



Technical Report

MLOps with NetApp and BeeGFS

Affordable MLOps for everyone

Joe McCormick, Peter Upton, NetApp
May 2021 | TR-4890

Abstract

Machine learning operations (MLOps) is an overall term for the platforms, tools, and best practices that allow AI projects to go from being proof of concept (POC) to production ready. BeeGFS is an open-source parallel file system supported by NetApp® that emphasizes ease-of-use, flexibility, and scalability. This technical report describes how to use BeeGFS as a preproduction workspace for data science projects that can easily transition into production using data pipelines supported by the NetApp portfolio. We use the NetApp Data Science Toolkit and MLOps platforms built on Kubernetes drawing from our experience building infrastructure to support data science initiatives in air-gapped environments securing high-value datasets and intellectual property.

TABLE OF CONTENTS

Introduction	4
Operationalizing AI	4
Concepts and components	5
Additional resources	5
BeeGFS	6
BeeGFS CSI driver	6
Data Science Toolkit	6
S3cmd	6
Deploying foundational MLOps infrastructure	6
Kubernetes	6
Configuring a load balancer	8
Installing the NVIDIA GPU operator	8
Deploying the BeeGFS CSI driver	9
Integrating other NetApp storage and Kubernetes	9
Setting up BeeGFS for data science	10
Overview	10
Organizing data in BeeGFS	10
Accessing existing directories in BeeGFS from Kubernetes	11
Enabling users to dynamically provision BeeGFS storage on-demand	14
Kubeflow	15
Overview	15
Deployment	15
Managing multiple users, projects, and teams in Kubeflow	18
Using BeeGFS with Jupyter notebooks as the workspace for data science	18
Building Kubeflow Pipelines using NetApp storage	24
Conclusion	42
Where to find additional information	42
Version history	42

LIST OF FIGURES

Figure 1) Kubernetes cluster.	8
Figure 2) BeeGFS file system.	10
Figure 3) Jupyter notebook server creation (storage options).	21

Figure 4) Jupyter notebook server creation (miscellaneous settings).....	21
Figure 5) Jupyter notebook server provisioning in progress.....	22
Figure 6) Jupyter notebook server launcher.....	22
Figure 7) JupyterLab theme.....	23
Figure 8) Verifying BeeGFS capacity and performance from a terminal in Jupyter.....	23
Figure 9) Cloning a Git repository from a terminal in Jupyter.....	23
Figure 10) Example Kubeflow Pipelines workflow.....	24
Figure 11) Viewing versions of a Kubeflow Pipeline.....	34
Figure 12) Viewing raw data from a Jupyter notebook server.....	40
Figure 13) Viewing Snapshot copies in ONTAP System Manager.....	41
Figure 14) Viewing the cleaned dataset and model in ONTAP System Manager.....	41

Introduction

Note: For simplicity, machine learning (ML), deep learning (DL), and similar techniques are often referred to as artificial intelligence (AI) in this document, unless context demands specificity.

By the end of this report, the reader should have a better understanding of the following concepts:

- How environments supporting AI and other data science initiatives can be set up to effectively share equipment, including GPUs and storage, between multiple data scientists/users.
- How to build pipelines that operationalize AI including orchestrating data workflows across NAS, object storage (S3), and parallel file systems.
- Strategies to leverage BeeGFS as a high-speed workspace and scratch space for data science.

This report is intended to be useful to readers from a wide range of backgrounds; therefore, we'll start with a high-level background on the emerging engineering discipline of MLOps.

The word MLOps is a compound of ML and operations. It defines a maturing set of platforms, tools, and best practices organizations can leverage to move AI initiatives from POC to production ready. A good MLOps platform enables initial research, experimentation, and collaboration between data scientists, while reducing the amount of equipment needed for them to work together efficiently. These platforms should also support development of end-to-end data pipelines simplifying deployment and refinement of AI models in production (sometimes referred to as operationalizing AI).

