

---

# Kedro Vertex AI Plugin

*Release 0.9.0*

**GetInData**

**May 15, 2023**



**CONTENTS:**

- 1 Introduction 1**
  - 1.1 What is GCP VertexAI Pipelines? . . . . . 1
  - 1.2 Why to integrate Kedro project with Vertex AI Pipelines? . . . . . 1
- 2 Installation 3**
  - 2.1 Installation guide . . . . . 3
  - 2.2 Configuration . . . . . 4
- 3 Getting started 9**
  - 3.1 Quickstart . . . . . 9
  - 3.2 GCP AI Platform support . . . . . 13
  - 3.3 Mlflow support . . . . . 15
  - 3.4 Continuous Deployment . . . . . 18
  - 3.5 Authenticating to Kubeflow Pipelines API . . . . . 19
- 4 Indices and tables 21**



## INTRODUCTION

### 1.1 What is GCP VertexAI Pipelines?

[Vertex AI Pipelines](#) is a Google Cloud Platform service that aims to deliver [Kubeflow Pipelines](#) functionality in a fully managed fashion. Vertex AI Pipelines helps you to automate, monitor, and govern your ML systems by orchestrating your ML workflow in a serverless manner.

### 1.2 Why to integrate Kedro project with Vertex AI Pipelines?

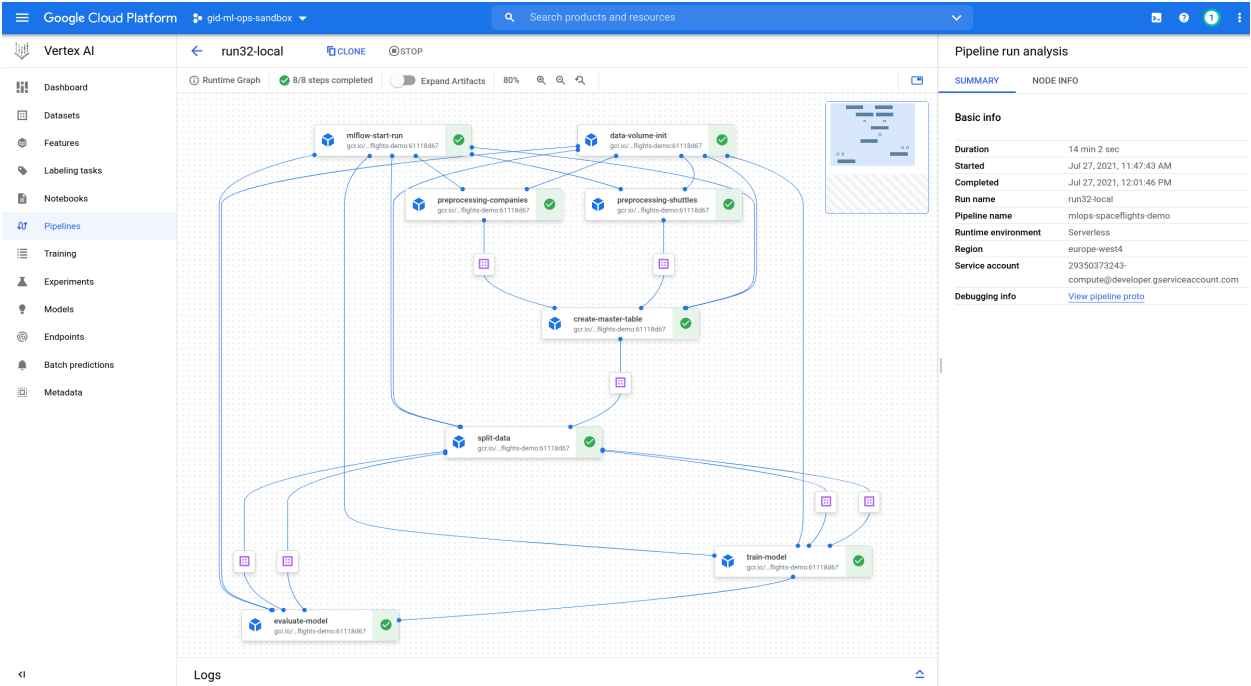
Throughout couple years of exploring ML Ops ecosystem as software developers we've been looking for a framework that enforces the best standards and practices regarding ML model development and Kedro Framework seems like a good fit for this position, but what happens next, once you've got the code ready?

It seems like the ecosystem grown up enough so you no longer need to release models you've trained with Jupyter notebook on your local machine on Sunday evening. In fact there are many tools now you can use to have an elegant model delivery pipeline that is automated, reliable and in some cases can give you a resource boost that's often crucial when handling complex models or a load of training data. With the help of some plugins **You can develop your ML training code with Kedro and execute it using multiple robust services** without changing the business logic.

