



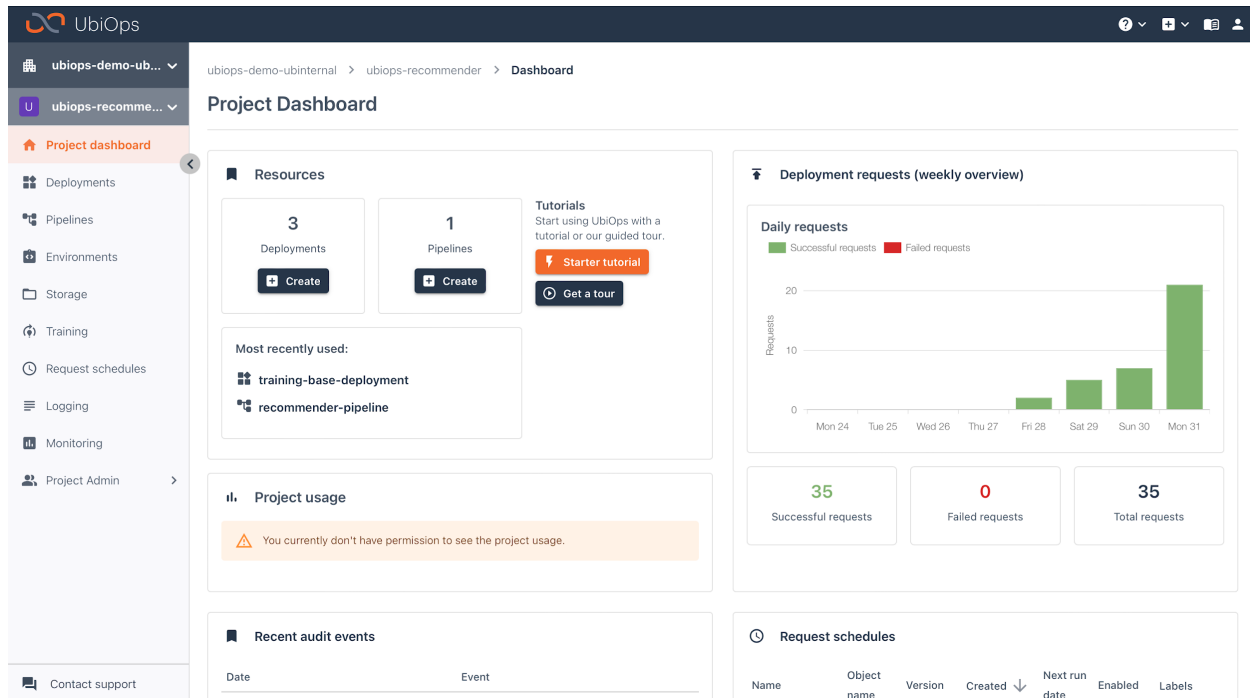
UbiOps & NVIDIA AI Enterprise Deployment Guide

Version 0.9.2 - October 2023

Table of contents

Table of contents	1
1. Introduction to UbiOps	2
2. Offering & High level architecture	4
3. Installing UbiOps	5
3.1. Kubernetes Cluster	6
3.2. Container registry	6
3.3. PostgreSQL database	6
3.4. MongoDB database	7
3.5. Redis	7
3.6. Object storage	7
3.7. Workspace (optional)	8
4. Configuring UbiOps with NVIDIA GPUs	9
4.1. Kubernetes nodepool	9
4.2. UbiOps configuration	13
4.3. Adding NVIDIA AI Enterprise environments to UbiOps	16
5. Using NVIDIA GPUs for AI workloads in UbiOps	19
5.1 Training	19
5.2 Inference	22
6. Examples for using NVIDIA AI Enterprise SDKs	24
6.1. Training a Tensorflow model on UbiOps	26
6.2. Accelerate data processing with NVIDIA Rapids	34
6.3. Running an NVIDIA Triton Inference Server	45
7. More information and resources	49

1. Introduction to UbiOps



UbiOps is developed for data scientists and teams who are looking for an easy, flexible and production-ready way to:

- Deploy, train, and run your own Machine Learning and data science code
- Deploy off-the-shelf LLM & GenAI models
- Run helper functions and other data processing tasks

UbiOps takes care of containerization of user code, deploying it as a microservice with its own API endpoint, as well as request handling and automatic scaling. There are also advanced features for creating data pipelines, version management, job scheduling, monitoring, security and governance. UbiOps has options for deploying AI & ML workloads in both single cloud as well as multi- & hybrid cloud environments.

The UbiOps MLOps platform makes extensive use of NVIDIA technology in the following ways:

- Various NVIDIA GPUs are supported for executing code on the UbiOps platform, e.g. NVIDIA T4, NVIDIA A100 and others
<https://ubiops.com/docs/scaling-resource-settings/#instance-type>. This includes MIG GPUs and VMs leveraging multiple GPUs. UbiOps can run training and inference on NVIDIA GPUs in various cloud provider environments from one interface. The GPUs can be sourced from different hyperscalers such as Amazon Web Services, Azure, and

Google Cloud Platform. Also supported are European Cloud providers such as Bytesnet, Escher Cloud and Intermax.

- UbiOps can orchestrate containerized code directly on VMs with the UbiOps Compute Platform by using nvidia-docker and NVIDIA GPU drivers. Additionally, it's possible to orchestrate containerized code utilizing NVIDIA GPUs on Kubernetes Engine. UbiOps supports Kubernetes installations with a special OS image with NVIDIA GPU ready container runtimes and NVIDIA drivers pre-installed on the nodes, and installations where an NVIDIA GPU operator installs the needed software after boot.
- UbiOps provides a set of environments <https://ubiops.com/docs/environments/> which are container images on which UbiOps client software is preinstalled to connect to the UbiOps API. This includes regular (CUDA) images such as *ubuntu:22.04* or *nvidia/cuda:11.7.1-cudnn8-runtime-ubuntu22.04* and NVIDIA AI Enterprise images such as *nvcr.io/nvaie/tensorflow-3-1:23.03-tf2-nvaie-3.1-py3*. It is possible for UbiOps administrators to make custom environments available on demand using a specific docker build. Alternatively, users can create their own environments with any custom dependencies, or CUDA versions, by using the UbiOps build system (see <https://ubiops.com/docs/howto/howto-install-custom-cuda/>). Users only need to worry about writing and uploading their code.
- The UbiOps team has extensive experience with machine learning applications including inference and training at scale. Support is available to ensure a high level of service for AI in production.

2. Offering & High level architecture

UbiOps has different installation options. UbiOps is available as a fully-managed SaaS solution on <https://app.ubiops.com> and it can also be installed as a private (single-tenant) solution or on your own cloud environment or local hardware.

Options for installation include a 'full installation' of UbiOps or a more lightweight 'nodepool installation'. The 'nodepool installation' means that only the compute environment resides at the customer while the UbiOps control plane is hosted and managed by UbiOps in the UbiOps cloud environment. This allows users to either make use of their own cloud resources, or, for instance, to keep their computing close to their storage.

A high level overview of the UbiOps platform is shown in the visual below. As seen in the visual, the UbiOps control plane offers several MLOps and model management capabilities. On the infrastructure level, the control plane can interface with multiple compute environments to orchestrate inference and training workloads. More details on the functionality of UbiOps can be found in its documentation on <https://ubiops.com/docs/>.