This document describes how to set up an environment that can use BeeGFS and other solutions in the NetApp portfolio to support MLOps platforms such as Kubeflow running on Kubernetes clusters. Although some customers might prefer to build their environments by using open-source platforms such as Kubeflow and Apache Airflow, others might wish to work with one of the many [NetApp partners](#) that provide fully integrated turnkey solutions. Given the tendency for open source and turnkey MLOps solutions to build on Kubernetes, this document provides details about integrating NetApp solutions into environments supporting AI and modern data analytics even beyond the specific platforms discussed in this document.

Because much of the software discussed is constantly being improved to support the growing AI and MLOps community, this report links to the latest external documentation instead of listing step-by-step directions for common tasks such as deployment and installation. This approach allows the document to focus on how everything comes together to create a complete MLOps solution and highlights where NetApp can remove roadblocks around data. Where applicable, we will supplement with notes from our own experiences that we hope the reader finds valuable when deploying and integrating these tools and platforms.

Operationalizing AI

AI introduces a number of unique challenges around data that make building storage solutions more nuanced than simply thinking about performance and capacity. For those with more of an Ops than ML background, it is helpful to think of data as the source code used to program (or train) the software that provides some artificial intelligence (or inferencing) capability. While each organization's use cases have unique requirements, we can make some generalizations when introducing the challenges that operationalizing AI presents, particularly with regards to managing data workflows.

Data collection

Typically, the first hurdle with new data science initiatives is moving relevant data from wherever it originated into an environment where data scientists can explore and manipulate it as part of model development. For example, web-scale object storage such as [StorageGRID](#) can ingest from edge data sources such as sensors or cameras using S3 APIs. While this is valuable to collect and organize large quantities of unstructured data from disparate sources, it might be inefficient (or impossible) for data scientists to interact with directly. Therefore, after the data relevant to a particular initiative is identified, it

is often useful to move it to a file-based workspace that provides POSIX access semantics that work with a range of applications, and performance more suited for live data analysis.

Data preparation and cleaning

Providing data scientists some form of scratch workspace is critical, because it is likely the data won't be in an optimal form for training and will require some sort of transformation. For example, extracting compressed data, transforming PDF files to text, parsing out relevant sections of each file, or any combination of other operations needed to get data into an appropriate format and structure based on the AI techniques/libraries the data scientist wishes to employ.

These types of operations are typically known as data preparation or cleaning. In poorly equipped environments, this operation can take more time than actually training the model. A parallel file system such as BeeGFS provides data scientists a high-speed workspace that can scale to fit any dataset (or any number of datasets and data scientists).

Note: In a production environment, this scratch space is often wiped or reclaimed after each batch job or pipeline run.

Training and versioning the model

With a cleaned dataset, data scientists can begin training and refining the model. As previously mentioned, the dataset can be thought of to represent the source code used to build the model. Therefore, changes to the dataset can both negatively and positively affect the accuracy and reliability of the model. Being able to quickly identify the dataset used to train a particular version of a model allows data scientists to debug model predictions. In some instances, organizations might be called on to explain why a model made a certain prediction, so having this reproducibility after the model moves to production might also be required for regulatory or compliance reasons.

The need for a version control system quickly becomes apparent, though traditional tools used in software development such as Git are ill-suited to tracking large datasets. By using NetApp ONTAP® data management software and NetApp Snapshot™ technology, organizations can keep highly efficient point-in-time (PiT) records of datasets and models that allow for traceability and version control with minimal overhead. These models can then be deployed anywhere in the world by using NetApp SnapMirror® technology and a data fabric powered by NetApp.

Moving to production and establishing an AI control plane

The long-term challenge of operationalizing AI is developing a data pipeline capable of automating the entire process of retraining the model as new data is generated. This frees data scientists to move on to other initiatives. The storage requirements remain the same between preproduction and production, but it is crucial that all components are highly automated and tightly integrated for hands-off operations. This includes ingesting new data, preparing the data, using the cleaned dataset to refine the model, performing regression and other testing, and updating the model in production.

