**Serving**

**Source:** https://docs.mlrun.org/en/latest/serving/serving-graph.html#simple-model-serving-router

**Web-Location:** (Documentation > External Add Ons > Tools for Serving > MLRun Serving)

<https://www.kubeflow.org/docs/external-add-ons/serving/>

**Content and Format:**

MLRun Serving

Model serving with MLRun

[MLRun serving graphs](https://docs.mlrun.org/en/latest/serving/serving-graph.html) allow you to build real-time data processing and advanced model serving pipelines and deploy them quickly to production with minimal effort.

The serving graphs can be composed of pre-defined graph blocks or of native python classes/functions. Graphs can auto-scale and span multiple function containers.

Graphs can run inside your IDE or Notebook for test and simulation and can be deployed into production serverless pipeline with a single command.

Accelerate performance and time to production[¶](https://docs.mlrun.org/en/latest/serving/serving-graph.html#accelerate-performance-and-time-to-production)

MLRun's underline serverless engine ([Nuclio](https://nuclio.io/" \l "about)) uses a high-performance parallel processing engine that maximizes the utilization of CPUs and GPUs.   
  
MLRun allows developers to focus on code and deploy faster by supporting:

* 13 protocols and invocation methods (HTTP, Cron, Kafka, Kinesis, etc....),
* Dynamic auto-scaling for http and streaming,
* Full life cycle: Including auto-generation of micro-services, APIs, load-balancing, logging, monitoring, and configuration management.

### Further Documentation[¶](https://docs.mlrun.org/en/latest/serving/serving-graph.html#in-this-document)

* **[Examples](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "examples)**
  + **[Simple model serving router](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "simple-model-serving-router)**
  + **[Advanced data processing and serving ensemble](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "advanced-data-processing-and-serving-ensemble)**
  + **[NLP processing pipeline with real-time streaming](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "nlp-processing-pipeline-with-real-time-streaming)**
* **[The Graph State Machine](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "the-graph-state-machine)**
  + **[Graph overview and usage](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "graph-overview-and-usage)**
  + **[Graph context and Event objects](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "graph-context-and-event-objects)**
  + **[Error handling and catchers](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "error-handling-and-catchers)**
  + **[Implement your own task class or function](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "implement-your-own-task-class-or-function)**
  + **[Building distributed graphs](https://docs.mlrun.org/en/latest/serving/serving-graph.html" \l "building-distributed-graphs)**

**Feature Store**

**Source:** https://docs.mlrun.org/en/latest/feature-store/feature-store.html

**Web-Location:** <https://www.kubeflow.org/docs/external-add-ons/feature-store/>

**Structure:** (Documentation > External Add Ons > Feature Store >

>Intro to Feature Stores \*  
> Introduction to Feast

(untouched)

>Getting Started with Feast

(untouched)

>MLRun\*

>Intro. to MLRun\*

>Training & Serving\*  
 >End-to-end Demo)\*

**Content and Formatting:**

Introduction to Feature Stores

An overview of feature stores in ML

A feature store functions as a single pane of glass for generating, sharing, and analyzing all available features across the organization. It is a repository where anyone on an ML team can easily find previously-created features. But a feature store is much more than just a repository. It is also a data transformation service where users can manipulate raw data and store it as features ready to be used by any machine learning model immediately. This service significantly reduces duplicate work, which accelerates time to production and reduces complexity.

Seamless feature engineering

Feature engineering is the process of generating new features (ie: comparing a person’s current spending to the average amount spent over the last 3 months). These features must be consistent for model training with historical data as well as model prediction with real-time data.

Feature consistency requires significant engineering effort and leads to model inaccuracy when not met. Additionally, monitoring solutions must be built to track features and results and send alerts of data or model drift.

Consider the spending example mentioned above. Creating such feature is easy if you have the full dataset in training. However, in serving, one needs to calculate this feature in an online/real-time manner.

The “brute-force” method is to have an ML engineer create an online pipeline that reimplements all the feature calculations done in the offline process. This is method is time-consuming, error-prone, and very difficult to maintain over time, leading to lengthy deployment time.

