Create an ML Pipeline in Plexus using Apache Airflow

Contents

Get started
Homogeneous environment and heterogeneous
Run AI and ML applications on Plexus
Reference
Run HPC applications on Plexus
Virtual data center
Apache Airflow DAG pipelines

Create an ML Pipeline in Plexus using Apache Airflow

Apache Airflow is an open-source platform for designing, scheduling, and monitoring workflows. It was created by Airbnb in 2014 and officially announced in 2015. Workflows are built programmatically by writing a **Directed Acyclic Graph (DAG)** in Python, using the framework dynamic and extensible. It also contains a rich user interface for visualizing and troubleshooting. See **airflow.apache.org** to learn more about Airflow.

The pipeline design process

The process involves creating an Airflow project, defining the collection of tasks to be run, the Directed Acyclic Graph or 'dag', and packaging the project so it can be uploaded to Plexus.

Step 1: Create a local project directory for the Airflow application

Your local project contain your dag file and any other scripts needed to run the application. Below is the project directory structure for the example Visualizer in VS Code Explorer.

The example project is named IMDB where it uses the IMDB movies dataset. The .dag file and .bin directories are at the top level. .venv is a Python **The dag file is a python script that contains the code for the workflow**. .dagignore is an empty folder that is used to store tasks and test data. .vscode contains the python modules that process the data and train the model.

Step 2: Write the DAG

The **DAG is the collection of tasks and their dependencies**. This is defined in dag.py by instantiating a DAG object and passing it to the tasks. A task is an instance of an Airflow operator class that executes some action. In our example below, we have created a simple DAG object and the first task. Plexus has its own operator within Airflow that is derived from BaseOperator class. For more information on dags, operators, and other core Airflow concepts, see the reference documentation: **airflow.apache.org**.

```
# -*-coding:utf-8-*-
# Copyright (c) 2020-2021. HPE.
# Licensed under the MIT License.
# Wrapper script for launching data ingest jobs on Plexus.
# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG
# Operators; we need this to operate!
from airflow.bash_operator import BashOperator
from airflow.plexus.operators.launchjob import LaunchJobOperator, LaunchJobArrayOperator
# These args will get passed on to each operator
# You can override them on a per-operator basis by passing kwargs to the operator
default_args = dict(
    owner='admin',
    email='admin@local',
    email_on_failure=False,
    email_on_retry=False,
    depends_on_past=False,
    retries=0,
    queue='cpu'
)
# Instantiate the DAG
dag = DAG(
    dag_id='ingest',
    description='Ingest datasets and train a BN model',
    default_args=default_args,
    schedule_interval=None,
    catchup=False,
    max_active_runs=1,
    tags=['Plexus']
)
# Ingest the raw data
ingest = LaunchJobOperator(
    task_id='ingest',
    jobdef='ingest:latest',
    jobqueue='cpu-spot',
    jobname='ingest',
    vcpus=1,
    memory=1024,
    jobtags='ingest:raw',
    stdout=True,
    dag=dag
)
# Process the raw data into a dataset
process = LaunchJobOperator(
    task_id='process',
    jobdef='process:latest',
    jobqueue='cpu-spot',
    jobname='process',
    vcpus=1,
    memory=1024,
    jobtags='process:imdb',
    stdout=True,
    dag=dag
)
# Train a model on the processed data
train = LaunchJobOperator(
    task_id='train',
    jobdef='train:latest',
    jobqueue='gpu-spot',
    jobname='train',
    vcpus=1,
    memory=1024,
    jobtags='train:bn',
    stdout=True,
    dag=dag
)
# Define the dependencies between the tasks
process.set_upstream(ingest)
train.set_upstream(process)
```

To create a new that will launch a Plexus job, import the PlexusLaunchJob class and instantiate it with the required parameters:

- **task_id** (str): a unique, meaningful id for the tasks.
- **job_dataset** (str): the set of parameters available to the jobs endpoint of the Plexus API. Fields that are marked **Required** must be passed in.
- **name** (str): (Required) name of the workload. Provided by user.
- **app** (str): (Required) name of the selected application
- **queue** (str): (Required) name of the selected queue
- **Train_cores** (int): (Required) number of cores per node to run the workload. Default = 1. Must specify a num_nodes is not specified.
- **Train_nodes** (int): (Required) number of nodes to run the workload. Default = 1. Must specify if num_cores is not specified.
- **procs_per_node** (int): number of processes per node to run the workload. Default = None.
- **extra_pmes** (int): additional processes requested to run the workload. Default = None.
- **expected_runtime** (int): max alotted time the workload is allowed to run. Provided in seconds. If the workload exceeds this limitation, it will be killed and marked as "Preempted." If not provided and the workload is run as a service, the user must manually shut it down. Default = None.
- **gpu** (bool): indicates if workload needs a GPU to run.
- **num_gpus** (int): number of gpus required by workload.
- **preexec_script** (str): script that will be executed before running workload. **Default = None**.
- **run_script** (str): script containing commands to run the workload. **Default = None**.
- **postexec_script** (str): script that will be executed after running the workload. Default = None.
- **dag** (DAG): instantiated DAG object

```
NOTE: Parameters that are required and wanted still use an extension indicates that either that parameter or an
alternate parameter must be specified. Both can optionally be specified, but neither specify the required or
the other and vice versa is not. Both specifications are also allowed.

The job_queues app and queue refer to the name of the app and the queue. These can be found by going to the
Plexus UI.
```

1. Login to Plexus at **plexus.cresco.ninfinity.com/login**.

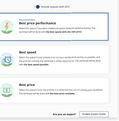
2. Click **Applications** and find the name of the application required to run your job.

3. Click **Queues** and locate the name of the queue to use.

```
NOTE: Plexus applications can be run as batch jobs or as a service. Applications run as a service expose
internal ports and require manual shut down either on expected runtime is passed or if you want to run an
application as a service, we recommend providing an expected_runtime value(in seconds) even though it is not
required. Leaving this field application indicates whether it is run as a service. Click the application and
create the service field.
```

In this example, two more tasks were created and their dependencies were defined. The dependencies were defined so that ingest is launched after task 1 is complete and so on. The Plexus docs show how you can define relationships between tasks, please reference Airflow's **documentation**.

Each task contains a run script that acts as a bucket run command once the container is launched. These scripts will run the bash executable specified at the top of the, which can include the python modules.



Step 3: Package the project

After you have finished creating the project, we recommend packaging it using python's wheelpkgs library. This will make it easier to upload into the Plexus environment when creating a workload. In the example, the project was packaged into a tar file which gets uploaded to the Airflow application as a plex.py script.

Create the Airflow workload

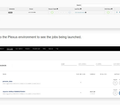
This section describes the step-by-step process to execute the Airflow procedure in Plexus.

To create the Airflow workload

1. Login to Plexus and click **Workloads**.

2. Select **New Workload**.

3. Select the **Apache Airflow** application and then click **Next**.


Choose the Application for this Workload
Search
Apache Airflow
Apache Airflow
Apache Airflow