The example used in this document shows how the NetApp portfolio delivers storage integrated with MLOps platforms and tools to solve each of the unique challenges imposed by various stages of AI data pipelines for both preproduction and production workflows.

Concepts and components

Additional resources

You might want to familiarize yourself with several related concepts and components including AI, containers, Kubernetes, NetApp Trident, and ONTAP, which are introduced in the “Concepts and Components” section of [TR-4798: NetApp AI Control Plane](#).

Note: It is not necessary to read TR-4798 in its entirety. Refer to the specific sections in TR-4798 as needed.

BeeGFS

[BeeGFS](#) is a POSIX-based parallel file system [supported by NetApp](#). This file system can distribute files and metadata across multiple servers and storage systems providing a storage solution with low latency and high throughput along with incredible scalability. BeeGFS also supports remote direct memory access over InfiniBand and RoCE, reducing CPU utilization on Kubernetes nodes that would otherwise be incurred by high-speed networking. These capabilities make BeeGFS ideal as a scratch workspace for reviewing and preparing unstructured data, which is the focus of this document. This is [not the only practical use case](#) for BeeGFS—it can easily be used as cost-effective storage behind various analytics platforms or more structured data sources.

If you don't have BeeGFS, and you want try it, you can [easily set it up](#) on one or more nodes. If you're deploying with NetApp E-Series, end-to-end deployment is [automated using Ansible](#).

BeeGFS CSI driver

The [BeeGFS Container Storage Interface](#) (or CSI) driver was developed by NetApp to bring the benefits of BeeGFS to applications running in Kubernetes. The driver allows directories in BeeGFS to be consumed as persistent volumes (PVs) in Kubernetes. The driver allows for dynamic storage provisioning where users can submit a persistent volume claim (PVC) that provides them isolated access to a new directory in BeeGFS. The driver also allows for static storage provisioning where an administrator manually creates a PVC referencing an existing directory in BeeGFS that users in Kubernetes wish to access. For more information about getting started with the driver, see the section “Deploying the BeeGFS CSI driver.”

Data Science Toolkit

The [NetApp Data Science Toolkit](#) is a Python library that makes it easier for data scientists and other end users to perform various data management tasks, including near instantaneous provisioning, cloning, or snapshotting of data volumes and JupyterLab workspaces.

S3cmd

[S3cmd](#) is a free command line tool and client that enables uploading, retrieving, and managing data using the S3 protocol.

Deploying foundational MLOps infrastructure

This section covers the Kubernetes deployment and the configuration needed to integrate with NVIDIA GPUs and NetApp storage solutions.

Kubernetes

Overview

This section discusses the options to deploy a new Kubernetes cluster that can be used to run Kubeflow or other MLOps platforms. As with choosing an MLOps platform, there are multiple options to deploy Kubernetes depending on whether you prefer a turnkey solution or a more open-source DIY approach. On the list of turnkey solutions includes offerings from various [cloud providers](#) and certified [Kubernetes Distributions](#) running on-premises or in the cloud. The [DIY route](#) typically involves either bootstrapping a cluster with `kubeadm` or installing Kubernetes using `kops` or `Kubespray`.

For this environment, we used [Kubespray](#), which uses [Ansible](#) as a deployment engine for Kubernetes. Because we already use Ansible to manage our environment, this allows us to easily overlay managing our Kubernetes cluster on top of our existing Ansible roles and playbooks managing the underlying servers and storage systems. To deploy nodes using NVIDIA GPUs, the [DeepOps](#) project builds on Kubespray to deploy Kubernetes with GPU support following NVIDIA's best practices. In our environment, we did not use DeepOps, so we'll briefly discuss how to add support for NVIDIA GPUs to an existing Kubernetes cluster.

Regardless of how you've deployed or plan to deploy your Kubernetes cluster, NetApp solutions can be integrated with the MLOps platform of your choice by using the [container storage interface](#).