As you scale your MLOps, the “brute-force” method will require you to increase the number of Data Engineers and Data Scientists to deal with creating, implementing, and monitoring the necessary features. Large corporations that deal extensively with AI as part of their core value have built their own feature stores. Leaner teams need to find their own solution, and luckily the market for feature store solutions is starting mature.

A feature store creates an easy way to create in-training features that are deployable to serving without defining all the “glue” code.

Feature stores are compromised of the following:

* **Features**—An individual measurable property or characteristic of a phenomenon being observed. This may be raw data (e.g., transaction amount, image pixel, etc.) or a derivation from one or more features (e.g., deviation from average, pattern on image, etc.).
* [**Feature set**](https://docs.mlrun.org/en/latest/feature-store/feature-sets.html)— A grouping of features. The grouping is done by setting the entity key or set-of keys (e.g., a transaction may be grouped by userID performing the transfer or by the deviceID used to perform the transaction). You can ingest data to a feature set.
* **Execution graph** — A set of operations performed on the data during ingestion. The graph contains steps which represent data sources, targets, data transformations ,and data enrichment.
* [**Feature vector**](https://docs.mlrun.org/en/latest/feature-store/feature-vectors.html)— A set of features, taken from one or more feature sets, defined prior to model training. Feature vectors serve as the input to the model training process. During model serving, the feature values in the vector are obtained from an online service

MLRun Feature Store

Serverless real-time + offline feature store with scalable and high-performance data transformation layer

The common flow to work with MLRun’s feature store is:

1. Define the features, transformations, and validation logic
2. Ingest data (Dataframe, Kubernetes, [Nuclio](https://nuclio.io/))
3. Create feature vectors for training
4. Serve models

Define Features, Transformations, and Validation Logic

MLRun introduces a robust transformation engine lets you perform complex operations in 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.

Ingest Data

MLRun allows you to ingest data directly from a DataFrame by calling the feature set ingest method.

You can 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 would like to schedule the job.

MLRun can also leverage [Nuclio](https://nuclio.io/) to perform real-time ingestion by calling the deploy\_ingestion\_service function. This function allows you to read and update feature values (e.g., you can update a sliding window aggregation as part of a model serving process).

Train Models

Define the feature vector by calling the get\_offline\_features function to join features across different feature sets.

Input 'store://feature-vectors/{project}/{feature\_vector\_name}' to train a model with the feature vector data.

Serve Models

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. This will return a feature service object.

In the class preprocess method, call the feature service get method to get the values of those features.

Further Documentation

* [MLRun Documentation](https://docs.mlrun.org/en/latest/feature-store/feature-store.html)
* Training & Serving
* Feature Store End-to-End Demo

Training & Serving

Use MLRun to train and serve your ML models

When working on a new model we usually care about the experiment’s reproducibility as well as ease of feature and model environment recreation for the serving task. MLRun’s feature store enables us to do all that in a simple and automated fashion.

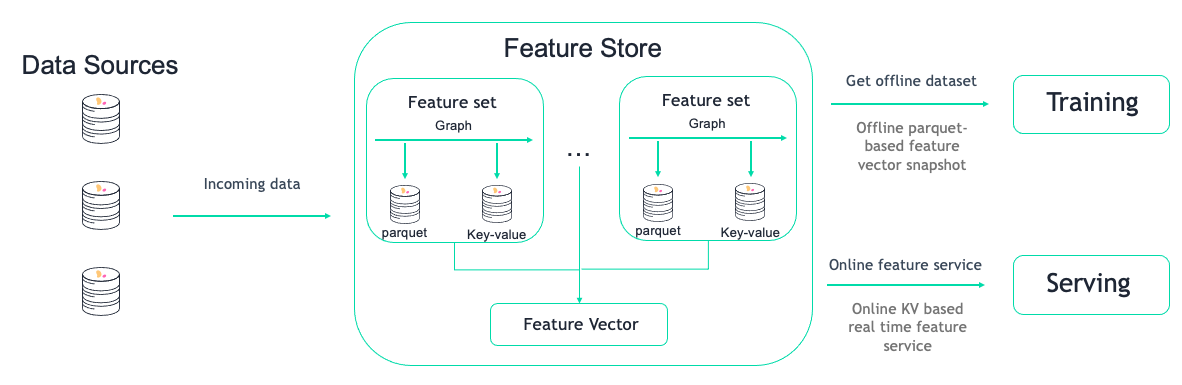
After defining our [feature sets](https://docs.mlrun.org/en/latest/feature-store/transformations.html) and proposed a [feature vector](https://docs.mlrun.org/en/latest/feature-store/feature-vectors.html) for the experiment, the feature store will enable us to automatically extract a versioned **offline** static dataset based on the parquet target defined in the feature sets for training.

