** – command line arguments (override the ones in command)
- **runtime** – runtime (job, nuclio, spark, dask ..) object/dict store runtime specific details and preferences
- **mode** –

  runtime mode, "args" mode will push params into command template, example:
  command=`mycode.py –x {xparam}` will substitute the *{xparam}* with the value of the xparam param

"pass" mode will run the command as is in the container (not wrapped by mlrun), the command can use *{}* for parameters like in the "args" mode

- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to "main:handler"
- **source** – valid path to git, zip, or tar file, e.g. *git://github.com/mlrun/something.git*, *http://some/url/file.zip*
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **kfp** – reserved, flag indicating running within kubeflow pipeline

Returns function object

mlrun.run.**run_local**(*task=None*, *command=''*, *name: str = ''*, *args: Optional[list] = None*, *workdir=None*, *project: str = ''*, *tag: str = ''*, *secrets=None*, *handler=None*, *params: Optional[dict] = None*, *inputs: Optional[dict] = None*, *artifact_path: str = ''*, *mode: Optional[str] = None*, *allow_empty_resources=None*)

Run a task on function/code (.py, .ipynb or .yaml) locally,

example:

```
# define a task
task = new_task(params={'p1': 8}, out_path=out_path)
# run
run = run_local(spec, command='src/training.py', workdir='src')
```

or specify base task parameters (handler, params, ..) in the call:

```
run = run_local(handler=my_function, params={'x': 5})
```

Parameters

- **task** – task template object or dict (see RunTemplate)
- **command** – command/url/function
- **name** – ad hook function name
- **args** – command line arguments (override the ones in command)
- **workdir** – working dir to exec in
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **secrets** – secrets dict if the function source is remote (s3, v3io, ..)
- **handler** – pointer or name of a function handler
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **artifact_path** – default artifact output path

Returns run object

mlrun.run.**run_pipeline**(*pipeline*, *arguments=None*, *project=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *url=None*, *ttl=None*, *remote: bool = True*)

remote KubeFlow pipeline execution

Submit a workflow task to KFP via mlrun API service

Parameters

- **pipeline** – KFP pipeline function or path to .yaml/.zip pipeline file
- **arguments** – pipeline arguments
- **project** – name of project
- **experiment** – experiment name
- **run** – optional, run name
- **namespace** – Kubernetes namespace (if not using default)
- **url** – optional, url to mlrun API service
- **artifact_path** – target location/url for mlrun artifacts
- **ops** – additional operators (.apply() to all pipeline functions)
- **ttl** – pipeline ttl in secs (after that the pods will be removed)
- **remote** – read kfp data from mlrun service (default=True)

Returns kubeflow pipeline id

mlrun.run.**wait_for_pipeline_completion**(*run_id*, *timeout=3600*, *expected_statuses: Optional[List[str]] = None*, *namespace=None*, *remote=True*, *project: Optional[str] = None*)

Wait for Pipeline status, timeout in sec

Parameters

- **run_id** – id of pipelines run
- **timeout** – wait timeout in sec
- **expected_statuses** – list of expected statuses, one of [Succeeded | Failed | Skipped | Error], by default [Succeeded]
- **namespace** – k8s namespace if not default
- **remote** – read kfp data from mlrun service (default=True)
- **project** – the project of the pipeline

Returns kfp run dict

mlrun.run.**wait_for_runs_completion**(*runs: list*, *sleep=3*, *timeout=0*, *silent=False*)

wait for multiple runs to complete

Note: need to use *watch=False* in *.run()* so the run will not wait for completion

example:

```
# run two training functions in parallel and wait for the results
inputs = {'dataset': cleaned_data}
run1 = train.run(name='train_lr', inputs=inputs, watch=False,
                 params={'model_pkg_class': 'sklearn.linear_model.
↪LogisticRegression',
```

(continues on next page)

(continued from previous page)

```
                        'label_column': 'label'})
run2 = train.run(name='train_lr', inputs=inputs, watch=False,
                 params={'model_pkg_class': 'sklearn.ensemble.
↪RandomForestClassifier',
                        'label_column': 'label'})
completed = wait_for_runs_completion([run1, run2])
```

Parameters

- **runs** – list of run objects (the returned values of function.run())
- **sleep** – time to sleep between checks (in seconds)
- **timeout** – maximum time to wait in seconds (0 for unlimited)
- **silent** – set to True for silent exit on timeout

Returns list of completed runs

mlrun.runtimes

class mlrun.runtimes.**BaseRuntime**(*metadata=None*, *spec=None*)

Bases: mlrun.model.ModelObj

as_step(*runspec: Optional[mlrun.model.RunObject] = None*, *handler=None*, *name: str = ''*, *project: str = ''*, *params: Optional[dict] = None*, *hyperparams=None*, *selector=''*, *hyper_param_options: Optional[mlrun.model.HyperParamOptions] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[dict] = None*, *workdir: str = ''*, *artifact_path: str = ''*, *image: str = ''*, *labels: Optional[dict] = None*, *use_db=True*, *verbose=None*, *scrape_metrics=False*)

Run a local or remote task.

Parameters

- **runspec** – run template object or dict (see RunTemplate)
- **handler** – name of the function handler
- **name** – execution name
- **project** – project name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params
- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **artifact_path** – default artifact output path (replace out_path)
- **workdir** – default input artifacts path
- **image** – container image to use
- **labels** – labels to tag the job/run with ({key:val, ..})

- **use_db** – save function spec in the db (vs the workflow file)
- **verbose** – add verbose prints/logs
- **scrape_metrics** – whether to add the *mlrun/scrape-metrics* label to this run's resources

Returns KubeFlow containerOp

doc()

export(*target=''*, *format='.yaml'*, *secrets=None*, *strip=True*)
save function spec to a local/remote path (default to./function.yaml)

Parameters

- **target** – target path/url
- **format** – *.yaml* (default) or *.json*
- **secrets** – optional secrets dict/object for target path (e.g. s3)
- **strip** – strip status data

Returns self

fill_credentials()

full_image_path(*image=None*, *client_version: Optional[str] = None*)

is_deployed()

kind = 'base'

property metadata

run(*runspec: Optional[*mlrun.model.RunObject*] = None*, *handler=None*, *name: str = ''*, *project: str = ''*, *params: Optional[dict] = None*, *inputs: Optional[Dict[str, str]] = None*, *out_path: str = ''*, *workdir: str = ''*, *artifact_path: str = ''*, *watch: bool = True*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*, *hyperparams: Optional[Dict[str, list]] = None*, *hyper_param_options: Optional[*mlrun.model.HyperParamOptions*] = None*, *verbose=None*, *scrape_metrics: Optional[bool] = None*, *local=False*, *local_code_path=None*, *auto_build=None*) → *mlrun.model.RunObject*
Run a local or remote task.

Parameters

- **runspec** – run template object or dict (see RunTemplate)
- **handler** – pointer or name of a function handler
- **name** – execution name
- **project** – project name
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **out_path** – default artifact output path
- **artifact_path** – default artifact output path (will replace out_path)
- **workdir** – default input artifacts path
- **watch** – watch/follow run log

- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html#module-apscheduler.triggers.cron
- **hyperparams** – dict of param name and list of values to be enumerated e.g. {"p1": [1,2,3]} the default strategy is grid search, can specify strategy (grid, list, random) and other options in the hyper_param_options parameter
- **hyper_param_options** – dict or *HyperParamOptions* struct of hyper parameter options
- **verbose** – add verbose prints/logs
- **scrape_metrics** – whether to add the *mlrun/scrape-metrics* label to this run's resources
- **local** – run the function locally vs on the runtime/cluster
- **local_code_path** – path of the code for local runs & debug
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs

Returns run context object (RunObject) with run metadata, results and status

save(*tag=''*, *versioned=False*, *refresh=False*) → str

set_db_connection(*conn*, *is_api=False*)

set_label(*key*, *value*)

property spec

property status

store_run(*runobj:* mlrun.model.RunObject)

to_dict(*fields=None*, *exclude=None*, *strip=False*)
convert the object to a python dictionary

try_auto_mount_based_on_config()

property uri

validate_and_enrich_service_account(*allowed_service_account*, *default_service_account*)

verify_base_image()

with_code(*from_file=''*, *body=None*, *with_doc=True*)
Update the function code This function eliminates the need to build container images every time we edit the code

Parameters

- **from_file** – blank for current notebook, or path to .py/.ipynb file
- **body** – will use the body as the function code
- **with_doc** – update the document of the function parameters

Returns function object

with_requirements(*requirements: Union[str, List[str]]*)
add package requirements from file or list to build spec.