Deploying Kubernetes using Kubespray

This optional section summarizes how to get started with Kubespray. If this information is not relevant to your environment, proceed to the next section. When deploying using Kubespray, pick a specific release instead of using the latest version of the master branch. This example uses [v2.15](#), so any documentation referenced is at that tag.

Using Kubespray can essentially be narrowed down to two steps:

1. Create a directory containing an Ansible inventory file as well as any files containing the host and group variables needed to describe the cluster you want to deploy. Sample inventory files are located [here](#).

Note: This example uses INI files, but YAML is also supported.

2. Run the Kubespray `cluster.yml` playbook against the inventory. This pulls in and executes all the necessary Ansible roles used to deploy Kubernetes.

Running the Kubespray playbook requires having Ansible (along with a number of other dependencies) installed. The easiest way to get started is to use the [prebuilt Docker images](#) on Quay (an alternative image registry to Docker Hub) that corresponds with the Kubespray release you want to use. For more information, see [how to setup your inventory](#) based on the environment you want to deploy. Specifically, if you are deploying in an [air-gapped environment](#), there are some additional variables you need to override.

After your inventory is set up, complete the following the steps to deploy Kubernetes using the Docker image:

Note: These commands will all be executed on a machine with your Ansible inventory (commonly known as an Ansible control node) and requires that machine to have password-less SSH set up to the future Kubernetes nodes and a shell where we can run Docker commands.

1. Pull the Docker image corresponding with your release.

```
docker pull quay.io/kubespray/kubespray:v2.15.0
```

2. Change to the directory containing your inventory.
3. Run the following command, replacing `<PATH_TO_.SSH>` with the absolute path to your local `.ssh/` directory so that you can use the password-less SSH configuration setup for the local user inside the Docker container:

```
docker run --rm -it --mount type=bind,source="$(pwd)",dst=/inventory --mount type=bind,source=<PATH_TO_.SSH>,dst=/root/.ssh/ quay.io/kubespray/kubespray:v2.15.0 bash
```

4. This attaches you to a bash shell inside the container where you can run Ansible commands, including the one to deploy Kubespray.

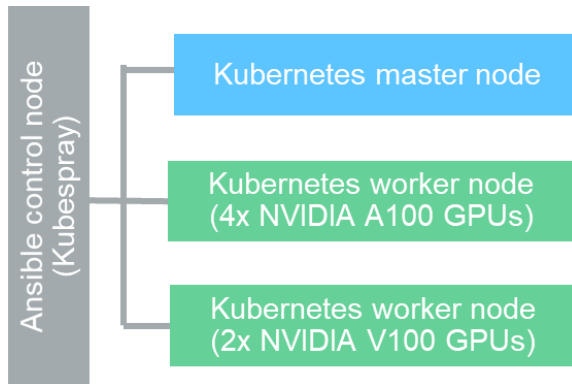
```
ansible-playbook -i /inventory/inventory.yml cluster.yml --ask-become-pass --become
```

Note: Unless Ansible is setup to use the root user you will need to use `--become`. We also use `--ask-become-pass` to avoid having to store the password, making it slightly more secure.

At this point, Kubespray will run and deploy Kubernetes. When it's complete, log into the nodes designated as master nodes and run the `kubectl` commands. For more information, see [Connecting to Kubernetes](#).

Figure 1 is an example of a Kubernetes cluster.

Figure 1) Kubernetes cluster.



To run workloads on the master node (for example, if this is a single node deployment for test purposes), run the following command to allow scheduling pods to the master node:

```
kubectl taint node --all node-role.kubernetes.io/master:NoSchedule-
```

Configuring a load balancer

Exposing applications such as Kubeflow running in Kubernetes to the outside world requires a load balancer. If you are running in the cloud, see your cloud provider's documentation for more detail. If you are running on-premises, [MetalLB](#) provides a straightforward solution.