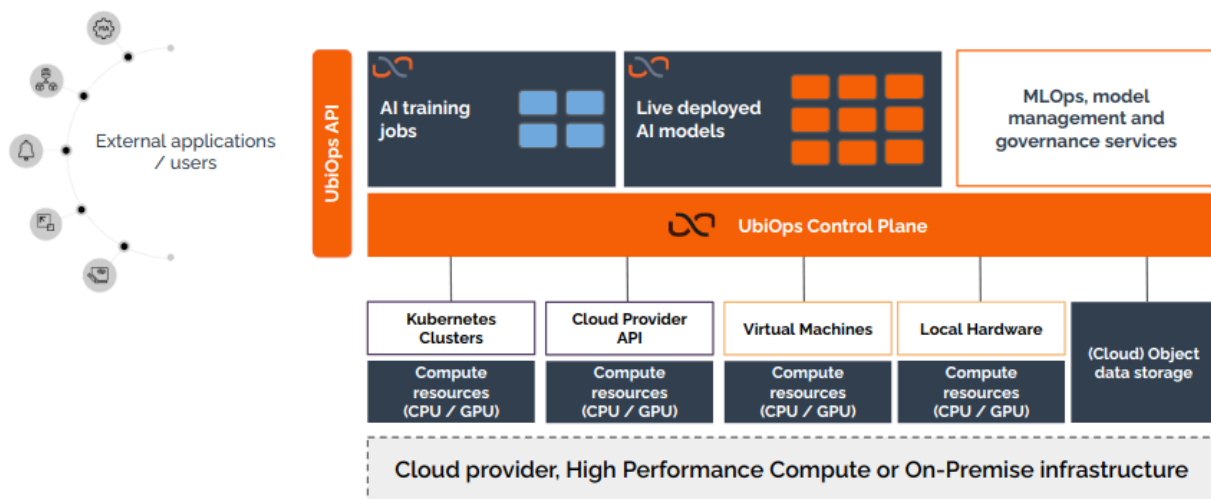


Figure 1: A high-level architecture of how the UbiOps core cluster interacts with external users and applications, and orchestrates workloads across different compute (GPU) environments.

Note: This deployment guide will focus on a Kubernetes-based compute environment. UbiOps offers an alternative installation possibility which makes use of the proprietary UbiOps Compute Platform. In this setup, UbiOps can also connect with UbiOps clients on local hardware, or interface with the API of a Cloud Service Provider for managing compute resources and orchestrating workloads.

3. Installing UbiOps

A high level overview of the infrastructure used by UbiOps can be found in the figure below. As can be seen, UbiOps consists of a set of microservices that run on Kubernetes and are accessible via a browser based *WebApp*, the *UbiOps platform API* and the deployment *Requests Page*. Additional services are used to store state, and common infrastructure such as load balancers, DNS, VPC are needed as well.

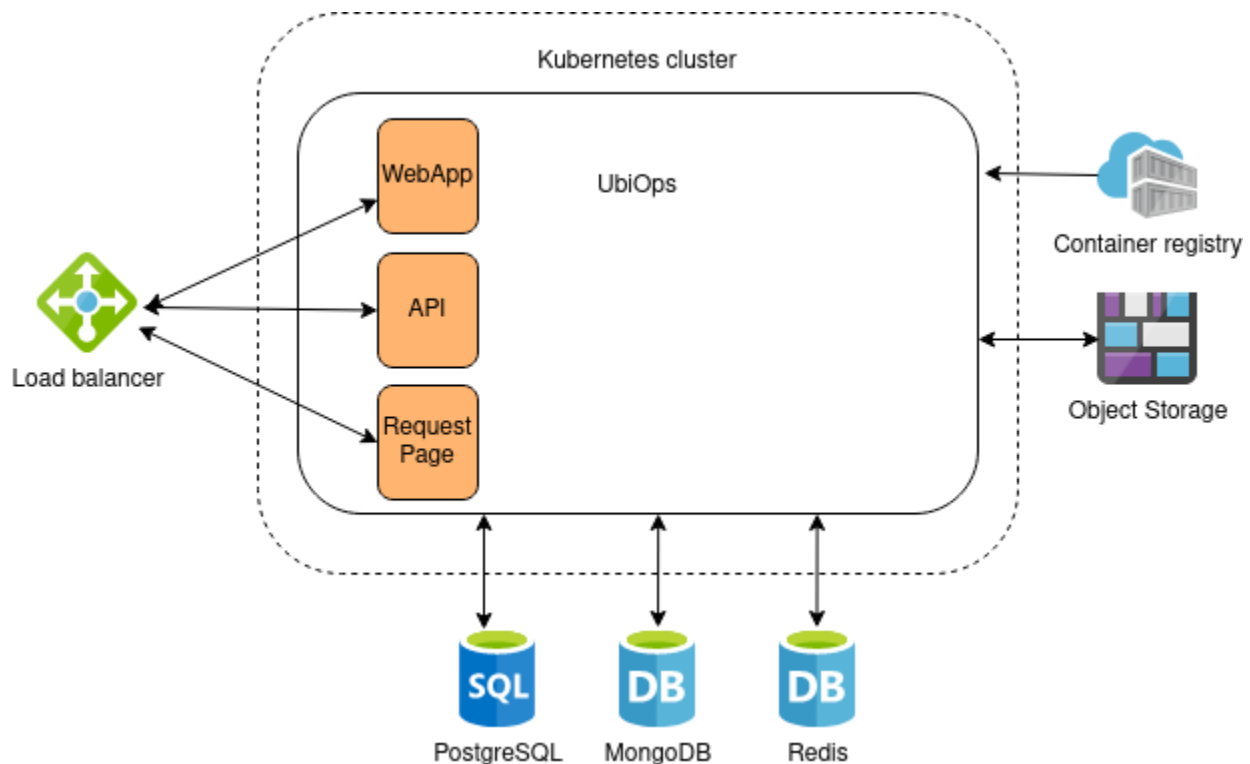


Figure 2: A high-level architecture of the UbiOps core cluster

UbiOps can be installed on infrastructure provisioned using major and smaller cloud providers, in private data centers and in development environments. Detailed steps can be found in <https://installation-guide.docs.ubiops.com/latest/> for the installation of UbiOps on major cloud providers such as Azure, Amazon Web Services (AWS) and Google Cloud Platform (GCP).

Installation on smaller cloud providers, private data centers and development environments is possible as well. Please contact info@ubiops.com for more information.

A brief description of the major components of the UbiOps installation is provided below.

3.1. Kubernetes Cluster

A UbiOps installation needs a Kubernetes cluster. The cluster needs to be set up before starting the UbiOps installation. The following versions are supported in the most recent versions of UbiOps: 1.22, 1.23, 1.24, 1.25, 1.26. Kubernetes versions outside this range may be supported but have not been tested (yet).

The Kubernetes cluster is used to run services like the UbiOps API and WebApp, as well as internal services.

3.2. Container registry

UbiOps requires access to a container registry, both for the Docker images of the UbiOps services as for storage of Docker images of deployments created in UbiOps. Each of the UbiOps Docker images needs to be copied to this container registry.

Each of the major Cloud Service Providers has their own container registry service. Alternatively it's possible to set up a [Docker registry manually](#).

Please make sure the registry is secured using the TLS security protocol, and that an account is available with the rights to create repositories, push and pull images.

3.3. PostgreSQL database

UbiOps requires a PostgreSQL database to store all application related information. We recommend PostgreSQL 13, although UbiOps should work with earlier versions up to 9.6 as well. At least 30 GB storage is required.

Most of the cloud providers offer a managed PostgreSQL service. Alternatively, it's possible to install and manage PostgreSQL manually on a virtual machine.

Recommended is to configure daily or hourly backups with a retention time of at least a couple of days for the PostgreSQL database. High-availability can be considered as well (in addition to the backups).

Please make sure to have an administrative user available that can create databases and users.

3.4. MongoDB database

UbiOps requires a MongoDB database to store logs and metrics. Recommended is the latest MongoDB 4 version, with at least 1 vCPU and 2 GB of memory.

It's recommended to install MongoDB on a virtual machine manually, as no cloud provider offers MongoDB as a managed service. Alternatively a service such as MongoDB Atlas can be considered.

Recommended is to configure daily or hourly backups with a retention time of at least a couple of days for the MongoDB database.

Please make sure to have an administrative user available that can create one database and create users.

3.5. Redis

UbiOps requires a Redis instance, we recommend version 4.0 or higher. It does not need to have persistent storage.

Most of the cloud providers provide a managed Redis service such as [AWS](#), [Azure](#), and [Google](#). A small instance with 1 GB memory is sufficient for a standard installation.

Alternatively, it's possible to install Redis manually on a [virtual machine](#), or even install it in the Kubernetes cluster using a [redis helm chart](#).

3.6. Object storage

UbiOps requires access to a high-performance object storage bucket to store request data and files.

Most of the cloud providers provide a managed blob storage service that is supported by UbiOps (Google Cloud Storage, AWS S3, Azure Blob Storage).

Alternatively, it's possible to set up [MinIO](#) or [Openstack swift](#) on a virtual machine, or even install it in the Kubernetes cluster. The access should be secured using TLS.