Parameters **requirements** – python requirements file path or list of packages

Returns function object

class mlrun.runtimes.**DaskCluster**(*spec=None*, *metadata=None*)

Bases: mlrun.runtimes.kubejob.KubejobRuntime

property client

close(*running=True*)

cluster()

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *show_on_failure: bool = False*)

deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

get_status()

gpus(*gpus*, *gpu_type='nvidia.com/gpu'*)

property initialized

is_deployed()

check if the function is deployed (have a valid container)

kind = 'dask'

property spec

property status

with_limits(*mem=None*, *cpu=None*, *gpus=None*, *gpu_type='nvidia.com/gpu'*)

set pod cpu/memory/gpu limits

with_requests(*mem=None*, *cpu=None*)

set requested (desired) pod cpu/memory resources

with_scheduler_limits(*mem=None*, *cpu=None*, *gpus=None*, *gpu_type='nvidia.com/gpu'*)

set scheduler pod resources limits

with_scheduler_requests(*mem=None*, *cpu=None*)

set scheduler pod resources requests

with_worker_limits(*mem=None*, *cpu=None*, *gpus=None*, *gpu_type='nvidia.com/gpu'*)

set worker pod resources limits

with_worker_requests(*mem=None*, *cpu=None*)

set worker pod resources requests

class mlrun.runtimes.**HandlerRuntime**(*metadata=None*, *spec=None*)

Bases: mlrun.runtimes.base.BaseRuntime, mlrun.runtimes.local.ParallelRunner

kind = 'handler'

class mlrun.runtimes.**KubejobRuntime**(*spec=None*, *metadata=None*)

Bases: mlrun.runtimes.pod.KubeResource

build_config(*image=''*, *base_image=None*, *commands: Optional[list] = None*, *secret=None*, *source=None*, *extra=None*, *load_source_on_run=None*, *with_mlrun=None*, *auto_build=None*)

specify builder configuration for the deploy operation

Parameters

- **image** – target image name/path
- **base_image** – base image name/path
- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret** – k8s secret for accessing the docker registry
- **source** – source git/tar archive to load code from in to the context/workdir e.g. git://github.com/mlrun/something.git#development
- **extra** – extra Dockerfile lines
- **load_source_on_run** – load the archive code into the container at runtime vs at build time
- **with_mlrun** – add the current mlrun package to the container build
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs

builder_status(*watch=True*, *logs=True*)

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*, *show_on_failure: bool = False*) → bool

deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

deploy_step(*image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *with_mlrun=True*, *skip_deployed=False*)

is_deployed()

check if the function is deployed (have a valid container)

kind = 'job'

with_source_archive(*source*, *workdir=None*, *handler=None*, *pull_at_runtime=True*)

load the code from git/tar/zip archive at runtime or build

Parameters

- **source** – valid path to git, zip, or tar file, e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **handler** – default function handler
- **workdir** – working dir relative to the archive root or absolute (e.g. './subdir')
- **pull_at_runtime** – load the archive into the container at job runtime vs on build/deploy

class mlrun.runtimes.**LocalRuntime**(*metadata=None*, *spec=None*)

Bases: mlrun.runtimes.base.BaseRuntime, mlrun.runtimes.local.ParallelRunner

is_deployed()

kind = 'local'

property spec

to_job(*image=''*)

with_source_archive(*source*, *workdir=None*, *handler=None*, *target_dir=None*)

load the code from git/tar/zip archive at runtime or build

Parameters

- **source** – valid path to git, zip, or tar file, e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **handler** – default function handler
- **workdir** – working dir relative to the archive root or absolute (e.g. './subdir')
- **target_dir** – local target dir for repo clone (by default its <current-dir>/code)

class mlrun.runtimes.**RemoteRuntime**(*spec=None*, *metadata=None*)

Bases: mlrun.runtimes.pod.KubeResource

add_model(*name*, *model_path*, ***kw*)

add_secrets_config_to_spec()

add_trigger(*name*, *spec*)

add a nuclio trigger object/dict

Parameters

- **name** – trigger name
- **spec** – trigger object or dict

add_v3io_stream_trigger(*stream_path*, *name='stream'*, *group='serving'*, *seek_to='earliest'*, *shards=1*, *extra_attributes=None*, *ack_window_size=None*, ***kwargs*)

add v3io stream trigger to the function

Parameters

- **stream_path** – v3io stream path (e.g. 'v3io:///projects/myproj/stream1')
- **name** – trigger name
- **group** – consumer group
- **seek_to** – start seek from: "earliest", "latest", "time", "sequence"
- **shards** – number of shards (used to set number of replicas)

- **extra_attributes** – key/value dict with extra trigger attributes
- **ack_window_size** – stream ack window size (the consumer group will be updated with the event id - ack_window_size, on failure the events in the window will be retransmitted)
- **kwargs** – extra V3IOStreamTrigger class attributes

add_volume(*local*, *remote*, *name='fs'*, *access_key=''*, *user=''*)

deploy(*dashboard=''*, *project=''*, *tag=''*, *verbose=False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*, *builder_env: Optional[dict] = None*)
Deploy the nuclio function to the cluster

Parameters

- **dashboard** – address of the nuclio dashboard service (keep blank for current cluster)
- **project** – project name
- **tag** – function tag
- **verbose** – set True for verbose logging
- **auth_info** – service AuthInfo
- **builder_env** – env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}

deploy_step(*dashboard=''*, *project=''*, *models=None*, *env=None*, *tag=None*, *verbose=None*, *use_function_from_db=None*)
return as a Kubeflow pipeline step (ContainerOp), recommended to use mlrun.deploy_function() instead

from_image(*image*)

invoke(*path: str*, *body: Optional[Union[str, bytes, dict]] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *dashboard: str = ''*, *force_external_address: bool = False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*)
Invoke the remote (live) function and return the results

example:

```
function.invoke("/api", body={"inputs": x})
```

Parameters

- **path** – request sub path (e.g. /images)
- **body** – request body (str, bytes or a dict for json requests)
- **method** – HTTP method (GET, PUT, ..)
- **headers** – key/value dict with http headers
- **dashboard** – nuclio dashboard address
- **force_external_address** – use the external ingress URL
- **auth_info** – service AuthInfo

kind = 'remote'

serving(*models: Optional[dict] = None*, *model_class=''*, *protocol=''*, *image=''*, *endpoint=''*, *explainer=False*, *workers=8*, *canary=None*)

set_config(*key*, *value*)

property spec

property status

with_http(*workers=8*, *port=0*, *host=None*, *paths=None*, *canary=None*, *secret=None*, *worker_timeout: Optional[int] = None*, *gateway_timeout: Optional[int] = None*, *trigger_name=None*, *annotations=None*, *extra_attributes=None*)

update/add nuclio HTTP trigger settings

Note: gateway timeout is the maximum request time before an error is returned, while the worker timeout if the max time a request will wait for until it will start processing, gateway_timeout must be greater than the worker_timeout.