We currently support:

- Kubeflow [kedro-kubeflow](#)
- Airflow on Kubernetes [kedro-airflow-k8s](#)

And with this **kedro-vertexai** plugin, you can run your code on GCP Vertex AI Pipelines in a fully managed fashion



## INSTALLATION

### 2.1 Installation guide

#### 2.1.1 Kedro setup

First, you need to install base Kedro package

```
$ pip install "kedro>=0.18.1,<0.19.0"
```

#### 2.1.2 Plugin installation

##### Install from PyPI

You can install `kedro-vertexai` plugin from PyPi with `pip`:

```
pip install --upgrade kedro-vertexai
```

##### Install from sources

You may want to install the develop branch which has unreleased features:

```
pip install git+https://github.com/getindata/kedro-vertexai.git@develop
```

#### 2.1.3 Available commands

You can check available commands by going into project directory and running:

```
$ kedro vertexai
Usage: kedro vertexai [OPTIONS] COMMAND [ARGS]...

  Interact with Google Cloud Platform :: Vertex AI Pipelines

Options:
  -e, --env TEXT  Environment to use.
  -h, --help      Show this message and exit.

Commands:
```

(continues on next page)

(continued from previous page)

<code>compile</code>	Translates Kedro pipeline into JSON file with VertexAI...
<code>init</code>	Initializes configuration for the plugin
<code>list-pipelines</code>	List deployed pipeline definitions
<code>run-once</code>	Deploy pipeline as a single run within given experiment...
<code>schedule</code>	Schedules recurring execution of latest version of the...
<code>ui</code>	Open VertexAI Pipelines UI in new browser tab

## init

`init` command takes two arguments: `PROJECT_ID` and `REGION`. This command generates a sample configuration file in `conf/base/vertexai.yaml`. The YAML file content is described in the [Configuration section](#).

## ui

`ui` command opens a web browser pointing to the currently configured Vertex AI Pipelines UI on GCP web console.

## list-pipelines

`list-pipelines` uses Vertex AI API to retrieve list of all pipelines

## compile

`compile` transforms Kedro pipeline into Vertex AI workflow. The resulting `json` file can be uploaded to Vertex AI Pipelines via [Python Client](#) e.g. from your CI/CD job.

## run-once

`run-once` is all-in-one command to compile the pipeline and run it in the GCP Vertex AI Pipelines environment.

# 2.2 Configuration

Plugin maintains the configuration in the `conf/base/vertexai.yaml` file. Sample configuration can be generated using `kedro vertexai init`:

```
# Configuration used to run the pipeline
project_id: my-gcp-mlops-project
region: europe-west1
run_config:
  # Name of the image to run as the pipeline steps
  image: eu.gcr.io/my-gcp-mlops-project/example_model:${commit_id}

  # Pull policy to be used for the steps. Use Always if you push the images
  # on the same tag, or Never if you use only local images
  image_pull_policy: IfNotPresent

  # Location of Vertex AI GCS root
```

(continues on next page)



(continued from previous page)

```

root: bucket_name/gcs_suffix

# Name of the kubeflow experiment to be created
experiment_name: MyExperiment

# Name of the scheduled run, templated with the schedule parameters
scheduled_run_name: MyExperimentRun

# Optional pipeline description
#description: "Very Important Pipeline"

# How long to keep underlying Argo workflow (together with pods and data
# volume after pipeline finishes) [in seconds]. Default: 1 week
ttl: 604800

# What Kedro pipeline should be run as the last step regardless of the
# pipeline status. Used to send notifications or raise the alerts
# on_exit_pipeline: notify_via_slack

# Optional section allowing adjustment of the resources
# reservations and limits for the nodes
resources:

# For nodes that require more RAM you can increase the "memory"
data-import-node:
  memory: 2Gi

# Training nodes can utilize more than one CPU if the algorithm
# supports it
model-training-node:
  cpu: 8
  memory: 60Gi

# GPU-capable nodes can request 1 GPU slot
tensorflow-node:
  gpu: 1

# Resources can be also configured via nodes tag
# (if there is node name and tag configuration for the same
# resource, tag configuration is overwritten with node one)
gpu_node_tag:
  cpu: 1
  gpu: 2

# Default settings for the nodes
__default__:
  cpu: 200m
  memory: 64Mi

# Optional section allowing to configure node selectors constraints
# like gpu accelerator for nodes with gpu resources.
# (Note that not all accelerators are available in all

```

(continues on next page)

(continued from previous page)

```

# regions - https://cloud.google.com/compute/docs/gpus/gpu-regions-zones)
# and not for all machines and resources configurations -
# https://cloud.google.com/vertex-ai/docs/training/configure-compute#specifying_gpus
node_selectors:
  gpu_node_tag:
    cloud.google.com/gke-accelerator: NVIDIA_TESLA_T4
  tensorflow-step:
    cloud.google.com/gke-accelerator: NVIDIA_TESLA_K80

# Optional section allowing to generate config files at runtime,
# useful e.g. when you need to obtain credentials dynamically and store them in
credentials.yaml
# but the credentials need to be refreshed per-node
# (which in case of Vertex AI would be a separate container / machine)
# Example:
# dynamic_config_providers:
#   - cls: kedro_vertexai.auth.gcp.MLFlowGoogleOAuthCredentialsProvider
#     params:
#       client_id: iam-client-id

dynamic_config_providers: []

```

## 2.2.1 Dynamic configuration support

The plugin relies on the project configuration and uses the same config loader that your project uses. For some cases, you need to modify the `settings.py` to work with our plugin. Follow the instructions below.