Please make sure to have a user available that can create a bucket and has full access to this bucket.

If you want to use the file browser in the UbiOps WebApp to manage files stored in object storage, you will need to configure the object storage to return CORS headers for the URL of the WebApp. Please refer to the documentation of your object storage provider for instructions.

3.7. Workspace (optional)

Regarding the installation process, it is recommended to perform and manage a UbiOps installation from one machine or location, such as a small virtual machine acting as a jumphost. This server will store the UbiOps infrastructure code and configuration, and all installation commands are issued from here.

Alternatively this can be done from a local system, possibly shared with team members using centralized storage, but that can be less convenient and less secure.

4. Configuring UbiOps with NVIDIA GPUs

In this chapter will be explained how to enable NVIDIA GPUs on UbiOps.

4.1. Kubernetes nodepool

UbiOps supports deployments and training jobs to run on a GPU. This feature can be enabled by adding GPU node pool(s) to a Kubernetes cluster. It is possible to add GPU node pools after setting up UbiOps.

A node pool needs to be created with the correct amount of resources to support the deployment instance types that will be used. The nodes in the node pool need to be slightly larger than the maximum size of deployments in UbiOps, to compensate for Kubernetes overhead.

For example, to support deployment instance types with 4 vCPU, 16 GiB RAM and 1 GPU, nodes with 5 vCPU and 20 GB RAM and 1 GPU are recommended. Or alternatively, larger nodes with 9 vCPU, 36 GiB RAM and 2 GPUs - these can support two deployments of the said instance type.

The Kubernetes nodes should have permissions to pull and push images from the container registry, where user code is stored. How this is configured depends on the Cloud provider, but is usually done by giving permissions to the Kubernetes service account. Alternatively, it's possible to configure registry credentials in the UbiOps Administration Portal.

4.1.1. GPU nodepool

The node pool for the GPU deployments should have the labels:

```
Unset
accelerator: nvidia-tesla-t4    # the type of GPU that is used in this
node pool
dedicated: deployments
nvidia.com/gpu: present
node_pool_id: <a randomly generated uuid for the nodepool>
```

And the taint:

Unset

```
NoSchedule dedicated=deployments  
NoSchedule nvidia.com/gpu=present
```

The nodepool id that has been configured will be used when a UbiOps administrator is defining the nodepool in UbiOps later.

The Kubernetes nodes need to have NVIDIA drivers installed and an NVIDIA device plugin installed to discover GPUs. Please follow the steps provided by the cloud provider, e.g. [Google Cloud Services](#) or install the NVIDIA GPU operator <https://github.com/NVIDIA/gpu-operator>.

4.1.2. Multi-instance GPU (MIG) nodepool

UbiOps supports running on multi-instance GPUs (e.g., NVIDIA Ampere series). In this situation each deployment version will be allocated a slice of a GPU instead of the entire GPU.

Depending on the infrastructure provider GPU slices need to be configured in advance or after creating the nodepool in Kubernetes.

It is required to configure a nodepool with the labels matching the MIG configuration. For example, in the case of partition of 1g5 configuration the node pool should have the labels:

Unset

```
accelerator: nvidia-a100-1g5    # the type of GPU that is used in this  
node pool  
dedicated: deployments  
nvidia.com/gpu: present  
node_pool_id: <a randomly generated uuid for the nodepool>
```

And the taint:

Unset

```
NoSchedule dedicated=deployments  
NoSchedule nvidia.com/gpu=present
```

The nodepool id that a user has given for the label will be used when they are defining the nodepool in UbiOps later.

Depending on the infrastructure provider additional labels are required to configure the MIG slices. For example, when using the NVIDIA GPU operator the following node label is needed to configure MIG slices.

Unset

```
nvidia.com/mig.config=all-1g.5gb
```

4.1.3. Multi-GPU nodepool

Multi-GPU nodepools (e.g., 2x NVIDIA A100) can accommodate larger training jobs, and are also supported on UbiOps. In this situation each deployment version will be allocated multiple GPUs.

Unset

```
accelerator: nvidia-a100      # the type of GPU that is used in this node  
pool  
dedicated: deployments  
nvidia.com/gpu: present  
node_pool_id: <a randomly generated uuid for the nodepool>
```

And the taint:

Unset

```
NoSchedule dedicated=deployments  
NoSchedule nvidia.com/gpu=present
```

The nodepool id that a user has given for the label will be used when they are defining the nodepool in the UbiOps administration portal, in a later stage.

4.1.4. Testing

It's recommended to test the Kubernetes GPU nodepool before connecting it to UbiOps with the following workload (or equivalent)

```
Unset
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      image:
        "nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda11.7.1-ubuntu20.04"
      resources:
        limits:
          cpu: 4
          memory: 16Gi
          nvidia.com/gpu: 1 # requesting 1 GPU
        requests:
          cpu: 4
          memory: 16Gi
          nvidia.com/gpu: 1 # requesting 1 GPU
  nodeSelector:
    dedicated: deployments
    node_pool_id: <the previously used uuid>
  tolerations:
    - key: "dedicated"
      value: "deployments"
      operator: "Equal"
      effect: "NoSchedule"
    - key: "nvidia.com/gpu"
      value: "present"
      operator: "Equal"
      effect: "NoSchedule"
```

4.2. UbiOps configuration

To allow UbiOps connecting to the Kubernetes nodepool, create a service account in the cluster.

```
Unset
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ubiops-service-account
  namespace: default
EOF
```

This service account needs access to the entire cluster. Assign the `cluster-admin` role for the user:

```
Unset
kubectl apply -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: ubiops-cluster-admin
  namespace: kube-system
subjects:
- kind: ServiceAccount
  name: ubiops-service-account
  namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
EOF
```

Create a long-lived token for the serviceaccount

Unset

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: ubiops-secret
annotations:
  kubernetes.io/service-account.name: ubiops-service-account
type: kubernetes.io/service-account-token
EOF
```

Get the token:

Unset

```
kubectl describe secret/ubiops-secret
```

We also need the certificate authority data for the cluster. Run the following command and get the field `ca.crt`:

Unset

```
kubectl get secret <SECRET_NAME> -o yaml
```

Now a nodepool can be added to UbiOps from the UbiOps Administration Portal. Navigate to: <https://api.domain.com/v2.1/admin> and login with a superuser account. In the Core/Clusters section, you can click on **Add Cluster**. The following fields are required to create a cluster:

- **name**: Name of the component
- **capacity**: Number of deployment instances that can be deployed on the component

- **master_endpoint**: Endpoint of the cluster
- **certificate_authority_data**: CA certificate for the Kubernetes master
- **credentials**: Required credentials to access the Kubernetes cluster. The service account token that was previously generated in this section should be provided in this field. Give a dictionary with a key **token** and value the token itself.

After adding cluster(s), a nodepool in the cluster needs to be added. Navigate to the **Nodepools** section. The following fields are required to create a nodepool:

- **id**: UUID of the nodepool. Use the nodepool id that you have provided to the nodepool in Kubernetes.
- **name**: Name of the nodepool
- **cluster**: Name of the cluster where the nodepool will be created
- Add resources of the following types:
 - **cpu_standard**: Amount of CPU available in the nodepool
 - **memory_standard**: Amount of memory available in the nodepool
 - **gpu_standard**: Amount of capacity of GPU available in the nodepool. It is possible to update this field later when
 - **storage_standard**: Amount of ephemeral storage that can be used on the nodes by containerized workloads. It's advised to keep 30GB of disk capacity free.

Add node pool

When creating a node pool, please follow the steps:

- Assign deployment instance types to the node pool. Below you can find **Instance types** section. Pick the instance types that can be deployed on this node pool.
- If no **id** is provided, it will be generated automatically.

Id:

Name:

deployments-gpu-a100

Cluster:














Priority:

0

☒ Is active

Schedule timeout:

300

RESOURCES		
RESOURCE	AMOUNT	DELETE?
cpu_standard  	<input type="text" value="11"/>	
memory_standard  	<input type="text" value="80000"/>	
gpu_standard  	<input type="text" value="1"/>	
storage_standard  	<input type="text" value="320000"/>	
 Add another Resource		

SAVE

4.3. Adding NVIDIA AI Enterprise environments to UbiOps

By default, UbiOps provides multiple base environments with different versions of python and CUDA pre-installed. Base environments based on NVIDIA AI Enterprise containers can be configured in the UbiOps administration portal which is located at [https://api.ubiops.\[domain\].com/v2.1/admin](https://api.ubiops.[domain].com/v2.1/admin).

In this section we will explain how to add an environment based on *nvcr.io/nvaie/tensorflow-3-1:23.03-tf2-nvaie-3.1-py3* image to UbiOps.

4.3.1. Creating a UbiOps base environment

A UbiOps base environment consists of a regular container image with UbiOps 'deployment instance' code added to it. The installed UbiOps 'deployment instance' code handles things like code download, requests, log and metric collection and acts as an interface to the UbiOps API. Administrators can create environments using a regular docker build process. For example, it's possible to create an environment based on the NVIDIA Enterprise image *nvcr.io/nvaie/tensorflow-3-1:23.03-tf2-nvaie-3.1-py3* using the following docker file.