Parameters

- **workers** – number of worker processes (default=8)
- **port** – TCP port
- **host** – hostname
- **paths** – list of sub paths
- **canary** – k8s ingress canary (% traffic value between 0 to 100)
- **secret** – k8s secret name for SSL certificate
- **worker_timeout** – worker wait timeout in sec (how long a message should wait in the worker queue before an error is returned)
- **gateway_timeout** – nginx ingress timeout in sec (request timeout, when will the gateway return an error)
- **trigger_name** – alternative nuclio trigger name
- **annotations** – key/value dict of ingress annotations
- **extra_attributes** – key/value dict of extra nuclio trigger attributes

Returns function object (self)

with_node_selection(***kwargs*)

Enables to control on which k8s node the job will run

Parameters

- **node_name** – The name of the k8s node
- **node_selector** – Label selector, only nodes with matching labels will be eligible to be picked
- **affinity** – Expands the types of constraints you can express - see https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity for details
- **tolerations** – Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints - see https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration for details

with_preemption_mode(***kwargs*)

Preemption mode controls whether pods can be scheduled on preemptible nodes. Tolerations, node selector, and affinity are populated on preemptible nodes corresponding to the function spec.

The supported modes are:

- **allow** - The function can be scheduled on preemptible nodes
- **constrain** - The function can only run on preemptible nodes
- **prevent** - The function cannot be scheduled on preemptible nodes

- **none** - No preemptible configuration will be applied on the function

The default preemption mode is configurable in mlrun.mlconf.function_defaults.preemption_mode, by default it's set to **prevent**

Parameters **mode** – allow | constrain | prevent | none defined in `PreemptionModes`

with_priority_class(***kwargs*)
Enables to control the priority of the pod If not passed - will default to mlrun.mlconf.default_function_priority_class_name

Parameters **name** – The name of the priority class

with_source_archive(*source*, *workdir=None*, *handler=None*, *runtime=''*)
Load nuclio function from remote source

Note: remote source may require credentials, those can be stored in the project secrets or passed in the function.deploy() using the builder_env dict, see the required credentials per source: v3io - "V3IO_ACCESS_KEY". git - "GIT_USERNAME", "GIT_PASSWORD". AWS S3 - "AWS_ACCESS_KEY_ID", "AWS_SECRET_ACCESS_KEY" or "AWS_SESSION_TOKEN".

param source a full path to the nuclio function source (code entry) to load the function from

param handler a path to the function's handler, including path inside archive/git repo

param workdir working dir relative to the archive root (e.g. 'subdir')

param runtime (optional) the runtime of the function (defaults to python:3.7)

Examples::

git:

fn.with_source_archive("git://github.com/org/repo#my-branch", handler="main:handler", workdir="path/inside/repo")

s3: fn.spec.nuclio_runtime = "golang" fn.with_source_archive("s3://my-bucket/path/in/bucket/my-functions-archive",

handler="my_func:Handler", workdir="path/inside/functions/archive", runtime="golang")

)

with_v3io(*local=''*, *remote=''*)
Add v3io volume to the function

Parameters

- **local** – local path (mount path inside the function container)
- **remote** – v3io path

class mlrun.runtimes.**RemoteSparkRuntime**(*spec=None*, *metadata=None*)
Bases: `mlrun.runtimes.kubejob.KubejobRuntime`

default_image = '.remote-spark-default-image'

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *show_on_failure: bool = False*)
deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)

- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

classmethod deploy_default_image()

is_deployed()
check if the function is deployed (have a valid container)

kind = 'remote-spark'

property spec

with_spark_service(*spark_service*, *provider='iguazio'*)
Attach spark service to function

class mlrun.runtimes.**ServingRuntime**(*spec=None*, *metadata=None*)
Bases: mlrun.runtimes.function.RemoteRuntime

MLRun Serving Runtime

add_child_function(*name*, *url=None*, *image=None*, *requirements=None*, *kind=None*)
in a multi-function pipeline add child function

example:

```
fn.add_child_function('enrich', './enrich.ipynb', 'mlrun/mlrun')
```

Parameters

- **name** – child function name
- **url** – function/code url, support .py, .ipynb, .yaml extensions
- **image** – base docker image for the function
- **requirements** – py package requirements file path OR list of packages
- **kind** – mlrun function/runtime kind

:return function object

add_model(*key: str*, *model_path: Optional[str] = None*, *class_name: Optional[str] = None*, *model_url: Optional[str] = None*, *handler: Optional[str] = None*, *router_step: Optional[str] = None*, *child_function: Optional[str] = None*, ***class_args*)
add ml model and/or route to the function.

Example, create a function (from the notebook), add a model class, and deploy:

```
fn = code_to_function(kind='serving')
fn.add_model('boost', model_path, model_class='MyClass', my_arg=5)
fn.deploy()
```

only works with router topology, for nested topologies (model under router under flow) need to add router to flow and use router.add_route()

Parameters

- **key** – model api key (or name:version), will determine the relative url/path
- **model_path** – path to mlrun model artifact or model directory file/object path
- **class_name** – V2 Model python class name or a model class instance (can also module.submodule.class and it will be imported automatically)
- **model_url** – url of a remote model serving endpoint (cannot be used with model_path)
- **handler** – for advanced users!, override default class handler name (do_event)
- **router_step** – router step name (to determine which router we add the model to in graphs with multiple router steps)
- **child_function** – child function name, when the model runs in a child function
- **class_args** – extra kwargs to pass to the model serving class __init__ (can be read in the model using .get_param(key) method)

add_secrets_config_to_spec()

deploy(*dashboard=''*, *project=''*, *tag=''*, *verbose=False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*, *builder_env: Optional[dict] = None*)
deploy model serving function to a local/remote cluster

Parameters

- **dashboard** – remote nuclio dashboard url (blank for local or auto detection)
- **project** – optional, override function specified project name
- **tag** – specify unique function tag (a different function service is created for every tag)
- **verbose** – verbose logging
- **auth_info** – The auth info to use to communicate with the Nuclio dashboard, required only when providing dashboard
- **builder_env** – env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}

kind = 'serving'

remove_states(*keys: list*)
remove one, multiple, or all states/models from the spec (blank list for all)

set_topology(*topology=None*, *class_name=None*, *engine=None*, *exist_ok=False*, ***class_args*) → Union[mlrun.serving.states.RootFlowStep, mlrun.serving.states.RouterStep]
set the serving graph topology (router/flow) and root class or params

examples:

```
# simple model router topology
graph = fn.set_topology("router")
fn.add_model(name, class_name="ClassifierModel", model_path=model_uri)

# async flow topology
graph = fn.set_topology("flow", engine="async")
graph.to("MyClass").to(name="to_json", handler="json.dumps").respond()
```

topology options are:

```
router - root router + multiple child route states/models
         route is usually determined by the path (route key/name)
         can specify special router class and router arguments

flow   - workflow (DAG) with a chain of states
         flow support "sync" and "async" engines, branches are not allowed in␣
↪sync mode
         when using async mode calling state.respond() will mark the state as␣
↪the
         one which generates the (REST) call response
```

Parameters

- **topology** –
  - graph topology, router or flow
- **class_name** –
  - optional for router, router class name/path or router object
- **engine** –
  - optional for flow, sync or async engine (default to async)
- **exist_ok** –
  - allow overriding existing topology
- **class_args** –
  - optional, router/flow class init args

:return graph object (fn.spec.graph)

set_tracking(*stream_path: Optional[str] = None*, *batch: Optional[int] = None*, *sample: Optional[int] = None*, *stream_args: Optional[dict] = None*)

set tracking stream parameters:

Parameters

- **stream_path** – path/url of the tracking stream e.g. v3io:///users/mike/mystream you can use the "dummy://" path for test/simulation
- **batch** – micro batch size (send micro batches of N records at a time)
- **sample** – sample size (send only one of N records)
- **stream_args** – stream initialization parameters, e.g. shards, retention_in_hours, ..

property spec

to_mock_server(*namespace=None*, *current_function='*'*, *track_models=False*, ***kwargs*) → mlrun.serving.server.GraphServer

create mock server object for local testing/emulation

Parameters

- **namespace** – one or list of namespaces/modules to search the steps classes/functions in
- **log_level** – log level (error | info | debug)
- **current_function** – specify if you want to simulate a child function, * for all functions