Every Kedro Vertex AI job gets injected with two environment variables:

- `KEDRO_CONFIG_JOB_NAME` - contains name of the job from Vertex AI
- `KEDRO_CONFIG_RUN_ID` - contains unique run identifier from Vertex AI

You can consume them as you like or use them within config loaders.

### Using OmegaConfigLoader

`kedro-vertexai` supports `OmegaConfigLoader`. In order to configure it, update the `settings.py` file in your Kedro project as follows:

```

from kedro.config import OmegaConfigLoader
CONFIG_LOADER_CLASS = OmegaConfigLoader
CONFIG_LOADER_ARGS = {
    # other args
    "config_patterns": {"vertexai": ["vertexai*"]}
}

```

Follow Kedro's official documentation, to see how to add templating, custom resolvers etc. ([https://docs.kedro.org/en/stable/configuration/advanced\\_configuration.html#how-to-do-templating-with-the-omegaconfigloader](https://docs.kedro.org/en/stable/configuration/advanced_configuration.html#how-to-do-templating-with-the-omegaconfigloader))

## Using TemplatedConfigLoader

TemplatedConfigLoader allows passing environment variables to configuration files. It reads all environment variables following KEDRO\_CONFIG\_<NAME> pattern, which you can later inject in configuration file using `${name}` syntax.

This feature is especially useful for keeping the executions of the pipelines isolated and traceable by dynamically setting output paths for intermediate data in the **Data Catalog**, e.g.

```
# ...
train_x:
  type: pandas.CSVDataSet
  filepath: gs://<bucket>/kedro-vertexai/${run_id}/05_model_input/train_x.csv

train_y:
  type: pandas.CSVDataSet
  filepath: gs://<bucket>/kedro-vertexai/${run_id}/05_model_input/train_y.csv
# ...
```

In this case, the `${run_id}` placeholder will be substituted by the unique run identifier from Vertex AI Pipelines.

## Dynamic config providers

When running the job in VertexAI it's possible to generate new configuration files **at runtime** if that's required, one example could be generating Kedro credentials on a Vertex AI's node level (the opposite would be supplying the credentials when starting the job).

Example:

```
run_config:
  # ...
  dynamic_config_providers:
    - cls: <fully qualified class name inheriting from kedro_vertexai.dynamic_config.
      ↪DynamicConfigProvider>
      params:
        # ... params passed to the constructor of the class
        abc: value1
        xyz: value2
```

The `cls` fields should contain a fully qualified reference to a class implementing abstract `kedro_vertexai.dynamic_config.DynamicConfigProvider`. All `params` will be passed as `kwargs` to the class's constructor. Two required methods are:

```
@property
def target_config_file(self) -> str:
    return "name-of-the-config-file.yml"

def generate_config(self) -> dict:
    return {
        "layout": {
            "of-the-target": {
                "config-file": "value"
            }
        }
    }
```

First one - `target_config_file` should return the name of the configuration file to be generated (e.g. `credentials.yml`) and the `generate_config` should return a dictionary, which will be then serialized into the target file as YAML. If the target file already exists during the invocation, it will be merged (see method `kedro_vertexai.dynamic_config.DynamicConfigProvider.merge_with_existing`) with the existing one and then saved again. Note that the `generate_config` has access to an initialized plugin config via `self.config` property, so any values from the `vertexai.yml` configuration is accessible.

## 2.2.2 Resources configuration

Optional `resources` and `node_selectors` sections enable adjustment of the resources reservations and limits for the selected Kedro nodes. Settings for individual nodes, we can define in two ways - using the name of the node or its `tag` (if there is node name and tag configuration for the same resource, tag configuration is overwritten with node one). For example, with the `vertexai.yml` configuration file shown at the beginning of the chapter and the Kedro pipeline containing such a node:

```
def create_pipeline(**kwargs):
    return Pipeline(
        [
            node(
                func=train_model,
                inputs=["X_train", "y_train"],
                outputs="regressor",
                name="model_training_node",
                tags="gpu_node_tag",
            ),
        ]
    )
```

we expect this particular node to run with two `NVIDIA_TESLA_T4` GPUs, eight CPUs, and memory allocated according to the specified `60Gi` limit. (Note that not all accelerators are available in all `regions` and not for all `machines and resources configurations`)

## GETTING STARTED

### 3.1 Quickstart

#### 3.1.1 Prerequisites

The quickstart assumes user have access to Vertex AI Pipelines service.