```
Unset
FROM nvcr.io/nvaie/tensorflow-3-1:23.03-tf2-nvaie-3.1-py3
...
# Setup a venv for the deployment instance with python 3.10
RUN python3.10 -m venv /var/deployment_instance/venv/

# Install deployment instance requirements
RUN . /var/deployment_instance/venv/bin/activate && pip install -U pip
    && pip install -r /var/deployment_instance/deployment/requirements.txt

# Install deployment instance source files under the deployment_instance user
COPY --chown=deployment:deployment main.py
    /var/deployment_instance/deployment/main.py
COPY --chown=deployment:deployment deployment_process.py
    /var/deployment_instance/deployment/deployment_process.py
COPY --chown=deployment:deployment config/config.yaml-docker
    /var/deployment_instance/deployment/config/config.yaml
COPY --chown=deployment:deployment controller
    /var/deployment_instance/deployment/controller/
```

```

COPY --chown=deployment:deployment deployment_interface
/var/deployment_instance/deployment/deployment_interface/
COPY --chown=deployment:deployment utilities
/var/deployment_instance/deployment/utilities/
...
CMD ["python3", "-u", "/var/deployment_instance/deployment/main.py"]

```

The resulting image with UbiOps dependencies packaged into it we call an environment in UbiOps. The image should be pushed to a container registry.

4.3.2. Making the environment available in UbiOps

The resulting environment should be added to UbiOps in the administration portal. The following fields are required to create an environment:

- **name:** The short name of the environment
- **Display Name:** The name of the environment that will be displayed in the UbiOps UI
- **language:** The coding language for code used in this environment
- **Image repository:** The location of the container image of the environment
- **Image tag:** The version tag of the container image of the environment

Change environment

HISTORY

nvaie3-1-ubuntu23-03-python3-8-tensorflow2

Id: ee81a050-0b83-4aaa-9862-Saefd9e2da1b

Name: nvaie3-1-ubuntu23-03-python3-8-tensorflow2

Display name: NVIDIA enterprise 3.1 TensorFlow 2

Language: Python

Creation date: Sept. 11, 2023, 8:57 a.m.

☒ Gpu required

Image repository: ubiops-deployment-instance-nvaie3-1-ubuntu

Image tag: v4.38.1-tf






☐ Deprecated

☐ Hidden

SAVE

4.3.3. Using the environment in UbiOps

For each deployment version UbiOps users can configure which environment should be used to execute code in. For example it's possible for a model written for tensorflow to run in an NVIDIA enterprise environment by selecting "NVIDIA enterprise 3.1 Tensorflow 2" environment.

tf  EDIT  LOGS  DUPLICATE  MAKE DEFAULT  DELETE

General

Requests


Metrics

Environment variables

Revisions

Status

Environment

 available

NVIDIA enterprise 3.1 TensorFlow 2

Examples will be provided in the next chapters.

5. Using NVIDIA GPUs for AI workloads in UbiOps

In the UbiOps documentation there is an extensive set of examples on how to utilize NVIDIA GPUs, see e.g. <https://ubiops.com/docs/deployments/gpu-deployments/>

End-users of the UbiOps platform can easily deploy workloads on GPUs. Specifically they can run inferencing jobs using *deployments*, or training runs using *experiments*. Generally, anything that runs on Python, can run on UbiOps.

A UbiOps workload contains three layers:

1. The instance type. This specifies the resource requests of the workload, and includes a number of vCPU cores, a disk size, memory, and/or a GPU card.
2. An environment. This specifies the base container image with UbiOps client software installed and is used to execute the customers code. Users can install additional OS packages and pip packages in an environment.
3. A code layer, with optionally additional files, such as model artifacts or look-up tables. The layer includes all relevant code and is downloaded to the environment during initialization of an instance. The user can specify code that runs when initiating a new instance of the deployment.

For training workloads, the code layer is different with each training run, and is part of the input of a training job that runs on the specified environment. This allows a user to experiment with different scripts more iteratively.

For inference workloads, the code layer is uploaded once and is then fixed, unless explicitly rebuilt.

We will now provide a high-level description of how users can deploy a training algorithm on top of NVIDIA AI Enterprise environments. Afterwards, we provide a similar description of how users can deploy a GPU-powered inference workload on such an environment.

5.1 Training

Let's say that a user wants to run a training job with a UbiOps environment based on an NVIDIA [enterprise 3.1 tensorflow 2](#) container image and make use of an NVIDIA Tesla T4 to speed up his training time. By making use of this environment the user is able to make use of all packages that have been pre-installed in this container. The user needs to write code in the format that UbiOps understands, see <https://ubiops.com/docs/training/>. For a training run, this means transforming his training script into a `train` method, e.g.:

```
Python
import module x,y,z

def train(training_data, parameters, context):
    # Training code
```

To deploy this script, he can make use of the `Training` functionality at UbiOps. Step one is to create an experiment and define the instance type and the code environment, on which all training scripts in this experiment will run. In this example, they will select a deployment instance type with a NVIDIA Tesla T4 GPU added to it, and select `NVIDIA enterprise 3.1 tensorflow 2` as the environment.

Training

Experiments

Evaluation

Create new

Name	Created ↓	Instance type	Environment
tensorflow-training	19-09-2023 12:09	16384 MB + NVIDIA Tesla T4 + NVAIE	NVIDIA enterprise 3.1 tensorflow 2

The training run is then a serverless set-up, which can receive instructions to initiate training a run. Input fields are the training script, (a reference to) training data and a set of (hyper)parameters. This allows a user to quickly initiate multiple training runs, e.g. with different input (hyper)parameters, or a different training script.

Multiple training runs can run in parallel. The scaling of all required compute happens in the background. The experiment overview pages allows users to keep track of all running training jobs.

tensorflow-training

[EDIT](#) [DELETE](#)

[Create run](#)

General Environment variables

Status	available
Created	19-09-2023 12:09
Edited	19-09-2023 12:09
Environment	NVIDIA enterprise 3.1 tensorflow 2
Instance type	16384 MB + NVIDIA Tesla T4 + NVAIE
Default bucket	default

Description

Train tensorflow experiment

Runs

Search

<input type="checkbox"/>	Status	Name	Created ↓	Duration	Script location	
<input type="checkbox"/>	completed	tensorflow-train-run	19-09-2023 14:59:21	00:00:21	ubiops-file:///default/experiments/tensorflow-training/runs/zudkueos/code/train.py	📄 📋 📄 ✖ 🗑
<input type="checkbox"/>	completed	tensorflow-train-run	19-09-2023 13:14:42	00:00:21	ubiops-file:///default/experiments/tensorflow-training/runs/dtgvuwx/code/train.py	📄 📋 📄 ✖ 🗑
<input type="checkbox"/>	failed	tensorflow-train-run	19-09-2023 13:01:22	00:00:07	ubiops-file:///default/experiments/tensorflow-training/runs/stsfvq/code/train.py	📄 📋 📄 ✖ 🗑

20 rows 1-3 of 3 1-3 of 3

During a training run, they can inspect logs to see if their job runs as planned.

tensorflow-train-run

[DUPLICATE](#) [DELETE](#)

[View results](#)

Status	completed
Created	19-09-2023 14:59
Started	19-09-2023 15:04
Completed	19-09-2023 15:05
Duration	00:00:21
Script location	ubiops-file:///default/experiments/tensorflow-t...
Output artefact	ubiops-file:///default/deployment_requests/09...

Description

Trying out a run

Logs

[Full screen](#)

<input checked="" type="checkbox"/>	19-09-2023 15:04:46	Request 0939aabc-40d4-45b9-8319-e549cc7086e2 started
<input checked="" type="checkbox"/>	19-09-2023 15:04:48	2023-09-19 15:04:48.355020: I tensorflow/core/platform/cpu_feature_guard.cc:194] This TensorFlow binary is optimized with o...
<input checked="" type="checkbox"/>	19-09-2023 15:04:48	To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
<input checked="" type="checkbox"/>	19-09-2023 15:04:50	Processing training run
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	Package Version
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	-----
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	abs1-py 1.0.0
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	argon2-cffi 21.3.0
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	argon2-cffi-bindings 21.2.0
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	asttokens 2.2.1
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	astunparse 1.6.3
<input checked="" type="checkbox"/>	19-09-2023 15:04:51	attrs 22.2.0

5.2 Inference

Similarly, inference workloads are supported as well. UbiOps takes care of the orchestration and scaling of the model based on the configured scaling parameters and the amount of traffic hitting the model endpoint. Instead of defining one step, as we did in a training job, you can now define two steps. One that runs when a model is initialized up, and one that runs when it receives a request (input data).