For serving, once we validated the feature vector, we will use the **online** feature service based on the **nosql** target defined in the feature set for real-time serving.

Using this feature-store-centric process and one computation graph definition for a feature set, we receive an automatic online and offline implementation for our feature vectors.

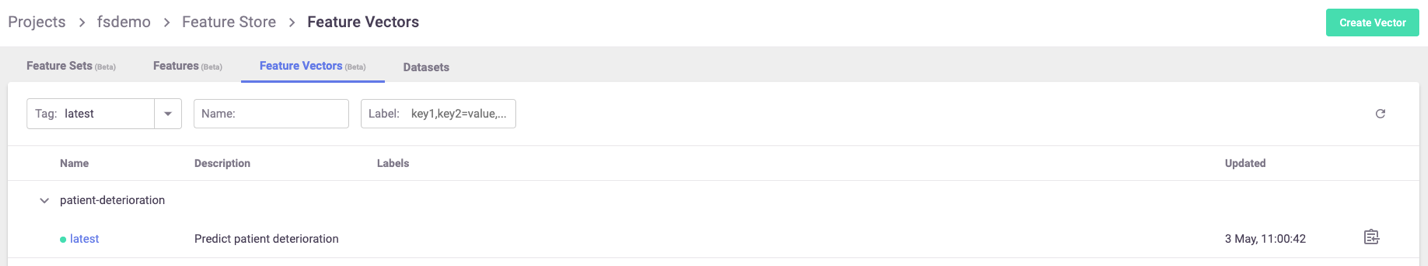
These feature vectors hold data versioned both in terms of the actual graph that was used to calculate each data point and the offline datasets that were created to train each model.

## How the solution should look like



## Creating an offline dataset

An “offline” dataset is a specific instance of our feature vector definition. To create this instance, use the feature store’s get\_offline\_features(<feature\_vector>, <target>) function on our 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() will run a local or Kubernetes job (can be specific by the run\_configparameter) to retrieve all the relevant data from the feature sets, merge them, and return it to the specified target. Targets can be a local parquet, AZ Blob store or any other type of available storage.

Once instantiated with a target, the feature vector will hold a reference to the instantiated dataset and will reference that as it’s 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 simple task. Keep reading to explore how to retrieve the offline dataset for EDA and in your training function.

To simply retrieve a feature vector’s offline dataset, you can us MLRun’s DataItem mechanism, simply referencing the feature vector and asking 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 simply 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 we can create our MLRun function and run it locally or over the Kubernetes cluster

*# Creating the training MLRun function with our code*

fn **=** mlrun**.**code\_to\_function**(**'training'**,**

kind**=**'job'**,**

handler**=**'my\_training\_function'**)**

*# Creating the task to run our funciton with our 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*

*# Running the function over the kubernetes cluster*

fn**.**run**(**task**)** *# Set local=True to run locally*

## 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 we will first create an online feature service with our feature vector.

**import** mlrun.feature\_store **as** fstore

svc **=** fstore**.**get\_online\_feature\_service**(<**feature vector name**>)**

After creating the service, use the feature vector’s Entity to get the latest feature vector for it. You can 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 serving model

MLRun enables you to easily serve your models using our [model server](https://docs.mlrun.org/en/latest/serving/serving-graph.html) ([example](https://github.com/mlrun/functions/blob/master/v2_model_server/v2_model_server.ipynb)). It enables you to define a serving model class and the computational graph required to run your entire prediction pipeline and deploy it as serverless functions using [Nuclio](https://github.com/nuclio/nuclio).