- **track_models** – allow model tracking (disabled by default in the mock server)

with_secrets(*kind*, *source*)

register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, example:

```
task.with_secrets('file', 'file.txt')
task.with_secrets('inline', {'key': 'val'})
task.with_secrets('env', 'ENV1,ENV2')
task.with_secrets('vault', ['secret1', 'secret2'...])

# If using an empty secrets list [] then all accessible secrets will be␣
↪available.
task.with_secrets('vault', [])

# To use with Azure key vault, a k8s secret must be created with the␣
↪following keys:
# kubectl -n <namespace> create secret generic azure-key-vault-secret \
#     --from-literal=tenant_id=<service principal tenant ID> \
#     --from-literal=client_id=<service principal client ID> \
#     --from-literal=secret=<service principal secret key>

task.with_secrets('azure_vault', {
    'name': 'my-vault-name',
    'k8s_secret': 'azure-key-vault-secret',
    # An empty secrets list may be passed ('secrets': []) to access all vault␣
↪secrets.
    'secrets': ['secret1', 'secret2'...]
})
```

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns The Runtime (function) object

mlrun.serving

class mlrun.serving.**GraphContext**(*level='info'*, *logger=None*, *server=None*, *nuclio_context=None*)

Bases: object

Graph context object

get_param(*key: str*, *default=None*)

get_remote_endpoint(*name*, *external=True*)

return the remote nuclio/serving function http(s) endpoint given its name

Parameters

- **name** – the function name/uri in the form [project/]function-name[:tag]
- **external** – return the external url (returns the external url by default)

get_secret(*key: str*)

push_error(*event*, *message*, *source=None*, ***kwargs*)

property server

class mlrun.serving.**GraphServer**(*graph=None*, *parameters=None*, *load_mode=None*, *function_uri=None*, *verbose=False*, *version=None*, *functions=None*, *graph_initializer=None*, *error_stream=None*, *track_models=None*, *secret_sources=None*, *default_content_type=None*)

Bases: mlrun.model.ModelObj

property graph

init_object(*namespace*)

init_states(*context*, *namespace*, *resource_cache: Optional[mlrun.datastore.store_resources.ResourceCache] = None*, *logger=None*, *is_mock=False*)

for internal use, initialize all steps (recursively)

kind = 'server'

run(*event*, *context=None*, *get_body=False*, *extra_args=None*)

set_current_function(*function*)

set which child function this server is currently running on

set_error_stream(*error_stream*)

set/initialize the error notification stream

test(*path: str = '/'*, *body: Optional[Union[str, bytes, dict]] = None*, *method: str = ''*, *headers: Optional[str] = None*, *content_type: Optional[str] = None*, *silent: bool = False*, *get_body: bool = True*, *event_id: Optional[str] = None*, *trigger: Optional[mlrun.serving.server.MockTrigger] = None*, *offset=None*, *time=None*)

invoke a test event into the server to simulate/test server behavior

example:

```
server = create_graph_server()
server.add_model("my", class_name=MyModelClass, model_path="{path}", z=100)
print(server.test("my/infer", testdata))
```

Parameters

- **path** – api path, e.g. (/{router.url_prefix}/{model-name}/..) path
- **body** – message body (dict or json str/bytes)
- **method** – optional, GET, POST, ..
- **headers** – optional, request headers, ..
- **content_type** – optional, http mime type
- **silent** – don't raise on error responses (when not 20X)
- **get_body** – return the body as py object (vs serialize response into json)
- **event_id** – specify the unique event ID (by default a random value will be generated)
- **trigger** – nuclio trigger info or mlrun.serving.server.MockTrigger class (holds kind and name)
- **offset** – trigger offset (for streams)
- **time** – event time Datetime or str, default to now()

wait_for_completion()
wait for async operation to complete

class mlrun.serving.**QueueStep**(*name: Optional[str] = None*, *path: Optional[str] = None*, *after: Optional[list] = None*, *shards: Optional[int] = None*, *retention_in_hours: Optional[int] = None*, *trigger_args: Optional[dict] = None*, ***options*)
Bases: mlrun.serving.states.BaseStep

queue step, implement an async queue or represent a stream

after_state(*after*)

after_step(*after*)
specify the previous step name

property async_object

default_shape = 'cds'

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'queue'

run(*event*, **args*, ***kwargs*)

class mlrun.serving.**RouterStep**(*class_name: Optional[Union[str, type]] = None*, *class_args: Optional[dict] = None*, *handler: Optional[str] = None*, *routes: Optional[list] = None*, *name: Optional[str] = None*, *function: Optional[str] = None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*)
Bases: mlrun.serving.states.TaskStep

router step, implement routing logic for running child routes

add_route(*key*, *route=None*, *class_name=None*, *handler=None*, *function=None*, ***class_args*)
add child route step or class to the router

Parameters

- **key** – unique name (and route path) for the child step
- **route** – child step object (Task, ..)
- **class_name** – class name to build the route step from (when route is not provided)
- **class_args** – class init arguments
- **handler** – class handler to invoke on run/event
- **function** – function this step should run in

clear_children(*routes: list*)
clear child steps (routes)

default_shape = 'doubleoctagon'

get_children()
get child steps (routes)

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'router'

plot(*filename=None*, *format=None*, *source=None*, ***kw*)
plot/save a graphviz plot

property routes
child routes/steps, traffic is routed to routes based on router logic

class mlrun.serving.**TaskStep**(*class_name: Optional[Union[str, type]] = None*, *class_args: Optional[dict] = None*, *handler: Optional[str] = None*, *name: Optional[str] = None*, *after: Optional[list] = None*, *full_event: Optional[bool] = None*, *function: Optional[str] = None*, *responder: Optional[bool] = None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*)
Bases: mlrun.serving.states.BaseStep

task execution step, runs a class or handler

property async_object
return the sync or async (storey) class instance

clear_object()

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'task'

respond()
mark this step as the responder.

step output will be returned as the flow result, no other step can follow

run(*event*, **args*, ***kwargs*)
run this step, in async flows the run is done through storey

class mlrun.serving.**V2ModelServer**(*context=None*, *name: Optional[str] = None*, *model_path: Optional[str] = None*, *model=None*, *protocol=None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, ***kwargs*)
Bases: mlrun.serving.utils.StepToDict

base model serving class (v2), using similar API to KFServing v2 and Triton

base model serving class (v2), using similar API to KFServing v2 and Triton

The class is initialized automatically by the model server and can run locally as part of a nuclio serverless function, or as part of a real-time pipeline default model url is: /v2/models/<model>[/versions/<ver>]/operation

You need to implement two mandatory methods: load() - download the model file(s) and load the model into memory predict() - accept request payload and return prediction/inference results

you can override additional methods : preprocess, validate, postprocess, explain you can add custom api endpoint by adding method op_xx(event), will be invoked by calling the <model-url>/xx (operation = xx)

model server classes are subclassed (subclass implements the *load()* and *predict()* methods) the subclass can be added to a serving graph or to a model router

defining a sub class:

```
class MyClass(V2ModelServer):
    def load(self):
        # load and initialize the model and/or other elements
        model_file, extra_data = self.get_model(suffix='.pkl')
        self.model = load(open(model_file, "rb"))
```

(continues on next page)

(continued from previous page)

```
    def predict(self, request):
        events = np.array(request['inputs'])
        dmatrix = xgb.DMatrix(events)
        result: xgb.DMatrix = self.model.predict(dmatrix)
        return {"outputs": result.tolist()}
```

usage example:

```
# adding a model to a serving graph using the subclass MyClass
# MyClass will be initialized with the name "my", the model_path, and an arg␣
↪called my_param
graph = fn.set_topology("router")
fn.add_model("my", class_name="MyClass", model_path="<model-uri>>", my_param=5)
```