```
Python
class Deployment:

    def __init__(self):
        # Load model into memory, open up connections to databases

    def request(self, data):
        # Process data
        return {"output":123}
```

Again, we can select the instance type and a code environment in which the deployment code will run. With deployments, an end-user has more control over compute settings. They can specify a minimum and maximum number of active instances, an idle time of these instances, and assign a static egress IP.

Below we show an example for a deployment that hosts a Stable Diffusion model, with two versions. Both versions run the same code. In the initialization of an instance, a Stable Diffusion model is loaded from an object storage into memory. During request handling, the model is used to generate an image from an input prompt. One version runs on an instance type with access to an NVIDIA T4 Tesla GPU, where the other version has access to an NVIDIA Ampere A100 (40GB) GPU card. Because of the separation of code (environment) from the instance type, solutions can easily be lifted and shifted to different instance types.

stable-diffusion

[EDIT](#) [LOGS](#) [DUPLICATE](#) [DELETE](#)[Create request](#)[General](#) [Configuration](#) [Audit events](#) [Environment variables](#) [Use deployment](#)

Created 08-06-2023 11:45

Edited 24-10-2023 18:58

Endpoint URL <https://api.ubiops.com/v2.1/projects/dem...>

Description

Application: computer-vision

No description available

Versions

[Create version](#)

Search

Status	Version	Environment	Minimum instances	Maximum instances	Created	Edited ↓	Labels
available	default v-t4	stable-diffusion-environment	0	1	08-06-2023 11:45	20-10-2023 14:18	
available	v-a100	stable-diffusion-environment	0	1	21-06-2023 12:49	25-07-2023 11:57	

20 rows

1-2 of 2

ubiops-demo-ubinternal > demo-main > Deployments > stable-diffusion > v-a100 > General

v-a100

[EDIT](#) [LOGS](#) [DUPLICATE](#) [MAKE DEFAULT](#) [DELETE](#)[Create request](#)[General](#) [Requests](#) [Metrics](#) [Environment variables](#) [Revisions](#)

Status available

Environment [stable-diffusion-environment](#)

Created 21-06-2023 12:49

Edited 25-07-2023 11:57

Last file upload 21-06-2023 12:49

Last request -

Endpoint URL <https://api.ubiops.com/v2.1/projects/dem...>

Notification groups

Failed requests	No group yet	EDIT
Finished requests	No group yet	EDIT

[Upload deployment file](#)

Instance type

76000MB + 11 vCPU + NVIDIA Ampere A100 40GB

Minimum instances

0

Maximum instances

1

Maximum idle time

1800 seconds

Request retention mode

Full (Metadata + request in- and output)

Request retention time

1 week (604800)

Maximum express queue size

100

Maximum batch queue size

100000

Static IP address

X

Description

No description available

6. Examples for using NVIDIA AI Enterprise SDKs

This chapter shows several examples on how to use the NVIDIA Enterprise SDKs in UbiOps for model training, deployment and inference. The examples are based on UbiOps tutorials and make use of the [UbiOps Python Client](#) to interface with the UbiOps API.

The code blocks in the examples are intended to run from a Jupyter Notebook environment. You can also find these examples in the [UbiOps documentation](#).

6.1. Training a Tensorflow model on UbiOps

This example is based on the following articles:

https://ubiops.com/docs/ubiops_tutorials/tensorflow-training/tensorflow-training/
<https://ubiops.com/training-ml-models-on-ubiops/>

In this example, we will show how to run a training job for a Tensorflow model on the UbiOps platform.

We will define and create a UbiOps training script. Using the UbiOps Python client we will create an experiment, where we specify the container in which the code will run and which we can analyze and track our results.

We will first show you how to set project variables and to initialize the UbiOps API Client, setting up a connection to your project.

1) Set project variables and initialize the UbiOps API Client

First, make sure you create an API token with **project editor** permissions in your UbiOps project and paste it below. Also fill in your corresponding UbiOps project name.

You can install the UbiOps Python Client using

```
Unset  
pip install --upgrade ubiops
```

For more information how to authenticate with the API using the Python Client, please see <https://ubiops.com/docs/interacting/client-libraries/>

Python

```
from datetime import datetime
dt = datetime.now()
import yaml
import os
import ubiops

API_TOKEN = 'Token ' # Paste your API token here. Don't forget the `Token`
prefix
PROJECT_NAME = '' # Fill in the corresponding UbiOps project name

configuration = ubiops.Configuration(host="https://api.ubiops.com/v2.1")
configuration.api_key[ 'Authorization' ] = API_TOKEN

api_client = ubiops.ApiClient(configuration)
core_instance = ubiops.CoreApi(api_client=api_client)
training_instance = ubiops.Training(api_client=api_client)
print(core_instance.service_status())
```

2) Initialize the UbiOps client library with the API token

Now we import the UbiOps Python client and authenticate with the API.

Python

```
import ubiops

configuration = ubiops.Configuration(host="https://api.ubiops.com/v2.1")
configuration.api_key[ 'Authorization' ] = API_TOKEN

client = ubiops.ApiClient(configuration)
api = ubiops.CoreApi(client)
api.service_status()
```

6.1.1. About the training code

The training function we will deploy expects a path to a zipped **training data** file, **the number of epochs**, and the **batch_size** as input. As output it will give the trained **model artifact** as well as the final **loss** and **accuracy** for the training job. - The training code and data is based on one of the Tensorflow tutorials for training a model on the 'flowers dataset'.

Source: https://www.tensorflow.org/tutorials/load_data/images

The corresponding URL for the training data archive is:

https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz

Training code in- and output variables		
	Fields (type)	Keys of dictionary
Input fields	training_data (file)	
	parameters (dict)	epochs (integer)
		batch_size (integer)
Output fields	artifact (file)	
	metrics (dict)	accuracy (float)
		loss (float)
		loss_history (list[float])
		acc_history (list[float])

After authenticating with the API using the UbiOps Python Client. Set-up a training instance in case you have not done this yet in your project. This action will create a base training deployment that is used to host training experiments.

```
Python
training_instance = ubiops.Training(api_client=api_client)
try:
    training_instance.initialize(project_name=PROJECT_NAME)
except ubiops.exceptions.ApiException as e:
    print(f"The training feature may already have been initialized in your project:\n{e}")
```

6.1.2. Create an experiment

The basis for model training in UbiOps is an 'Experiment'. An experiment has a fixed code environment and compute (instance) definition, but it can hold many different 'Runs'.

You can create an experiment in the WebApp or use the Client Library, as we're here. When creating the environment, we can specify the instance type and the code environment. Our training code needs an environment to run in, with a specific Python language version, and some dependencies, like [Tensorflow](#). We can use the **NVIDIA AI Enterprise 3.1 TensorFlow** base environment for this, which is the specialized NVIDIA container for the TensorFlow library. This base environment is using [this](#) NVIDIA AI Enterprise container.

This bucket will be used to store your training jobs and any model callbacks. In case you want to continue without creating a bucket, you can use the [default](#) bucket that is always present inside your account.