#### 3.1.2 Install the toy project with Vertex AI Pipelines support

It is a good practice to start by creating a new virtualenv before installing new packages. Therefore, use `virtualenv` command to create new env and activate it:

```
$ virtualenv venv-demo
created virtual environment CPython3.8.12.final.0-64 in 764ms
  creator CPython3Posix(dest=/home/getindata/kedro/venv-demo, clear=False, no_vcs_
↳ ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle,
↳ via=copy)
    added seed packages: pip==22.0.4, setuptools==60.9.3, wheel==0.37.1
  activators BashActivator,CShellActivator,FishActivator,NushellActivator,
↳ PowerShellActivator,PythonActivator
$ source venv-demo/bin/activate
```

Then, `kedro` must be present to enable cloning the starter project, along with the latest version of `kedro-kubeflow` plugin and `kedro-docker` (required to build docker images with the Kedro pipeline nodes):

```
$ pip install 'kedro>=0.18.1,<0.19.0' kedro-vertexai kedro-docker
```

With the dependencies in place, let's create a new project:

```
$ kedro new --starter=spaceflights

Project Name:
=====
Please enter a human readable name for your new project.
Spaces and punctuation are allowed.
[New Kedro Project]: Vertex AI Plugin Demo

Repository Name:
=====
```

(continues on next page)

(continued from previous page)

Please enter a directory name for your new project repository.  
 Alphanumeric characters, hyphens and underscores are allowed.  
 Lowercase is recommended.  
 [vertex-ai-plugin-demo]:

Python Package Name:

=====

Please enter a valid Python package name for your project package.  
 Alphanumeric characters and underscores are allowed.  
 Lowercase is recommended. Package name must start with a letter  
 or underscore.

[vertex\_ai\_plugin\_demo]:

Change directory to the project generated in /Users/getindata/vertex-ai-plugin-demo

A best-practice setup includes initialising git and creating a virtual environment,  
 ↪ before running ``kedro install`` to install project-specific dependencies. Refer to,  
 ↪ the Kedro documentation: <https://kedro.readthedocs.io/>

Finally, go the demo project directory and ensure that kedro-vertexai plugin is activated:

```
$ cd vertexai-plugin-demo/
$ pip install -r src/requirements.txt
(...)
```

Requirements installed!

```
$ kedro vertexai --help
```

```
Usage: kedro vertexai [OPTIONS] COMMAND [ARGS]...
```

```
Interact with Google Cloud Platform :: Vertex AI Pipelines
```

Options:

```
-e, --env TEXT Environment to use.
-h, --help Show this message and exit.
```

Commands:

```
compile Translates Kedro pipeline into JSON file with VertexAI...
init      Initializes configuration for the plugin
list-pipelines List deployed pipeline definitions
run-once  Deploy pipeline as a single run within given experiment...
schedule  Schedules recurring execution of latest version of the...
ui        Open VertexAI Pipelines UI in new browser tab
```

### 3.1.3 Build the docker image to be used in Vertex AI Pipelines runs

First, initialize the project with `kedro-docker` configuration by running:

```
$ kedro docker init
```

This command creates a several files, including `.dockerignore`. This file ensures that transient files are not included in the docker image and it requires small adjustment. Open it in your favourite text editor and extend the section `# except the following` by adding there:

```
!data/01_raw
```

#### Ensure right requirements.txt

You need to make sure that before you build the docker image and submit the job to Vertex AI Pipelines, all of your project's Python package requirements are properly saved in `requirements.txt`, that includes **this plugin**. Ensure that the `src/requirements.txt` contains this line

```
kedro-vertexai
```

#### Adjusting Data Catalog to be compatible with Vertex AI

This change enforces raw input data existence in the image. While running locally, every intermediate dataset is stored as a `MemoryDataSet`. When running in VertexAI Pipelines, there is no shared-memory, Kedro-VertexAI plugin automatically handles intermediate dataset serialization - every intermediate dataset will be stored (as a compressed cloud-pickle file) in GCS bucket specified in the `vertexai.yml` config under `run_config.root` key. Adjusted `catalog.yml` should look like this (note: remove the rest of the entries which comes with the `spaceflights` starter - you need only `companies, reviews, shuttles`.)

```
companies:
  type: pandas.CSVDataSet
  filepath: data/01_raw/companies.csv
  layer: raw

reviews:
  type: pandas.CSVDataSet
  filepath: data/01_raw/reviews.csv
  layer: raw

shuttles:
  type: pandas.ExcelDataSet
  filepath: data/01_raw/shuttles.xlsx
  layer: raw
```

All intermediate and output data will be stored in the location with the following pattern:

```
gs://<run_config.root from vertexai.yml>/kedro-vertexai-temp/<vertex ai job name>/*.bin
```

Of course if you want to use intermediate/output data and store it a location of your choice, add it to the catalog. Be aware that you cannot use local paths - use `gs://` paths instead.