If you are using Kubespray (with or without DeepOps) to deploy and manage Kubernetes, there is an [option to have MetalLB automatically deployed](#). In this example, we set the following in `group_vars/k8s-cluster/addons.yml` to configure a single IP for MetalLB.

```
# MetalLB deployment
metallb_enabled: true
metallb_ip_range:
  - "<IPv4>/32" # To specify a single IP address in a pool use /32 in the CIDR notation.
                # Reference: https://metallb.universe.tf/configuration/
```

If Kubernetes is already deployed, rerun the Kubespray playbook to apply the configuration.

Note: This example uses a single IP with MetalLB that has a DNS record for each application running in the Kubernetes cluster that we want to expose externally.

Installing the NVIDIA GPU operator

The NVIDIA GPU Operator allows you to add support for NVIDIA GPUs to a new or existing Kubernetes cluster and will deploy and manage the lifecycle of the drivers needed to use the GPUs. A comprehensive getting started guide can be found [here](#). Pay close attention to the prerequisites, particularly the need to blacklist the `nouveau` driver on some systems.

Deploying the operator is usually done using Helm, with [directions available for air-gapped clusters](#). The operator requires [Node Feature Discovery \(NFD\)](#), which automatically annotates nodes that have GPUs installed (along with a number of other capabilities) and is how pods can be scheduled to the correct node when a GPU is requested. By default, the operator installs NFD, but it can optionally be disabled if it is already setup. A few notes from our experience:

- It is strongly recommended that all nodes in your cluster are running the same kernel version to avoid unpredictable interoperability issues.
- If you need to uninstall the GPU operator (for example, to facilitate troubleshooting), the NVIDIA driver modules might still be loaded. Nodes either need to be rebooted or you can run `rmmmod nvidia nvidia_modeset nvidia_uvm` before reinstalling the GPU Operator. Failing to do so can cause major problems.

Deploying the BeeGFS CSI driver

Comprehensive documentation for the [latest release](#) of the driver, including how to deploy the driver in an [air-gapped environment](#), can be found on [GitHub](#). For information about how to use the driver to enable MLOps environments, see the section titled, “Setting up BeeGFS for data science.”

Integrating other NetApp storage and Kubernetes

To use other storage solutions in the NetApp portfolio with Kubernetes, install [NetApp Trident](#). Trident is a CSI-compliant container storage orchestrator that provides persistent storage integration for Kubernetes into the broad NetApp portfolio. It currently includes support for NetApp E-Series, ONTAP, and NetApp SolidFire® and a range of cloud solutions including Azure NetApp Files, NetApp Cloud Volumes Service on Google Cloud, and the Cloud Volumes Service on Amazon Web Services (AWS).

Trident can be [deployed](#) either using the Trident Operator or `tridentctl`. In our environment, we chose to install using the [manual operator installation](#). This allowed us to maintain manifest files that showed how the entire Kubernetes cluster was deployed. We also completed the setup to [configure on-demand volume snapshots](#) to enable us to use ONTAP Snapshot copies to manage versions of datasets and models.

Deploying Trident in an air-gapped environment using the operator

Trident can easily be deployed in an air-gapped environment if the necessary Docker images are available through a proxy or an alternative internal image registry. From [step 3](#) of the Trident Operator deployment guide, before running `kubectl create` with the `tridentorchestrator_cr.yaml` file, make the following additions to the spec field:

1. Set `silenceAutoSupport` to disable the call home functionality (unless you have a proxy in place).
2. Use `imageRegistry` to set an alternate image registry for required sig-storage images.
3. Use `tridentImage` to specify the registry and image name for Trident.
4. Use `autosupportImage` to specify the registry and image name for the Trident autosupport image.