```
Python
EXPERIMENT_NAME = 'training-experiment-demo' # str
BUCKET_NAME = 'default'

try:
    experiment = training_instance.experiments_create(
        project_name=PROJECT_NAME,
        data=ubiops.ExperimentCreate(
            instance_type='16384mb_t4_nvaie',
            description='Train test experiment',
            name=EXPERIMENT_NAME,
            environment='nvaie3-1-ubuntu23-03-python3-8-tensorflow2',
            default_bucket=BUCKET_NAME
        )
    )
except ubiops.exceptions.ApiException as e:
    print(e)
```

6.1.3. Load the training data

We will download the publicly available **flower photos** dataset. We will train our model on this file.

```
Python
import urllib.request

url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
training_data = "flower_photos.tgz"

urllib.request.urlretrieve(url, training_data)

print(f"File downloaded successfully to '{training_data}'.")
```

(Optional) Extract the zip-file, to inspect its content. The tar file will be applied directly to the training job, so this step is not required.

```
Python
import tarfile

file_dir = "flower_photos"
with tarfile.open(training_data, 'r:gz') as tar:
    path = tar.extractall("./")
```

6.1.4. Define and start a training run

A training job in UbiOps is called a run. To run any Python code on UbiOps, we need to create a file named **train.py** and include our training code here. This code will execute as a single 'Run' as part of an 'Experiment' and uses the code environment and instance type (hardware) as defined with the experiment as shown before. Let's take a look at the training script.

The UbiOps **train.py** structure is quite simple. It only requires a **train()** function, with input parameters **training_data** (a file path to your training data) and **parameters** (a dictionary that contains parameters of your choice). If we upload this training code, along with the

`training_data` file and some values for our input parameters, a training run is initiated! You can easily execute multiple training runs with different hyperparameters, or with slightly altered code. Each training run can either reuse the same code with different parameters, or contain a different version of the `train.py` file.

```
Python
RUN_NAME = 'training-run'
RUN_SCRIPT = f'{RUN_NAME}.py'

%%writefile {RUN_SCRIPT}
import logging
import pathlib
import subprocess
import sys
import tarfile

import joblib
import tensorflow as tf

logger = logging.getLogger("TrainingInformation")

def train(training_data, parameters, context):
    """All code inside this function will run when a call to the deployment is
    made."""
    logger.info("Processing training run")
    subprocess.check_call(args=[sys.executable, "-m", "pip", "list"])
    subprocess.check_call(args=["nvcc", "--version"])

    img_height = 180
    img_width = 180
    batch_size = int(parameters["batch_size"]) # Specify the batch size
    nr_epochs = int(parameters["nr_epochs"]) # Specify the number of epochs

    # Load the training data
    extract_dir = "flower_photos"

    with tarfile.open(training_data, "r:gz") as tar:
        tar.extractall("./")

    data_dir = pathlib.Path(extract_dir)
```

```

train_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size,
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size,
)

class_names = train_ds.class_names
print(class_names)

# Standardize the data
normalization_layer = tf.keras.layers.Rescaling(1.0 / 255)
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))

# Configure the dataset for performance
AUTOTUNE = tf.data.AUTOTUNE
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

# Train the model
num_classes = 5

model = tf.keras.Sequential(
    [
        tf.keras.layers.Rescaling(1.0 / 255),
        tf.keras.layers.Conv2D(32, 3, activation="relu"),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(32, 3, activation="relu"),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(32, 3, activation="relu"),
        tf.keras.layers.MaxPooling2D(),
    ]
)

```

```

        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(num_classes),
    ]
)

model.compile(
    optimizer="adam",
    loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=["accuracy"],
)

history = model.fit(train_ds, validation_data=val_ds, epochs=nr_epochs)

eval_res = model.evaluate(val_ds)

# Return the trained model file and metrics
joblib.dump(model, "model.pkl")
fin_loss = eval_res[0]
fin_acc = eval_res[1]

print(history)
print(history.history)
return {
    "artifact": "model.pkl",
    "metrics": {
        "fin_loss": fin_loss,
        "fin_acc": fin_acc,
        "loss_history": history.history["loss"],
        "acc_history": history.history["accuracy"],
    },
}

```

Python

```

new_run = training_instance.experiment_runs_create(
    project_name=PROJECT_NAME,
    experiment_name=EXPERIMENT_NAME,
    data=ubiops.ExperimentRunCreate(
        name=RUN_NAME,
        description='Trying out a first run run',
        training_code= RUN_SCRIPT,

```



```

training_data= training_data,
parameters={
    'nr_epochs': 20, # example parameters
    "batch_size" : 32
},
timeout=14400
)
)

```

We can easily finetune our training code and execute a new training code, and analyze the logs along the way. When training a model it is important to keep track of the training progress and convergence. We do this by looking at the training loss and accuracy metrics. Packages like Tensorflow will print these for you continuously, and we're able to track them in the logging page of the UbiOps UI. If you notice a training job is not converging, you're able to cancel the request and try it again with different data or different parameters.

6.1.5. Evaluating the output

When the training runs are completed, the training run will provide you with the trained parameter file, the final accuracy and loss. The parameter file is stored inside a UbiOps bucket. You can easily navigate to this location from the training-run interface. You can compare metrics of different training runs easily inside the Evaluation page of the Training tab, allowing you to analyze which code or which hyperparameters worked best.

Training

Experiments

Evaluation

Add runs to compare

Experiment

Select...

▼

Training-experiment-demo: run-epochs-10-batch-8

Training-experiment-demo: run-epochs-10-batch-64

Training-experiment-demo: run-epochs-10-batch-32

Edit columns

Name	Experiment	fin_loss	fin_acc	loss_history	acc_history	
run-epochs-10-batch-8	training-experiment-demo	2.628247022628784	0.6198909878730774	[1.2490973472595215,0.94071739...	[0.473433256149292,0.637261569...	
run-epochs-10-batch-64	training-experiment-demo	1.8176277875900269	0.6212534308433533	[1.3690218925476074,1.09670400...	[0.40497276186943054,0.554836...	
run-epochs-10-batch-32	training-experiment-demo	1.7870458364486694	0.6021798253059387	[1.3388144969940186,1.04256057...	[0.4134877324104309,0.59264302...	

6.2. Accelerate data processing with NVIDIA Rapids

This example is based on the following tutorial:

https://ubiops.com/docs/ubiops_tutorials/nvidia-rapids-benchmark/nvidia-rapids-benchmark-tutorial/

NVIDIA RAPIDS is a suite of open-source software libraries and APIs developed by NVIDIA that gives scientists and data analysts the ability to execute end-to-end data science and analytics pipelines completely on GPUs. This makes many different data analytics and machine learning workflows a lot faster. This tutorial will showcase how you can train a Linear Regression classifier on a synthetic dataset, and optimize its runtime by using libraries that are part of the NVIDIA RAPIDS suite. We will run the benchmark on the NVIDIA RAPIDS container.

Again, we are first going to define some project variables and initialize the UbiOps API Client

```
Python
!pip install --upgrade ubiops

import ubiops

API_TOKEN = "Token ..." # TODO: Add your UbiOps token here
PROJECT_NAME = "" # TODO: Add your project name here

DEPLOYMENT_NAME = "nvidia-rapids-benchmark"
VERSION_NAME = "v1"

DEPLOYMENT_DIR = "deployment_package"
ENVIRONMENT_DIRECTORY_NAME = "environment_package"

configuration = ubiops.Configuration(host="https://api.ubiops.com/v2.1")
configuration.api_key['Authorization'] = API_TOKEN

api_client = ubiops.ApiClient(configuration)
core_instance = ubiops.CoreApi(api_client=api_client)
training_instance = ubiops.Training(api_client=api_client)

print(core_instance.service_status())

# Create a directory to store our deployment code
!mkdir {DEPLOYMENT_DIR}
```

Now our workspace is all set up, let's start creating our baseline model.

6.2.1. Create a baseline model

In order to showcase the performance improvements by utilizing NVIDIA RAPIDS, we want to have a baseline model to test against first. For this, we will create a simple **Random Forest** classifier. We are going to use **Scikit-Learn** and **Pandas** for this.

We are creating the following functions for the baseline model:

- **generate_dataset**: Generate a random dataset for a certain amount of samples and features
- **convert_to_pandas**: Convert our dataset to a **Pandas Dataframe** (useful for when we start creating an NVIDIA RAPIDS accelerated model)
- **train_lr**: Train a Linear Regression model (with Scikit-Learn)
- **make_predictions**: Make model predictions
- **calculate_mse**: Calculate the Mean Square Error (MSE)