## Disable telemetry or ensure consent

Latest version of Kedro starters come with the `kedro-telemetry` installed, which by default prompts the user to allow or deny the data collection. Before submitting the job to Vertex AI Pipelines you have two options:

- allow the telemetry by setting `consent: true` in the `.telemetry` file in the project root directory
- disable telemetry by removing `kedro-telemetry` from the `src/requirements.txt`.

If you leave the `.telemetry` file with default `consent: false`, the pipeline will crash in runtime in Vertex AI, because `kedro-telemetry` will spawn an interactive prompt and ask for the permission to collect the data.

The usage of `${run_id}` is described in section *Dynamic configuration support*.

## Build the image

Execute:

```
kedro docker build --docker-args "--build-arg BASE_IMAGE=python:3.8-buster"
```

When execution finishes, your docker image is ready. If you don't use local cluster, you should push the image to the remote repository:

```
docker tag vertex-ai-plugin-demo:latest remote.repo.url.com/vertex-ai-plugin-demo:latest
docker push remote.repo.url.com/vertex-ai-plugin-demo:latest
```

### 3.1.4 Run the pipeline on Vertex AI

First, run `init` script to create the sample configuration. There are 2 parameters:

- `PROJECT_ID` which is ID of your Google Cloud Platform project - can be obtained from [GCP Console](#) or from command line (`gcloud config get-value project`)
- `REGION` - Google Cloud Platform region in which the Vertex AI pipelines should be executed (e.g. `europe-west1`).

```
kedro vertexai init <GCP PROJECT ID> <GCP REGION>
(...)
Configuration generated in /Users/getindata/vertex-ai-plugin-demo/conf/base/vertexai.yaml
```

Then adjust the `conf/base/vertexai.yaml`, especially:

- `image:` key should point to the full image name (like `remote.repo.url.com/vertex-ai-plugin-demo:latest` if you pushed the image at this name).
- `root:` key should point to the GCS bucket that will be used internally by Vertex AI and the plugin itself, e.g. `your_bucket_name/subfolder-for-vertexai`

Finally, everything is set to run the pipeline on Vertex AI Pipelines. Execute `run-once` command:

```
$ kedro vertexai run-once
2022-03-18 13:44:27,667 - kedro_vertexai.client - INFO - Generated pipeline definition_
↳ was saved to /var/folders/0b/mdxthmvd74x90fp84zl4mb5h00000gn/T/kedro-vertexai2jyrt89b.
↳ json
See the Pipeline job here: https://console.cloud.google.com/vertex-ai/locations/europe-
↳ west1/pipelines/runs/vertex-ai-plugin-demo-20220318124425?project=gid-ml-ops-sandbox
```



As you can see, the pipeline was compiled and started in Vertex AI Pipelines. When you visit the link shown in logs you can observe the running pipeline:

## 3.2 GCP AI Platform support

Google Cloud's AI Platform offers couple services that simplify Machine Learning tasks with use of Kubeflow based components.

### 3.2.1 Using kedro with AI Platform Notebooks

[AI Platform Notebooks](#) provides an easy way to manage and host JupyterLab based data science workbench environment. What we've found out is that the default images provided by a service cause some dependency conflicts. To avoid this issues make sure you use isolated virtual environment, e.g. [virtualenv](#). New virtual environment can be created by simply invoking `python -m virtualenv venv` command.

### 3.2.2 Using kedro-kubeflow with AI Platform Pipelines

[AI Platform Pipelines](#) is a service that allows to easily deploy [Kubeflow Pipelines](#) on new or existing Google Kubernetes Engine clusters.

In general `kedro-kubeflow` plugin should work with AI Platform Pipelines out of the box, with the only exception is that it requires authentication. Note that the `host` variable should point to a dashboard URL generated by AI Platform Pipelines service (e.g. <https://653hddae86eb7b0-dot-europe-west1.pipelines.googleusercontent.com/>), just open the dashboard from the [service page](#) and copy url from the browser.

Below is the list of authentication scenarios supported so far:

#### 1. Connecting to AI Pipelines from AI Platform Notebooks

In this scenario authentication works out of the box with *default credentials* mechanism.

#### 2. Authentication to AI Pipelines from local environment

To interact with AI Platform Pipelines from local environment you can use the mechanisms provided by [Google Cloud SDK](#). After installing the SDK run `google cloud application-default login` to initialize *default credentials* on your local machine.

You can use service account key for authentication as well. To make that work just set `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the path of where the service account key file is stored.

### 3. Authenticating through IAP Proxy

[Identity Aware Proxy](#) is a product that allows securing your cloud based applications with Google Identity.

To authenticate with IAP find out which *oauth client ID* is the proxy configured with and then save it in `IAP_CLIENT_ID` environment variable. The authentication should work seamlessly assuming identity you are using has been granted access to the application.