Parameters

- **context** – for internal use (passed in init)
- **name** – step name
- **model_path** – model file/dir or artifact path
- **model** – model object (for local testing)
- **protocol** – serving API protocol (default “v2”)
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {“data”: {“a”: 5, “b”: 7}}, input_path=”data.b” means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {“x”: 5} , result_path=”resp” means the returned response will be written to event[“y”] resulting in {“x”: 5, “resp”: <result>}
- **kwargs** – extra arguments (can be accessed using self.get_param(key))

do_event(*event*, **args*, ***kwargs*)
main model event handler method

explain(*request: Dict*) → Dict
model explain operation

get_model(*suffix=''*)
get the model file(s) and metadata from model store

the method returns a path to the model file and the extra data (dict of dataitem objects) it also loads the model metadata into the self.model_spec attribute, allowing direct access to all the model metadata attributes.

get_model is usually used in the model .load() method to init the model .. rubric:: Examples

```
def load(self):
    model_file, extra_data = self.get_model(suffix='.pkl')
    self.model = load(open(model_file, "rb"))
    categories = extra_data['categories'].as_df()
```

Parameters **suffix** (*str*) – optional, model file suffix (when the model_path is a directory)

Returns

- *str* – (local) model file
- *dict* – extra dataitems dictionary

get_param(*key: str*, *default=None*)
get param by key (specified in the model or the function)

load()
model loading function, see also .get_model() method

logged_results(*request: dict*, *response: dict*, *op: str*)
hook for controlling which results are tracked by the model monitoring

this hook allows controlling which input/output data is logged by the model monitoring allow filtering out columns or adding custom values, can also be used to monitor derived metrics for example in image classification calculate and track the RGB values vs the image bitmap

the request["inputs"] holds a list of input values/arrays, the response["outputs"] holds a list of corresponding output values/arrays (the schema of the input/output fields is stored in the model object), this method should return lists of alternative inputs and outputs which will be monitored

Parameters

- **request** – predict/explain request, see model serving docs for details
- **response** – result from the model predict/explain (after postprocess())
- **op** – operation (predict/infer or explain)

Returns the input and output lists to track

post_init(*mode='sync'*)
sync/async model loading, for internal use

postprocess(*request: Dict*) → Dict
postprocess, before returning response

predict(*request: Dict*) → Dict
model prediction operation

preprocess(*request: Dict*, *operation*) → Dict
preprocess the event body before validate and action

set_metric(*name: str*, *value*)
set real time metric (for model monitoring)

validate(*request*, *operation*)
validate the event body (after preprocess)

class mlrun.serving.**VotingEnsemble**(*context=None*, *name: Optional[str] = None*, *routes=None*, *protocol: Optional[str] = None*, *url_prefix: Optional[str] = None*, *health_prefix: Optional[str] = None*, *vote_type=None*, *executor_type=None*, *prediction_col_name=None*, ***kwargs*)
Bases: mlrun.serving.routers.BaseModelRouter

Voting Ensemble

The *VotingEnsemble* class enables you to apply prediction logic on top of the different added models.

You can use it by calling: - <prefix>/<model>[/versions/<ver>]/operation

Sends the event to the specific <model>[/versions/<ver>]

- **<prefix>/operation** Sends the event to all models and applies *vote(self, event)*

The *VotingEnsemble* applies the following logic: Incoming Event -> Router Preprocessing -> Send to model/s -> Apply all model/s logic (Preprocessing -> Prediction -> Postprocessing) -> Router Voting logic -> Router Postprocessing -> Response

This enables you to do the general preprocessing and postprocessing steps once on the router level, with only model-specific adjustments at the model level.

- When enabling model tracking via *set_tracking()* the ensemble logic

predictions will appear with model name as the given VotingEnsemble name or "VotingEnsemble" by default.

Example:

```
# Define a serving function
# Note: You can point the function to a file containing you own Router or␣
↪Classifier Model class
#       this basic class supports sklearn based models (with `<model>.predict()`␣
↪api)
fn = mlrun.code_to_function(name='ensemble',
                            kind='serving',
                            filename='model-server.py'
                            image='mlrun/ml-models')

# Set the router class
# You can set your own classes by simply changing the `class_name`
fn.set_topology(class_name='mlrun.serving.routers.VotingEnsemble')

# Add models
fn.add_model(<model_name>, <model_path>, <model_class_name>)
fn.add_model(<model_name>, <model_path>, <model_class_name>)
```

The VotingEnsemble applies its logic using the *logic(predictions)* function. The *logic()* function receives an array of (# samples, # predictors) which you can then use to apply whatever logic you may need.

If we use this *VotingEnsemble* as an example, the *logic()* function tries to figure out whether you are trying to do a **classification** or a **regression** prediction by the prediction type or by the given *vote_type* parameter. Then we apply the appropriate *max_vote()* or *mean_vote()* which calculates the actual prediction result and returns it as the VotingEnsemble's prediction.

Parameters

- **context** – for internal use (passed in init)
- **name** – step name
- **routes** – for internal use (routes passed in init)
- **protocol** – serving API protocol (default "v2")
- **url_prefix** – url prefix for the router (default /v2/models)
- **health_prefix** – health api url prefix (default /v2/health)
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **vote_type** – Voting type to be used (from *VotingTypes*). by default will try to self-deduct upon the first event:

- float prediction type: regression
- int prediction type: classification

- **executor_type** – Parallelism mechanism, out of *ParallelRunnerModes*, by default *threads*
- **prediction_col_name** – The dict key for the predictions column in the model's responses output. Example: If the model returns

  {id: <id>, model_name: <name>, outputs: {..., prediction: [<predictions>], ...}} the prediction_col_name should be *prediction*.

  by default, *prediction*
- **kwargs** – extra arguments

do_event(*event*, **args*, ***kwargs*)
Handles incoming requests.

Parameters **event** (*nuclio.Event*) – Incoming request as a nuclio.Event.

Returns Event response after running the requested logic

Return type Response

extract_results_from_response(*response*)
Extracts the prediction from the model response. This function is used to allow multiple model return types. and allow for easy extension to the user's ensemble and models best practices.

Parameters **response** (*Union[List, Dict]*) – The model response's *output* field.

Returns The model's predictions

Return type List

logic(*predictions*)

post_init(*mode='sync'*)

validate(*request*)
Validate the event body (after preprocessing)

Parameters **request** (*dict*) – Event body.

Returns Event body after validation

Return type dict

Raises

- **Exception** – *inputs* key not found in *request*
- **Exception** – *inputs* should be of type List

mlrun.serving.**create_graph_server**(*parameters={}*, *load_mode=None*, *graph=None*, *verbose=False*, *current_function=None*, ***kwargs*) → mlrun.serving.server.GraphServer
create graph server host/emulator for local or test runs

Usage example:

```
server = create_graph_server(graph=RouterStep(), parameters={})
server.init(None, globals())
server.graph.add_route("my", class_name=MyModelClass, model_path="{path}", z=100)
print(server.test("/v2/models/my/infer", testdata))
```

class mlrun.serving.remote.**BatchHttpRequests**(*url: Optional[str] = None*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints in parallel

class for calling remote endpoints in parallel

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
function = mlrun.new_function("myfunc", kind="serving")
flow = function.set_topology("flow", engine="async")
flow.to(
    BatchHttpRequests(
        url_expression="event['url']",
        body_expression="event['data']",
        method="POST",
        input_path="req",
        result_path="resp",
    )
).respond()

server = function.to_mock_server()
# request contains a list of elements, each with url and data
request = [{"url": f"{base_url}/{i}", "data": i} for i in range(2)]
resp = server.test(body={"req": request})
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url)
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"
- **body_expression** – an expression for getting the request body from the event, e.g. "event['data']"
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}

- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

__init__(*url: Optional[str] = None*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints in parallel

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
function = mlrun.new_function("myfunc", kind="serving")
flow = function.set_topology("flow", engine="async")
flow.to(
    BatchHttpRequests(
        url_expression="event['url']",
        body_expression="event['data']",
        method="POST",
        input_path="req",
        result_path="resp",
    )
).respond()

server = function.to_mock_server()
# request contains a list of elements, each with url and data
request = [{"url": f"{base_url}/{i}", "data": i} for i in range(2)]
resp = server.test(body={"req": request})
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url)
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"
- **body_expression** – an expression for getting the request body from the event, e.g. "event['data']"
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}

- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

class mlrun.serving.remote.**RemoteStep**(*url: str*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *max_in_flight=None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints

class for calling remote endpoints

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
flow = function.set_topology("flow", engine="async")
flow.to(name="step1", handler="func1")
    .to(RemoteStep(name="remote_echo", url="https://myservice/path", method="POST
↪"))
    .to(name="laststep", handler="func2").respond()
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url), use *$path* to use the event.path
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"
- **body_expression** – an expression for getting the request body from the event, e.g. "event['data']"
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

__init__(*url: str*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *max_in_flight=None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
flow = function.set_topology("flow", engine="async")
flow.to(name="step1", handler="func1")
    .to(RemoteStep(name="remote_echo", url="https://myservice/path", method=
↪"POST"))
    .to(name="laststep", handler="func2").respond()
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url), use *$path* to use the event.path
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"
- **body_expression** – an expression for getting the request body from the event, e.g. "event['data']"
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

storey.transformations - Graph transformations

Graph transformations are contained in the *storey.transformations* module. For convenience, they can also be imported directly from the storey package. Note that the transformation functions are actually encapsulated in classes, so that they can be referenced by name of class from graph step definitions.

class storey.transformations.**AggregateByKey**(*aggregates: Union[List[storey.dtypes.FieldAggregator], List[Dict[str, object]]], table: Union[storey.table.Table, str], key: Optional[Union[str, Callable[[storey.dtypes.Event], object]]] = None, emit_policy: Union[storey.dtypes.EmitPolicy, Dict[str, object]] = <storey.dtypes.EmitEveryEvent object>, augmentation_fn: Optional[Callable[[storey.dtypes.Event, Dict[str, object]], storey.dtypes.Event]] = None, enrich_with: Optional[List[str]] = None, aliases: Optional[Dict[str, str]] = None, use_windows_from_schema: bool = False, **kwargs*)

Aggregates the data into the table object provided for later persistence, and outputs an event enriched with the requested aggregation features. Persistence is done via the *NoSqlTarget* step and based on the Cache object persistence settings.

Parameters

- **aggregates** – List of aggregates to apply for each event. accepts either list of FieldAggregators or a dictionary describing FieldAggregators.
- **table** – A Table object or name for persistence of aggregations. If a table name is provided, it will be looked up in the context object passed in kwargs.
- **key** – Key field to aggregate by, accepts either a string representing the key field or a key extracting function. Defaults to the key in the event's metadata. (Optional)
- **emit_policy** – Policy indicating when the data will be emitted. Defaults to EmitEveryEvent
- **augmentation_fn** – Function that augments the features into the event's body. Defaults to updating a dict. (Optional)
- **enrich_with** – List of attributes names from the associated storage object to be fetched and added to every event. (Optional)
- **aliases** – Dictionary specifying aliases for enriched or aggregate columns, of the format *{'col_name': 'new_col_name'}*. (Optional)

class storey.transformations.**Assert**(***kwargs*)

Exposes an API for testing the flow between steps.

class storey.transformations.**Batch**(*max_events: Optional[int] = None, flush_after_seconds: Optional[int] = None, key: Optional[Union[str, Callable[[storey.dtypes.Event], str]]] = None, **kwargs*)

Batches events into lists of up to max_events events. Each emitted list contained max_events events, unless flush_after_seconds seconds have passed since the first event in the batch was received, at which the batch is emitted with potentially fewer than max_events event.

Parameters

- **max_events** – Maximum number of events per emitted batch. Set to None to emit all events in one batch on flow termination.
- **flush_after_seconds** – Maximum number of seconds to wait before a batch is emitted.
- **key** – The key by which events are grouped. By default (None), events are not grouped. Other options may be: Set a '$key' to group events by the Event.key property. set a 'str' key to group events by Event.body[str]. set a Callable[Any, Any] to group events by a a custom key extractor.

class storey.transformations.**Choice**(*choice_array*, *default=None*, ***kwargs*)

Redirects each input element into at most one of multiple downstreams.

Parameters

- **choice_array** (*tuple of (Flow, Function (Event=>boolean))*) – a list of (downstream, condition) tuples, where downstream is a step and condition is a function. The first condition in the list to evaluate as true for an input element causes that element to be redirected to that downstream step.
- **default** (*Flow*) – a default step for events that did not match any condition in choice_array. If not set, elements that don't match any condition will be discarded.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (Choice).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**Extend**(*fn*, *long_running=None*, ***kwargs*)

Adds fields to each incoming event.

Parameters

- **fn** (*Function (Event=>Dict)*) – Function to transform each event to a dictionary. The fields in the returned dictionary are then added to the original event.
- **long_running** – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other

concurrent processing. Default is False. :type long_running: boolean :param name: Name of this step, as it should appear in logs. Defaults to class name (Extend). :type name: string :param full_event: Whether user functions should receive and/or return Event objects (when True), or only the payload (when False).

Defaults to False.

class storey.transformations.**Filter**(*fn*, *long_running=None*, ***kwargs*)

Filters events based on a user-provided function.

Parameters

- **fn** (*Function (Event=>boolean)*) – Function to decide whether to keep each event.
- **long_running** – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other

concurrent processing. Default is False. :type long_running: boolean :param name: Name of this step, as it should appear in logs. Defaults to class name (Filter). :type name: string :param full_event: Whether user functions should receive and/or return Event objects (when True), or only the payload (when False).

Defaults to False.

class storey.transformations.**FlatMap**(*fn*, *long_running=None*, ***kwargs*)

Maps, or transforms, each incoming event into any number of events.

Parameters

- **fn** (*Function (Event=>list of Event)*) – Function to transform each event to a list of events.
- **long_running** – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other

concurrent processing. Default is False. :type long_running: boolean :param name: Name of this step, as it should appear in logs. Defaults to class name (FlatMap). :type name: string :param full_event: Whether user functions should receive and/or return Event objects (when True), or only the payload (when False).

Defaults to False.

storey.transformations.**Flatten**(***kwargs*)

Flatten is equivalent to FlatMap(lambda x: x).

class storey.transformations.**ForEach**(*fn*, *long_running=None*, ***kwargs*)

Applies given function on each event in the stream, passes original event downstream.

class storey.transformations.**JoinWithTable**(*table: Union[storey.table.Table, str]*, *key_extractor: Union[str, Callable[[storey.dtypes.Event], str]]*, *attributes: Optional[List[str]] = None*, *inner_join: bool = False*, *join_function: Optional[Callable[[Any, Dict[str, object]], Any]] = None*, ***kwargs*)

Joins each event with data from the given table.

Parameters

- **table** – A Table object or name to join with. If a table name is provided, it will be looked up in the context.
- **key_extractor** – Key's column name or a function for extracting the key, for table access from an event.
- **attributes** – A comma-separated list of attributes to be queried for. Defaults to all attributes.
- **inner_join** – Whether to drop events when the table does not have a matching entry (join_function won't be called in such a case). Defaults to False.
- **join_function** – Joins the original event with relevant data received from the storage. Event is dropped when this function returns None. Defaults to assume the event's body is a dict-like object and updating it.
- **name** – Name of this step, as it should appear in logs. Defaults to class name (JoinWithTable).
- **full_event** – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

- **context** – Context object that holds global configurations and secrets.

class storey.transformations.**Map**(*fn*, *long_running=None*, ***kwargs*)

Maps, or transforms, incoming events using a user-provided function.

Parameters

- **fn** (*Function (Event=>Event)*) – Function to apply to each event
- **long_running** – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other

concurrent processing. Default is False. :type long_running: boolean :param name: Name of this step, as it should appear in logs. Defaults to class name (Map). :type name: string :param full_event: Whether user functions should receive and/or return Event objects (when True), or only the payload (when False).