For example, the `tridentorchestrator_cr.yaml` that was used in our environment.

```
apiVersion: trident.netapp.io/v1
kind: TridentOrchestrator
metadata:
  name: trident
spec:
  debug: true
  namespace: trident
  silenceAutosupport: true
  imageRegistry: docker.repo.eng.netapp.com/sig-storage
  tridentImage: docker.repo.eng.netapp.com/netapp/trident:21.01.1
  autosupportImage: docker.repo.eng.netapp.com/netapp/trident-autosupport:21.01
```

Setting up BeeGFS for data science

Overview

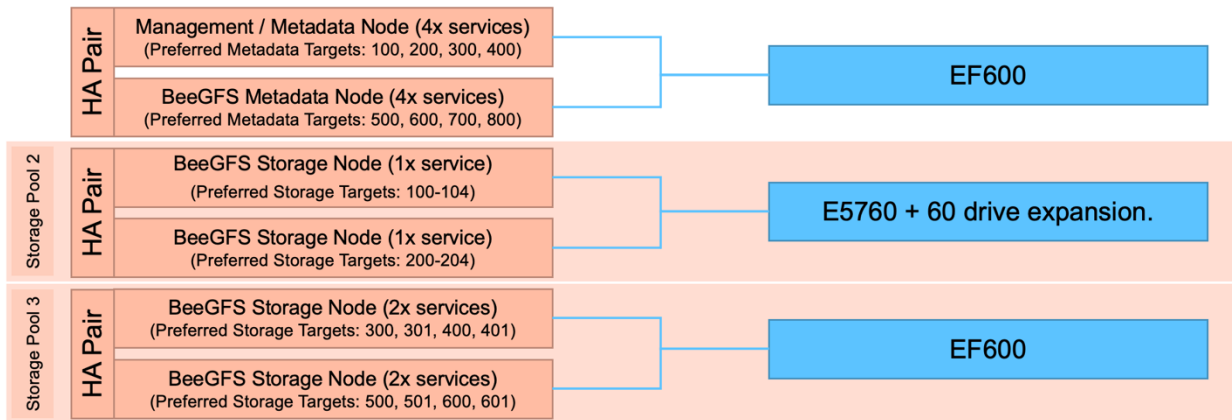
This section demonstrates how existing directories in BeeGFS can be exposed to workloads running in Kubernetes using a [static provisioning](#) workflow. Although the examples walk through one possible way to organize data in BeeGFS, any existing directory in BeeGFS can be easily presented in Kubernetes using this workflow. This section also covers setting up [dynamic provisioning](#) using Kubernetes storage classes, allowing users to self-provision storage as isolated directories in an existing BeeGFS file system.

The examples in this section are used as the storage foundation for demonstrating how BeeGFS can be used with the MLOps platforms discussed in subsequent sections.

Organizing data in BeeGFS

The outputs in Figure 2 show an example of a BeeGFS file system backed by a single [E5760](#) containing NL-SAS HDDs and an [EF600](#) with NVMe SSDs. Each E-Series storage system presents multiple logical volumes (consisting of multiple physical drives) that the BeeGFS file system can use to store distributed metadata and [stripe each file across](#). To reduce the rack footprint, each BeeGFS server runs multiple BeeGFS services leveraging NUMA zones to optimize performance. BeeGFS metadata services have one E-Series volume as their metadata target. BeeGFS storage services backed by nearline-SAS drives have five E-Series volumes as storage targets, while services backed by NVMe-SSD drives have two E-Series volumes as storage targets.

Figure 2) BeeGFS file system.



BeeGFS target IDs are numbered to correspond with the node ID of their BeeGFS service. For example, targets with IDs 100–104 belong to the BeeGFS storage service with ID 100. You can see all the metadata and storage targets in the file system by using `beegfs-df`:

```
[jmcormi@ictml625h1b namespaces]$ beegfs-df
```

```
METADATA SERVERS:
```

TargetID	Cap. Pool	Total	Free	%	ITotal	IFree	%
=====	=====	=====	=====	=	=====	=====	=
100	normal	936.9GiB	934.1GiB	100%	655.4M	624.2M	95%
200	normal	936.9GiB	933.3GiB	100%	655.4M	613.9M	94%

```
[...]
```

```
STORAGE TARGETS:
```

TargetID	Cap. Pool	Total	Free	%	ITotal	IFree	%
=====	=====	=====	=====	=	=====	=====	=
100	normal	72944.0GiB	68061.1GiB	93%	1529.8M	1529.8M	100%