```
Python
%%writefile {DEPLOYMENT_DIR}/baseline_model.py

import time

import pandas as pd
from sklearn.datasets import make_classification
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

class BaselineModel:
    def __init__(self):
        self.sklearn_lr = LinearRegression()

    @staticmethod
    def generate_dataset(n_samples, n_features=20):
        x, y = make_classification(n_samples=n_samples, n_features=n_features)
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
        return x_train, x_test, y_train, y_test

    @staticmethod
    def convert_to_pandas(x_train, y_train, x_test):
        pandas_x_train = pd.DataFrame(x_train)
```

```

pandas_y_train = pd.Series(y_train)
pandas_x_test = pd.DataFrame(x_test)
return pandas_x_train, pandas_y_train, pandas_x_test

def train_lr(self, pandas_x_train, pandas_y_train):
    start_time = time.time()
    self.sklearn_lr.fit(pandas_x_train, pandas_y_train)
    return time.time() - start_time

def make_predictions(self, pandas_x_test):
    start_time = time.time()
    sklearn_predictions = self.sklearn_lr.predict(pandas_x_test)
    return sklearn_predictions, time.time() - start_time

@staticmethod
def calculate_mse(y_test, sklearn_predictions):
    return mean_squared_error(y_test, sklearn_predictions)

```

6.2.3. Accelerate the model with NVIDIA RAPIDS

Now that we have our baseline model, we can accelerate this model by using the corresponding NVIDIA RAPIDS equivalent libraries/functions. The table below lists the NVIDIA RAPIDS library equivalents to the "standard" libraries.

Standard Library	NVIDIA RAPIDS Equivalent
Pandas	cuDF
Scikit-learn	cuML

```

Python
%%writefile {DEPLOYMENT_DIR}/rapids_model.py

import time
import cudf
from cuml.linear_model import LinearRegression
from cuml.metrics import mean_squared_error

```

```

class RapidsModel:
    def __init__(self):
        self.cu_lr = LinearRegression()

    @staticmethod
    def convert_to_cudf(pandas_x_train, pandas_y_train, pandas_x_test):
        cudf_x_train = cudf.DataFrame.from_pandas(pandas_x_train)
        cudf_y_train = cudf.Series(pandas_y_train)
        cudf_x_test = cudf.DataFrame.from_pandas(pandas_x_test)
        return cudf_x_train, cudf_y_train, cudf_x_test

    def make_predictions(self, cudf_x_test):
        start_time = time.time()
        cu_predictions = self.cu_lr.predict(cudf_x_test)
        return cu_predictions, time.time() - start_time

    def train_lr(self, cudf_x_train, cudf_y_train):
        start_time = time.time()
        self.cu_lr.fit(cudf_x_train, cudf_y_train)
        return time.time() - start_time

    @staticmethod
    def calculate_mse(y_test, cu_predictions):
        return mean_squared_error(y_test, cu_predictions)

```

As you can see in the code block above, the core is exactly the same as in the baseline model! Some parameters are changed to give a better description, but all the function calls are entirely the same. The only difference is the library from which it is imported. In the baseline model, this is *sklearn*, in the accelerated model, it's *cudf* and *cuml*.

6.2.4. Implement the models into a UbiOps deployment

Now that we've written our code for a baseline model and a NVIDIA RAPIDS accelerated model, we can integrate both into a UbiOps deployment. UbiOps deployment requires fixed in- and outputs, as is outlined in the [documentation](#).

We will use the following input/output structure:

Input/Output	Name	Type	Description
Input	n_samples	Integer	Number of samples in the dataset
Input	n_features	Integer	Number of features per sample
Output	scikit-mse	Double Precision	Mean Squared Error using scikit-learn
Output	cuml-mse	Double Precision	Mean Squared Error using cuML
Output	scikit-train-time	Double Precision	Training time using scikit-learn
Output	cuml-train-time	Double Precision	Training time using cuML

Let's integrate the models into the UbiOps deployment structure, with the inputs/outputs as specified in the table above!

```

Python
%%writefile {DEPLOYMENT_DIR}/deployment.py

import time

from baseline_model import BaselineModel
from rapids_model import RapidsModel

class Deployment:
    def __init__(self):
        self.baseline_model = None
        self.rapids_model = None

    def request(self, data):
        n_samples = data.get("n_samples", 1000000)
        n_features = data.get("n_features", 20)

        self.baseline_model = BaselineModel()
        self.rapids_model = RapidsModel()

        start_time = time.time()
        x_train, x_test, y_train, y_test =
self.baseline_model.generate_dataset(n_samples, n_features)
        print("Dataset generation time: ", time.time() - start_time)

        start_time = time.time()

```

```

        pandas_x_train, pandas_y_train, pandas_x_test =
self.baseline_model.convert_to_pandas(x_train, y_train, x_test)
        print("Pandas conversion time: ", time.time() - start_time)

        # Delete the dataframes to free up memory
        del x_train, x_test, y_train

        start_time = time.time()
        cudf_x_train, cudf_y_train, cudf_x_test =
self.rapids_model.convert_to_cudf(
            pandas_x_train,
            pandas_y_train,
            pandas_x_test
        )
        print("CuDF conversion time: ", time.time() - start_time)

        sklearn_train_time = self.baseline_model.train_lr(
            pandas_x_train,
            pandas_y_train,
        )
        cu_train_time = self.rapids_model.train_lr(cudf_x_train, cudf_y_train)

        sklearn_predictions, sklearn_prediction_time =
self.baseline_model.make_predictions(pandas_x_test)
        cu_predictions, cu_prediction_time =
self.rapids_model.make_predictions(cudf_x_test)

        sklearn_mse = self.baseline_model.calculate_mse(y_test,
sklearn_predictions)
        cu_mse = self.rapids_model.calculate_mse(y_test, cu_predictions)

        return {
            "scikit-mse": sklearn_mse,
            "cuml-mse": cu_mse.tolist(),
            "scikit-train-time": sklearn_train_time,
            "cuml-train-time": cu_train_time
        }

```

We can now continue to create our UbiOps deployment and upload our deployment files.

6.2.5. Create and upload the deployment to UbiOps

Finally, we've reached the last step of the setup process: creating a deployment on UbiOps and uploading our deployment code to UbiOps. We can use the **NVIDIA AI Enterprise 3.1 Rapids** base environment for this, which is the specialized NVIDIA container for the Rapids library. This base environment is using the NVIDIA RAPIDS container from NVIDIA AI Enterprise.

Let's begin by creating a new deployment.

```
Python
input_fields = [
    {'name': 'n_samples', 'data_type': 'int'},
    {'name': 'n_features', 'data_type': 'int'}
]

output_fields = [
    {'name': 'scikit-mse', 'data_type': 'double'},
    {'name': 'cuml-mse', 'data_type': 'double'},
    {'name': 'scikit-train-time', 'data_type': 'double'},
    {'name': 'cuml-train-time', 'data_type': 'double'}
]

deployment_template = ubiops.DeploymentCreate(
    name=DEPLOYMENT_NAME,
    description='Deployment to demonstrate NVIDIA RAPIDS model acceleration',
    input_type='structured',
    output_type='structured',
    input_fields=input_fields,
    output_fields=output_fields
)

deployment = core_instance.deployments_create(project_name=PROJECT_NAME,
data=deployment_template)
```

Now we add a deployment version to the newly created deployment:

Python

```
version_template = ubiops.DeploymentVersionCreate(
    version=VERSION_NAME,
    environment='nvaie3.1-ubuntu22.04-python3.8-rapids',
    instance_type='16384mb_t4_nvaie',
    maximum_instances=1,
    minimum_instances=0,
    maximum_idle_time=600, # = 10 minutes
    request_retention_mode='full'
)

core_instance.deployment_versions_create(
    project_name=PROJECT_NAME,
    deployment_name=DEPLOYMENT_NAME,
    data=version_template
)
```

At last, we zip our deployment code and upload it to the newly created deployment version:

Python