To embed the online feature service in your model server, all you need to do is 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 our [function marketplace](https://github.com/mlrun/functions) or write your own. As example of a scikit-learn based model server (taken from our 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'**)**

*# 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 our 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 we can deploy with:

*# Create the serving function from our 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 our 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**>})**

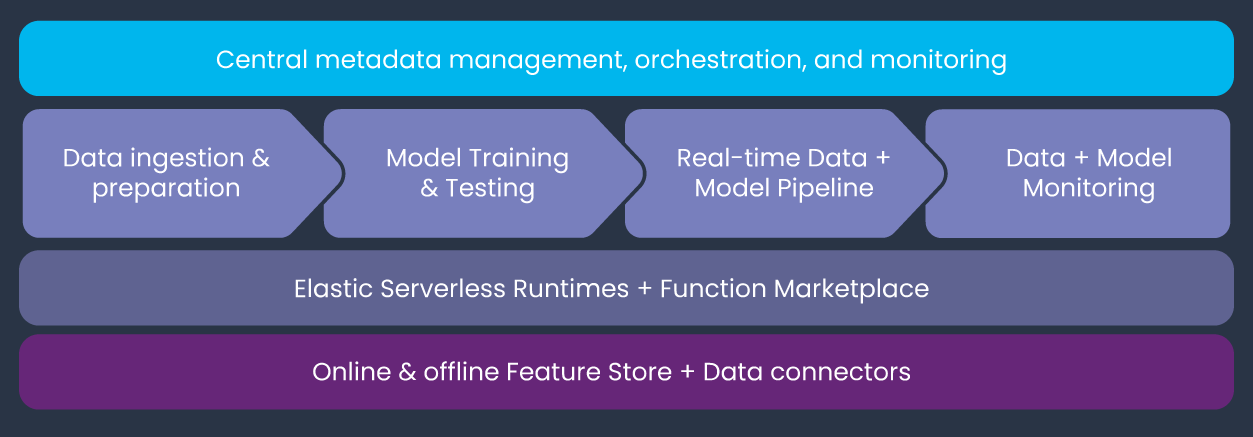
# Feature Store End-to-End Demo

Demo: How to use the MLRun feature store step-by-step

This demo will showcase:

* [**Data ingestion & preparation**](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/01-ingest-datasources.html)
* [**Model training & testing**](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/02-create-training-model.html)
* [**Real-time data & model pipeline**](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/03-deploy-serving-model.html)

These steps are the first key steps in the MLRun architecture:



Patient monitoring example

In this demo we will learn how to **Ingest** different data sources to our **Feature Store**.

Healthcare facilities need to closely monitor their patients and identify early indicators for medical intervention. Time is a key factor, the earlier the medical teams can attend to an issue, the better the outcome. Therefore, an effective system with real-time alerts can save lives.

We will use encrypted COVID-19 treatment data from patients prior to their condition becoming severe or critical. Our medical dataset will include three types of data:

* **Healthcare systems**: Batch updated dataset, containing different lab test results (e.g., Blood test results).
* **Patient records**: Static dataset containing general patient details.
* **Real-time sensors**: Real-time patient metric monitoring sensor.

Timeline

Description automatically generated with medium confidence

We will walk through creation of ingestion pipeline for each data source with all the needed preprocessing and validation. We will run the pipeline locally within a notebook and then launch a real-time function to **ingest live data** or schedule a cron to run the task when needed.

Following the ingestion, we will create a feature vector and train several models using this vector. We will then deploy the model and showcase the feature vector and model serving.   
  
Visit the following links to see the step-by-step walk-through of this example:

* [Part 1: Data Ingestion](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/01-ingest-datasources.html)
* [Part 2: Training](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/02-create-training-model.html)
* [Part 3: Serving](https://docs.mlrun.org/en/latest/feature-store/end-to-end-demo/03-deploy-serving-model.html)