101	normal	72944.0GiB	67503.9GiB	93%	1529.8M	1528.8M	100%
102	normal	72944.0GiB	67578.3GiB	93%	1529.8M	1529.8M	100%
103	normal	72944.0GiB	67573.4GiB	93%	1529.8M	1528.8M	100%
104	normal	72944.0GiB	67577.8GiB	93%	1529.8M	1529.8M	100%
200	normal	72944.0GiB	67801.7GiB	93%	1529.8M	1528.8M	100%
201	normal	72944.0GiB	67854.9GiB	93%	1529.8M	1529.8M	100%
202	normal	72944.0GiB	68074.2GiB	93%	1529.8M	1528.8M	100%
203	normal	72944.0GiB	68078.6GiB	93%	1529.8M	1529.8M	100%
204	normal	72944.0GiB	68105.5GiB	93%	1529.8M	1528.8M	100%
300	normal	2993.5GiB	2036.0GiB	68%	314.0M	314.0M	100%
301	normal	2993.5GiB	2035.9GiB	68%	314.0M	314.0M	100%
400	normal	2993.5GiB	2035.9GiB	68%	314.0M	314.0M	100%
401	normal	2993.5GiB	2036.0GiB	68%	314.0M	314.0M	100%
500	normal	2993.5GiB	2036.0GiB	68%	314.0M	314.0M	100%
501	normal	2993.5GiB	2035.9GiB	68%	314.0M	314.0M	100%
600	normal	2993.5GiB	2035.9GiB	68%	314.0M	314.0M	100%
601	normal	2993.5GiB	2036.0GiB	68%	314.0M	314.0M	100%

You can also use storage pools to organize these targets based on the storage system from which they originated.

```
[jmccormi@ictml625h1b namespaces]$ beegfs-ctl --liststoragepools
```

Pool ID	Pool Description	Targets	Buddy Groups
1	Default		
2	E5760	100,101,102,103,104,200,201,202,203,204	
3	EF600	300,301,400,401,500,501,600,601	

In this example, data is organized in BeeGFS under a `k8s/namespaces` directory where `k8s` corresponds to the name of the Kubernetes cluster (in case there is more than one). To set up storage for the `netapp` Kubernetes namespace, create a subdirectory `netapp` initially containing directories for `raw-datasets` and `shared-scratch`.

```
[jmccormi@ictml625h1b namespaces]$ tree
```

```

.
├── netapp
│   ├── raw-datasets
│   └── shared-scratch

```

3 directories, 0 files

Note: Kubernetes namespaces are required when managing multiuser access to MLOps platforms such as Kubeflow. This is discussed in detail in the subsequent sections of this document.

The `raw-datasets` directory is used to store datasets that might be periodically updated from their original source. These datasets will eventually take up a lot of storage space, so they should be placed on the E5760 represented by storage pool 2.

```
[jmccormi@ictml625h1b netapp]$ sudo beegfs-ctl --setpattern --storagepoolid=2 raw-datasets/
```

New storage pool ID: 2

Lastly, `shared-scratch` provides some space for data scientists to actively explore and manipulate the datasets. To provide the best performance, place data written to that directory on an all-flash EF600, represented by storage pool 3:

```
[jmccormi@ictml625h1b netapp]$ sudo beegfs-ctl --setpattern --storagepoolid=3 shared-scratch/
```

New storage pool ID: 3

Accessing existing directories in BeeGFS from Kubernetes

To use existing BeeGFS directories in Kubernetes, create PVs and PVCs using a Kubernetes manifest file. Although the following example uses the directories created in the previous example, you can use

any existing directory in BeeGFS that contains data the users want to access. These directories can also be shared by users of other workload managers such as SLURM.

In a single YAML file, first define the PV and PVC for the `raw-datasets` directory.