```
import shutil
deployment_archive = shutil.make_archive(DEPLOYMENT_DIR, 'zip', DEPLOYMENT_DIR)

core_instance.revisions_file_upload(
    project_name=PROJECT_NAME,
    deployment_name=DEPLOYMENT_NAME,
    version=VERSION_NAME,
    file=deployment_archive
)
```

6.2.7. Run the deployment

Now it's time to use our deployment. Let's define a function to create a request and a function to plot results:

```

Python
!pip install matplotlib
import matplotlib.pyplot as plt

# function to create deployment requests
def create_request(core_instance, features, samples):
    data = {
        "n_features": features,
        "n_samples": 10**samples
    }
    request = core_instance.deployment_version_requests_create(
        project_name=PROJECT_NAME,
        deployment_name=DEPLOYMENT_NAME,
        version=VERSION_NAME,
        data=data
    )
    result_save = {
        "n_samples": data["n_samples"],
        "n_features": data["n_features"],
        **request.result
    }
    print(request.result)
    return result_save

def plot_graph(results, time_key, title, feature_list):
    plt.figure(figsize=(10, 10))
    plt.title(title)
    plt.xlabel("Number of samples")
    plt.ylabel("Time (s)")
    plt.xscale("log")

    for i, features in enumerate(feature_list):
        filtered_results = [result for result in results if result["n_features"] ==
features]
        n_samples = [result["n_samples"] for result in filtered_results]
        scikit_times = [result[f'scikit-{time_key}'] for result in filtered_results]
        cuml_times = [result[f'cuml-{time_key}'] for result in
filtered_results]
        color = 'blue' if features == 5 else 'red'

        plt.plot(n_samples, scikit_times, label=f"Scikit-learn {features} features",
linestyle="dashed", color=color)
        plt.plot(n_samples, cuml_times, label=f"CuML {features} features",
linestyle="solid", color=color)
    plt.legend()

```

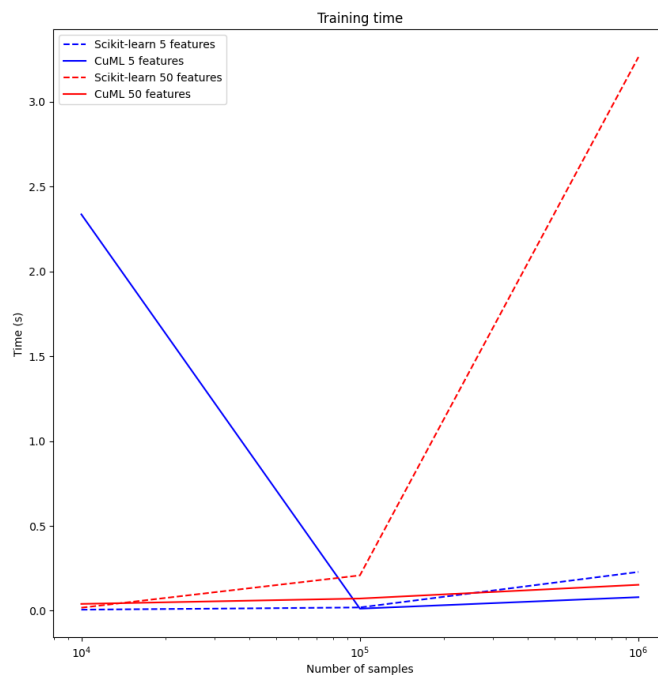
Let's call our function now and save the results:

```
Python
features = [5, 50]
range_samples = range(4, 7)

results = [create_request(core_instance, feature, n_samples) for n_samples in
range_samples for feature in features]
```

We can proceed to plot the results now:

```
Python
plot_graph(results, "train-time", "Training time", features)
plt.show()
```



We can conclude that using NVIDIA RAPIDS libraries greatly speeds up our training time, especially on larger datasets.

6.3. Running an NVIDIA Triton Inference Server

This example is based on the following tutorial:

<https://ubiops.com/docs/howto/howto-nvidia-triton/>

This example will outline how to run a [NVIDIA Triton Inference Server](#) inside a Ubiops deployment. The NVIDIA Triton Inference Server will be deployed with [PyTriton](#), a Python client for the Triton Inference Server. This can allow you to use one UbiOps deployment to host multiple models. We will show how to do this in the following steps:

1. Set up a UbiOps environment
2. Create a Triton Inference Server deployment and bind model(s) to the Triton server
3. Create the UbiOps deployment `request` method

6.3.1. Environment Setup

We need the `nvidia-pytriton` package to set-up a Triton server. We can use the **NVIDIA AI Enterprise 3.1 Triton** base environment for this, which is the specialized NVIDIA container for the Triton library. This base environment is using [this](#) container from NVIDIA AI Enterprise.

6.3.2. Create a Triton Inference Server deployment and bind model(s) to the Triton server

Setting up a (basic) Triton server consists of the following steps:

1. Create a Triton object
2. Bind a model to the Triton object
3. Run/serve the Triton object

The implementation of these steps for a UbiOps deployment is shown in the following code block:

```
Python
from pytriton.triton import Triton
from pytriton.model_config import ModelConfig
class Deployment:
```

```

def __init__(self):
    # Step 1: Call the Triton constructor
    self.triton = Triton()
    # Step 2: Bind a model to the Triton object
    self.triton.bind(
        model_name="Your Model name", # TODO: Add your model name
        infer_func=self.your_infer_function, # TODO: Add your infer function
        inputs=[
            # TODO: Add your input tensors
        ],
        outputs=[
            # TODO: Add your output tensors
        ],
        config=ModelConfig() # TODO: Add your model config
    )

    # Step 3: Run the Triton object
    self.triton.run()

```

6.3.3. Create the request method

Finally, we construct the `request` method of the deployment. This method is called when a request is made to the deployment. We would like to select the model that is used for the inference and the data that is sent to this specific model. Therefore, we need to add the two following inputs to the deployment:

Input	Type
json_data	String
model_name	String

The `json_data` input contains the data that is sent to Triton Server. The `model_name` input contains the name of the model that is used for the inference. The `request` method looks like this:

```

Python
import requests

```

```

class Deployment:
    def request(self, data):
        model_name = data["model_name"]
        json_data = data["json_data"]
        print(f"Received json_data: {json_data}")
        url = f"http://localhost:8000/v2/models/{model_name}/infer"
        headers = {"Content-Type": "application/json"}
        response = requests.post(url, headers=headers, data=json_data)

        return {"output": response.text}

```

6.3.4. Final code

The final code looks as follows:

```

Python
from pytriton.triton import Triton
from pytriton.model_config import ModelConfig
import requests

class Deployment:

    def __init__(self):
        # Step 1: Call the Triton constructor
        self.triton = Triton()
        # Step 2: Bind a model to the Triton object
        self.triton.bind(
            model_name="Your Model name", # TODO: Add your model name
            infer_func=self.your_infer_function, # TODO: Add your infer function
            inputs=[
                # TODO: Add your input tensors
            ],
            outputs=[
                # TODO: Add your output tensors
            ],
            config=ModelConfig() # TODO: Add your model config
        )

```

```

# Step 3: Run the Triton object
self.triton.run()

def request(self, data):

    model_name = data["model_name"]
    json_data = data["json_data"]

    print(f"Received json_data: {json_data}")
    url = f"http://localhost:8000/v2/models/{model_name}/infer"

    headers = {"Content-Type": "application/json"}
    response = requests.post(url, headers=headers, data=json_data)

    return {"output": response.text}

```

We have now created a deployment (and environment) that runs a NVIDIA Triton Inference Server inside UbiOps. This deployment is able to host multiple models at the same time in a single deployment, with many more benefits!

7. More information and resources

You can find more information about installation options for UbiOps at <https://installation-guide.docs.ubiops.com/latest/>

The examples are based on the following tutorials and docs pages if you would like more information:

https://ubiops.com/docs/ubiops_tutorials/tensorflow-training/tensorflow-training/

<https://ubiops.com/training-ml-models-on-ubiops/>

https://ubiops.com/docs/ubiops_tutorials/nvidia-rapids-benchmark/nvidia-rapids-benchmark-tutorial/

<https://ubiops.com/docs/howto/howto-nvidia-triton/>