The above will work if you are connecting from within GCP VM or locally with specified service account credentials. It will *NOT* work for credentials obtained with `google cloud application-default login`.

### 3.2.3 Using kedro-kubeflow with Vertex AI Pipelines (EXPERIMENTAL)

[Vertex AI Pipelines](#) is a fully managed service that allows to easily deploy [Kubeflow Pipelines](#) on a serverless Google service. [Vertex AI Pipelines](#) was still in a Preview mode when this plugin version was released, therefore plugin capability is also limited.

#### 1. Preparing configuration

In order the plugin picks Vertex AI Pipelines as a target infrastructure, it has to be indicated in configuration. As the solution is serverless, no URL is to be provided. Instead, special set of parameters has to be passed, so that connection is established with proper GCP service.

```
host: vertex-ai-pipelines
project_id: hosting-project
region: europe-west4
run_config:
  root: vertex-ai-pipelines-accessible-gcs-bucket/pipelines-specific-path
```

If the pipeline requires access to services that are not exposed to public internet, you need to configure [VPC peering between Vertex internal network and VPC that hosts the internal service](#) and then set the VPC identifier in the configuration. Optionally, you can add custom host aliases:

```
run_config:
  vertex_ai_networking:
    vpc: projects/12345/global/networks/name-of-vpc
    host_aliases:
      - ip: 10.10.10.10
        hostnames: ['mlflow.internal']
      - ip: 10.10.20.20
        hostnames: ['featurestore.internal']
```

#### 2. Preparing environment variables

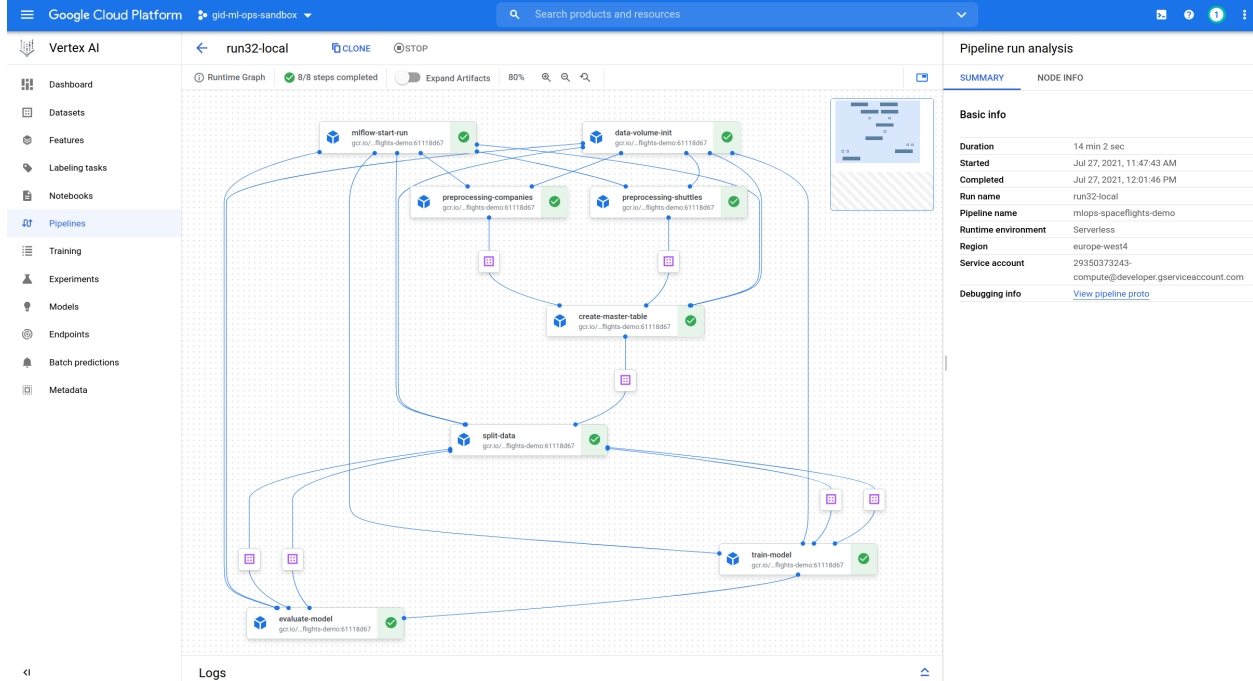
There are the following specific environment variables required for the pipeline to run correctly:

- `SERVICE_ACCOUNT` - full email of service account that job will use to run the pipeline. Account has to have access to `run_config.root` path. Variable is optional, if no given, project compute account is used
- `MLFLOW_TRACKING_TOKEN` - identity token required if MLFlow is used inside the project and mlflow access is protected. Token is passed as it is to kedro nodes in order to authenticate against MLFlow service. Can be generated via `gcloud auth print-identity-token` command.