Defaults to False.

class storey.transformations.**MapClass**(*long_running=None*, ***kwargs*)

Similar to Map, but instead of a function argument, this class should be extended and its do() method overridden.

class storey.transformations.**MapWithState**(*initial_state*, *fn*, *group_by_key=False*, ***kwargs*)

Maps, or transforms, incoming events using a stateful user-provided function, and an initial state, which may be a database table.

Parameters

- **initial_state** (*dictionary or Table if group_by_key is True. Any object otherwise.*) – Initial state for the computation. If group_by_key is True, this must be a dictionary or a Table object.
- **fn** (*Function ((Event, state)=>(Event, state))*) – A function to run on each event and the current state. Must yield an event and an updated state.
- **group_by_key** (*boolean*) – Whether the state is computed by key. Optional. Default to False.
- **full_event** (*boolean*) – Whether fn will receive and return an Event object or only the body (payload). Optional. Defaults to False (body only).

class storey.transformations.**Partition**(*predicate: Callable[[Any], bool]*, ***kwargs*)

Partitions events by calling a predicate function on each event. Each processed event results in a *Partitioned* namedtuple of (left=Optional[Event], right=Optional[Event]).

For a given event, if the predicate function results in *True*, the event is assigned to *left*. Otherwise, the event is assigned to *right*.

Parameters **predicate** – A predicate function that results in a boolean.

class storey.transformations.**ReifyMetadata**(*mapping: Iterable[str]*, ***kwargs*)

Inserts event metadata into the event body. :param mapping: Dictionary from event attribute name to entry key in the event body (which must be a dictionary). Alternatively, an iterable of names may be provided, and these will be used as both attribute name and entry key. :param name: Name of this step, as it should appear in logs. Defaults to class name (ReifyMetadata). :type name: string

class storey.transformations.**SampleWindow**(*window_size: int, emit_period: storey.steps.sample.EmitPeriod = <EmitPeriod.FIRST: 1>, emit_before_termination: bool = False, key: Optional[Union[str, Callable[[storey.dtypes.Event], str]]] = None, **kwargs*)

Emits a single event in a window of *window_size* events, in accordance with *emit_period* and *emit_before_termination*.

Parameters

- **window_size** – The size of the window we want to sample a single event from.
- **emit_period** – What event should this step emit for each *window_size* (default: EmitPeriod.First).

Available options: 1.1) EmitPeriod.FIRST - will emit the first event in a window *window_size* events. 1.2) EmitPeriod.LAST - will emit the last event in a window of *window_size* events.

Parameters **emit_before_termination** – On termination signal, should the step emit the last event it seen (default: False).

Available options: 2.1) True - The last event seen will be emitted downstream. 2.2) False - The last event seen will NOT be emitted downstream.

Parameters **key** – The key by which events are sampled. By default (None), events are not sampled by key. Other options may be: Set to '$key' to sample events by the Event.key property. set to 'str' key to sample events by Event.body[str]. set a Callable[[Event], str] to sample events by a custom key extractor.

class storey.transformations.**SendToHttp**(*request_builder*, *join_from_response*, ***kwargs*)

Joins each event with data from any HTTP source. Used for event augmentation.

Parameters

- **request_builder** (*Function (Event=>HttpRequest)*) – Creates an HTTP request from the event. This request is then sent to its destination.
- **join_from_response** (*Function ((Event, HttpResponse)=>Event)*) – Joins the original event with the HTTP response into a new event.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (SendToHttp).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**ToDataFrame**(*index: Optional[str] = None, columns: Optional[List[str]] = None, **kwargs*)

Create pandas data frame from events. Can appear in the middle of the flow, as opposed to ReduceToDataFrame

Parameters

- **index** – Name of the column to be used as index. Optional. If not set, DataFrame will be range indexed.
- **columns** – List of column names to be passed as-is to the DataFrame constructor. Optional.

for additional params, see documentation of `storey.flow.Flow`

See also the *index of all functions and classes*.

2.1.23 Command-Line Interface (Tech preview)

- *CLI commands*
- *Building and running a function from a Git Repository*
- *Using a sources archive*

CLI commands

Use the following commands of the MLRun command-line interface (CLI) — `mlrun` — to build and run MLRun functions:

- *build*
- *clean*
- *config*
- *get*
- *logs*
- *project*
- *run*
- *version*
- *watch*
- *watch-stream*

Each command supports many flags, some of which are listed in their relevant sections. To view all the flags of a command, run `mlrun <command name> --help`.

build

Use the `build` CLI command to build all the function dependencies from the function specification into a function container (Docker image).

Usage: mlrun build [OPTIONS] FUNC_URL

Example: `mlrun build myfunc.yaml`

Flag	Description
name TEXT	Function name
project TEXT	Project name
tag TEXT	Function tag
-i, image TEXT	Target image path
-s, source TEXT	Path/URL of the function source code. A PY file, or if `-a|--archive` is specified, a directory to archive. (Default: './')
-b, base-image TEXT	Base Docker image
-c, command TEXT	Build commands; for example, '-c pip install pandas'
secret-name TEXT	Name of a container-registry secret
-a, archive TEXT	Path/URL of a target function-sources archive directory: as part of the build, the function sources (see `-s|--source`) are archived into a TAR file and then extracted into the archive directory
silent	Do not show build logs
with-mlrun	Add the MLRun package ("mlrun")
db TEXT	Save the run results to path or DB url
-r, runtime TEXT	Function spec dict, for pipeline usage
kfp	Running inside Kubeflow Piplines, do not use
skip	Skip if already deployed

Note: For information about using the `-a|--archive` option to create a function-sources archive, see *Using a Sources Archive* later in this tutorial.

clean

Use the `clean` CLI command to clean runtime resources. When run without any flags, it cleans the resources for all runs of all runtimes.

Usage: mlrun clean [OPTIONS] [KIND] [id]

Examples:

- Clean resources for all runs of all runtimes: `mlrun clean`
- Clean resources for all runs of a specific kind (e.g. job): `mlrun clean job`
- Clean resources for specific job (by uid): `mlrun clean mpijob 15d04c19c2194c0a8efb26ea3017254b`

Flag	Description
kind	Clean resources for all runs of a specific kind (e.g. job).
id	Delete the resources of the mlrun object twith this identifier. For most function runtimes, runtime resources are per Run, and the identifier is the Run's UID. For DASK runtime, the runtime resources are per Function, and the identifier is the Function's name.

Options	Description
api	URL of the mlrun-api service.
-ls, label-selector	Delete only runtime resources matching the label selector.
-f, force	Delete the runtime resource even if they're not in terminal state or if the grace period didn't pass.
-gp, grace-period	Grace period, in seconds, given to the runtime resource before they are actually removed, counted from the moment they moved to the terminal state.

config

Use the `config` CLI command to show the mlrun client environment configuration, such as location of artifacts and api.

Example: `mlrun config`

get

Use the `get` CLI command to list one or more objects per kind/class.

Usage: get pods | runs | artifacts | func [name]

Examples:

- `mlrun get runs --project getting-started-admin`
- `mlrun get pods --project getting-started-admin`
- `mlrun get artifacts --project getting-started-admin`
- `mlrun get func prep-data --project getting-started-admin`

Flag	Description
name	Name of object to return
-s, selector	Label selector
-n, namespace	Kubernetes namespace
uid	Object ID
project	Project name to return
-t, tag	Artifact/function tag of object to return
db	db path/url of object to return

logs

Use the `logs` CLI command to get or watch task logs.

Usage: logs [OPTIONS] uid

Example: `mlrun logs ba409c0cb4904d60aa8f8d1c05b40a75 --project getting-started-admin`

Flag	Description
-p, project TEXT	Project name
offset INTEGER	Retrieve partial log, get up to size bytes starting at the offset from beginning of log
db TEXT	API service url
-w, watch	Retrieve logs of a running process, and watch the progress of the execution until it completes. Prints out the logs and continues to periodically poll for, and print, new logs as long as the state of the runtime that generates this log is either `pending` or `running`.

project

Use the `project` CLI command to load and/or run a project.