### 3. Supported commands

Following commands are supported:

```
kedro kubeflow compile
kedro kubeflow run-once
kedro kubeflow schedule
kedro kubeflow list-pipelines
```



### 3.3 Mlflow support

If you use [MLflow](#) and [kedro-mlflow](#) for the Kedro pipeline runs monitoring, the plugin will automatically enable support for:

- starting the experiment when the pipeline starts,
- logging all the parameters, tags, metrics and artifacts under unified MLFlow run.

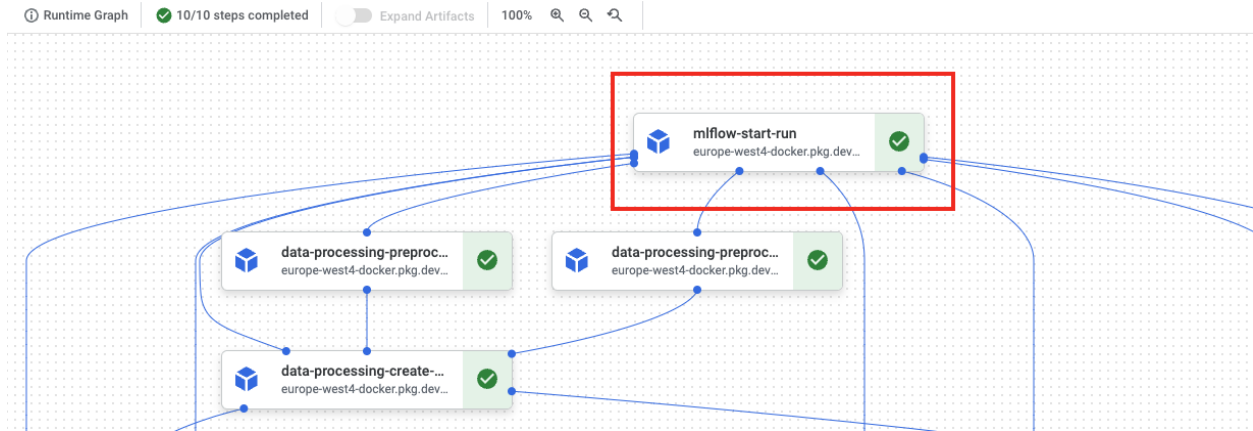
To make sure that the plugin discovery mechanism works, add `kedro-mlflow` and `kedro-vertexai` as a dependencies to `src/requirements.txt` and run:

```
$ pip install -r src/requirements.txt
$ kedro mlflow init
```

Then, adjust the `kedro-mlflow` configuration and point to the `mlflow` server by editing `conf/base/mlflow.yml` and adjusting `server.mlflow_tracking_uri` key. Then, build the image:

```
$ kedro docker build
```

And re-push the image to the remote registry. And verify how does it look in the Kubeflow UI. You should notice `mlflow-start-run` step on the very top.



Finally, start the pipeline. While it executes, the new Mlflow run is started and it's constantly updated with the attributes provided by the next steps. Finally, the experiments runs page looks like:

	Start Time	Parameters	Tags	node_names
<input type="checkbox"/>	2021-01-12 14:03:12	{'test_size': 0.3, 'random_state': 42}	kubeflow_run_id: e8f67fcc-159f-4091-a60c-98d237610117	('evaluate_model([X_test, regressor.y_test]) -> None')
<input type="checkbox"/>	2021-01-12 14:01:25	{'test_size': 0.2, 'random_state': 3}	kubeflow_run_id: 8a27b1ad-57ba-446f-b899-d35884666d66	('train_model([X_train, y_train]) -> [regressor]')
<input type="checkbox"/>	2021-01-12 13:59:32	{'test_size': 0.2, 'random_state': 3}	kubeflow_run_id: 21d2baac-d5a0-4ec0-aa2f-e0a10510aaf3	('evaluate_model([X_test, regressor.y_test]) -> None')
<input type="checkbox"/>	2021-01-12 13:54:20	{'test_size': 0.2, 'random_state': 3}	kubeflow_run_id: 9211a78f-ef32-4b01-b66f-f9721cc709e2	('evaluate_model([X_test, regressor.y_test]) -> None')

The UI presents the pipeline status (in form of the icon) and latest node that was run (for failed runs in indicates at what step did the pipeline fail). Also, the `vertexai_run_id` and `vertexai_job_name` tags can be used to correlate Mlflow run with the Vertex AI pipeline execution.

### 3.3.1 Authorization

For MLflow deployments that are secured with some authorization mechanism, the requests being made need to (usually) have the `Authorization` header set. MLflow allows to plug-in custom headers via `request_header_provider` entry point. We rely on [official kedro-mlflow approach](#). We're providing 2 implementations of the `Authorization` header provider, which obtain ID token from Google's endpoints - either OAuth or IAM. Of course, you can implement your own authorization mechanism by inheriting from the `kedro_vertexai.auth.mlflow_request_header_provider.RequestHeaderProviderWithKedroContext` class.