Usage: mlrun project [OPTIONS] [CONTEXT]

Example: `mlrun project -r workflow.py .`

Flag	Description
-n, name TEXT	Project name
-u, url TEXT	Remote git or archive url of the project
-r, run TEXT	Run workflow name of .py file
-a, arguments TEXT	Kubeflow pipeline arguments name and value tuples (with -r flag), e.g. -a x=6
-p, artifact_path TEXT	Target path/url for workflow artifacts, The string '{{workflow.uid}}' is replaced by workflow id
-x, param TEXT	mlrun project parameter name and value tuples, e.g. -p x=37 -p y='text'
-s, secrets TEXT	Secrets file= or env=ENV_KEY1,..
namespace TEXT	k8s namespace
db TEXT	API service url
init_git	For new projects init git the context dir
-c, clone	Force override/clone into the context dir
sync	Sync functions into db
-w, watch	Wait for pipeline completion (with -r flag)
-d, dirty	Allow run with uncommitted git changes
git_repo TEXT	git repo (org/repo) for git comments
git_issue INTEGER	git issue number for git comments
handler TEXT	Workflow function handler name
engine TEXT	Workflow engine (kfp/local)
local	Try to run workflow functions locally

run

Use the `run` CLI command to execute a task and inject parameters by using a local or remote function.

Usage: mlrun [OPTIONS] URL [ARGS]...

Examples:

- `mlrun run -f db://getting-started-admin/prep-data --project getting-started-admin`
- `mlrun run -f myfunc.yaml -w -p p1=3`

Flag	Description
-p, param TEXT	Parameter name and value tuples; for example, `-p x=37 -p y='text'`
-i, inputs TEXT	Input artifact; for example, `-i infile.txt=s3://mybucket/infile.txt`
in-path TEXT	Base directory path/URL for storing input artifacts
out-path TEXT	Base directory path/URL for storing output artifacts
-s, secrets TEXT	Secrets, either as `file=<filename>` or `env=<ENVAR>,...`; for example, `-s file=secrets.txt`
name TEXT	Run name
project TEXT	Project name or ID
-f, func-url TEXT	Path/URL of a YAML function-configuration file, or db:///[:tag] for a DB function object
task TEXT	Path/URL of a YAML task-configuration file
handler TEXT	Invoke the function handler inside the code file

version

Use the `version` CLI command to get the mlrun server version.

The watch Command

Use the `watch` CLI command to read the current or previous task (pod) logs.

Usage: mlrun watch [OPTIONS] POD

Example: `mlrun watch prep-data-6rf7b`

Flag	Description
-n, namespace	kubernetes namespace
-t, timeout	Timeout in seconds

watch-stream

Use the `watch-stream` CLI command to watch a v3io stream and print data at a recurring interval.

Usage: mlrun watch-stream [OPTIONS] URL

Examples:

- `mlrun watch-stream v3io:///users/my-test-stream`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1 -s 2`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1 -s 2 --seek EARLIEST`

Flag	Description
-s, shard-ids	Shard id to listen on (can be multiple).
–seek TEXT	Where to start/seek (EARLIEST or LATEST)
-i, interval	Interval in seconds. Default = 3
-j, is-json	Indicates that the payload is json (will be deserialized).

Building and running a function from a Git repository

To build and run a function from a Git repository, start out by adding a YAML function-configuration file in your local environment. This file should describe the function and define its specification. For example, create a **myfunc.yaml** file with the following content in your working directory:

```
kind: job
metadata:
  name: remote-demo1
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image: .mlrun/func-default-remote-demo-ps-latest
  image_pull_policy: Always
  build:
    base_image: mlrun/mlrun:1.0.0
    source: git://github.com/mlrun/mlrun
```

Then, run the following CLI command and pass the path to your local function-configuration file as an argument to build the function's container image according to the configured requirements. For example, the following command builds the function using the **myfunc.yaml** file from the current directory:

```
mlrun build myfunc.yaml
```

When the build completes, you can use the `run` CLI command to run the function. Set the `-f` option to the path to the local function-configuration file, and pass the relevant parameters. For example:

```
mlrun run -f myfunc.yaml -w -p p1=3
```

You can also try the following function-configuration example, which is based on the MLRun CI demo:

```
kind: job
metadata:
  name: remote-git-test
  project: default
  tag: latest
spec:
  command: 'myfunc.py'
  args: []
  image_pull_policy: Always
  build:
    commands: []
    base_image: mlrun/mlrun:1.0.0
    source: git://github.com/mlrun/ci-demo.git
```

Using a sources archive

The `-a|--archive` option of the CLI *`build`* command enables you to define a remote object path for storing TAR archive files with all the required code dependencies. The remote location can be, for example, in an AWS S3 bucket or in a data container in an Iguazio MLOps Platform ("platform") cluster. Alternatively, you can also set the archive path by using the `MLRUN_DEFAULT_ARCHIVE` environment variable. When an archive path is provided, the remote builder archives the configured function sources (see the `-s|-source` *`build`* option) into a TAR archive file, and then extracts (untars) all of the archive files (i.e., the function sources) into the configured archive location.

To use the archive option, first create a local function-configuration file. For example, you can create a **function.yaml** file in your working directory with the following content; the specification describes the environment to use, defines a Python base image, adds several packages, and defines **examples/training.py** as the application to execute on `run` commands:

```
kind: job
metadata:
  name: remote-demo4
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image_pull_policy: Always
  build:
    commands: []
    base_image: mlrun/mlrun:1.0.0
```

Next, run the following MLRun CLI command to build the function; replace the `<...>` placeholders to match your configuration:

```
mlrun build <function-configuration file path> -a <archive path/URL> [-s <function-
↪sources path/URL>]
```

For example, the following command uses the **function.yaml** configuration file (`.`), relies on the default function-sources path (`./`), and sets the target archive path to `v3io:///users/$V3IO_USERNAME/tars`. So, for a user named "admin", for example, the function sources from the local working directory will be archived and then extracted into an **admin/tars** directory in the "users" data container of the configured platform cluster (which is accessed via the `v3io` data mount):

```
mlrun build . -a v3io:///users/$V3IO_USERNAME/tars
```

Note:

- `.` is a shorthand for a **function.yaml** configuration file in the local working directory.
- The `-a|--archive` option is used to instruct MLRun to create an archive file from the function-code sources at the location specified by the `-s|--sources` option; the default sources location is the current directory (`./`).

After the function build completes, you can run the function with some parameters. For example:

```
mlrun run -f . -w -p p1=3
```

2.1.24 Examples

MLRun has many code examples and tutorial Jupyter notebooks with embedded documentation, ranging from examples of basic tasks to full end-to-end use-case applications, including the following. Some of the examples are found in other mlrun GitHub repositories.

- Learn MLRun basics — **mlrun_basics.ipynb**
- Convert local runs to Kubernetes jobs and create automated pipelines in a single notebook — **mlrun_jobs.ipynb**
- End-to-end ML pipeline— **demos/scikit-learn**, including:
  - Data ingestion and analysis
  - Model training
  - Verification
  - Model deployment
- MLRun with scale-out runtimes —
  - Distributed TensorFlow with Horovod and MPIJob, including data collection and labeling, model training and serving, and implementation of an automated workflow — **demos/image-classification-with-distributed-training**
  - Serverless model serving with Nuclio — **xgb_serving.ipynb**
  - Dask — **mlrun_dask.ipynb**
  - Spark — **mlrun_sparkk8s.ipynb**
- MLRun project and Git life cycle —
  - Load a project from a remote Git location and run pipelines — **load-project.ipynb**
  - Create a new project, functions, and pipelines, and upload to Git — **new-project.ipynb**
- Import and export functions using files or Git — **mlrun_export_import.ipynb**
- Query the MLRun DB — **mlrun_db.ipynb**

Additional Examples

- Additional end-to-end use-case applications — mlrun/demos repo
- MLRun functions Library — mlrun/functions repo

Back to top

PYTHON MODULE INDEX

m

s

INDEX

Symbols

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W