### Authorization with a service account email and OAuth Client ID (IAM)

Works well with Identity-Aware-Proxy deployments of MLflow, such as [MLflow on App Engine](#).

In the `mlflow.yml`

```
server:
    # ... rest of the config
    request_header_provider:
        type: kedro_vertexai.auth.gcp.MLflowGoogleIAMRequestHeaderProvider
        pass_context: true
        init_kwargs:
            client_id: <OAuth Client ID>
            service_account: <service account email>
```

### OAuth2.0 based authorization

In the `mlflow.yml`

```
server:
    # ... rest of the config
    request_header_provider:
        type: kedro_vertexai.auth.gcp.MLflowGoogleOAuthRequestHeaderProvider
        pass_context: true
        init_kwargs:
            client_id: <OAuth Client ID>
```

In the `vertexai.yml`

```
run_config:
    # ... rest of the config
    mlflow:
        request_header_provider_params:
            client_id: <OAuth Client ID>
```

### Custom authorization method

You can inherit from `kedro_vertexai.auth.mlflow_request_header_provider.RequestHeaderProviderWithKedroContext` class and extend it with your own authorization method. For example, if you want to use a custom header, you can do it like this:

```
from kedro_vertexai.auth.mlflow_request_header_provider import
↳ RequestHeaderProviderWithKedroContext
from cachetools import cached, TTLCache

class MyCustomMLflowHeaderProvider(RequestHeaderProviderWithKedroContext):
    def in_context(self):
        # here, self.params will contain all values from the mlflow.yml:server.request_
↳ header_provider.init_kwargs dictionary
        return "my_auth_info" in self.params

    @cached(TTLCache(1, ttl=3600)) # it's a good practice to cache the token for some time
```

(continues on next page)

(continued from previous page)

```
def request_headers(self):
    token = obtain_your_token_here()
    return {"Authorization": f"Bearer {token}", "X-My-Custom-Header": self.params[
↪ "my_auth_info"]}
```

In the `mlflow.yml`

```
server:
    # ... rest of the config
    request_header_provider:
        type: path.to.your.MyCustomMLflowHeaderProvider
        pass_context: true # if you want to pass context. it must be named ``kedro_
↪ context`` in the ``__init__`` method of your custom ``request_header_provider``
        init_kwargs:
            my_kwarg: 1
```

## 3.4 Continuous Deployment

With kedro pipelines started on the remote Kubeflow Pipelines clusters, changes in the code require re-building docker images and (sometimes) changing the pipeline structure. To simplify this workflow, Kedro-kubeflow plugin is capable of creating configuration for the most popular CI/CD automation tools.

The auto generated configuration defines these actions:

- on any new push to the repository - image is re-built and the pipeline is started using `run-once`,
- on merge to master - image is re-built, the pipeline is registered in the Pipelines and scheduled to execute on the daily basis.

The behaviour and parameters (like schedule expression) can be adjusted by editing the generated files. The configuration assumes that Google Container Registry is used to store the images, but users can freely adapt it to any (private or public) docker images registry.

### 3.4.1 Github Actions

If the Kedro project is stored on github (either in private or public repository), Github Actions can be used to automate the Continuous Deployment. To configure the repository, go to Settings->Secrets and add there:

- `GKE_PROJECT`: ID of the google project.
- `GKE_SA_KEY`: service account key, encoded with base64 (this service account must have access to push images into registry),
- `IAP_CLIENT_ID`: id of the IAP proxy client to communicate with rest APIs.

Next, re-configure the project using

```
kedro kubeflow init --with-github-actions https://<endpoint_name>.endpoints.<project-
↪ name>.cloud.google.com/pipelines
```

This command will generate Github Actions in `.github/workflows` directory. Then push the code to any branch and go to “Actions” tab in Github interface.

## 3.5 Authenticating to Kubeflow Pipelines API

Plugin supports 2 ways of authenticating to Kubeflow Pipelines API:

### 3.5.1 1. KFP behind IAP proxy on Google Cloud

It's already described in *GCP AI Platform support* chapter.

### 3.5.2 2. KFP behind Dex+authservice

Dex is the recommended authentication mechanism for on-premise Kubeflow clusters. The usual setup looks in a way that:

- `oidc-authservice` redirect unauthenticated users to Dex,
- `Dex` authenticates user in remote system, like LDAP or OpenID and also acts as OpenID provider,
- `oidc-authservice` asks Dex for token and creates the session used across entire Kubeflow.

In order to use `kedro-kubeflow` behind Dex-secured clusters, use the following manual:

1. Setup `staticPassword` authentication method and add a user that you're going to use as CI/CD account.
2. Point your Kedro project to `/pipeline` API on Kubeflow, for example: `https://kubeflow.local/pipeline`
3. Set environment variables `DEX_USERNAME` and `DEX_PASSWORD` before calling `kedro kubeflow`





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`