Note: The use of `---` allows you to define multiple Kubernetes objects in the same file.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: netapp-raw-datasets
spec:
  accessModes:
    - ReadWriteMany
  capacity:
    storage: 500Ti
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: beegfs.csi.netapp.com
    volumeHandle: beegfs://192.168.3.100/k8s/namespaces/netapp/raw-datasets
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: raw-datasets
  namespace: netapp
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Ti
  storageClassName: ""
  volumeName: netapp-raw-datasets
```

On the `PersistentVolume`, the name is arbitrary; however, under `accessModes`, `ReadWriteMany` is included to allow the volume to be mounted as read-write by multiple Kubernetes nodes. By using a `persistentVolumeReclaimPolicy` of `Retain`, the volume (along with its backing directory) is prevented from being deleted if the PVC is deleted. This is strongly recommended to avoid accidental data deletion. For more information, see the Kubernetes documentation on [Persistent Volumes](#).

For `capacity`, we choose `500Ti` because the storage pool backed by the E5760 provides more than `500TiB`. This configuration gives us a quick way to know how much data we could place in that PV. There is nothing preventing overprovisioning (giving administrators flexibility to create multiple PVs in excess of a storage pool's capacity), then monitor for actual user consumption and add capacity to BeeGFS as needed.

Note: In the 1.0 version of the BeeGFS CSI driver, the `capacity` field has no effect since the PV maps to a directory, and `capacity` has little meaning in the context of any POSIX compliant file system. For more information, see the [driver documentation](#). Other CSI drivers have similar limitations.

The driver must be set to `beegfs.csi.netapp.com`. The `volumeHandle` indicates the BeeGFS filesystem and directory to use for the static PVC (using the following format):

```
beegfs://<BeeGFS Management IP or Hostname>/<Directory Path>
```

Note: Consumers of PVs are only able to access the provided directory path (and any subdirectories) allowing administrators to restrict users to intended sections of the file system.

On the `PersistentVolumeClaim` object, the `namespace` is set to `netapp`, which corresponds to the Kubernetes namespace to which you want to grant access. The `volumeName` and `storage` fields must correspond with the name and capacity (must be less than or equal to) of the PV, but the `name` of the PVC itself is arbitrary.

In the same file, define a PV and PVC for the `shared-scratch` directory.

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: netapp-shared-scratch
spec:
  accessModes:
    - ReadWriteMany
  capacity:
    storage: 100Ti
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: beegfs.csi.netapp.com
    volumeHandle: beegfs://192.168.3.100/k8s/namespaces/netapp/shared-scratch
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-scratch
  namespace: netapp
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 100Ti
  storageClassName: ""
  volumeName: netapp-shared-scratch
```

Apply the new PVs and PVCs as follows:

```
jmccormi-mac-0:static-pvcs jmccormi$ kubectl apply -f netapp.yml
persistentvolume/netapp-raw-datasets created
persistentvolumeclaim/raw-datasets created
persistentvolume/netapp-shared-scratch created
persistentvolumeclaim/shared-scratch created
```

Note: To make updates to the objects, rerun the above command.

To verify that the objects were successfully created, run the following commands:

```
jmccormi-mac-0:static-pvcs jmccormi$ kubectl get pvc -n netapp
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
raw-datasets	Bound	netapp-raw-datasets	500Ti	RWX		9s
shared-scratch	Bound	netapp-shared-scratch	100Ti	RWX		9s

Lastly, in this cluster, there are some existing datasets in BeeGFS that non-Kubernetes users are using that shouldn't be duplicated. Specifically, there is an ImageNet dataset often used for experimentation or benchmarking purposes. Create a static PV/PVC granting access to Kubernetes users in the NetApp namespace at the current location so the existing user workflows are not disrupted. As when creating the previous PV/PVCs, the desired Kubernetes objects are described by using a YAML manifest file and applied by using `kubectl apply -f <file>`.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: imagenet
spec:
  accessModes:
    - ReadWriteMany
  capacity:
    storage: 150Gi
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: beegfs.csi.netapp.com
```

```

    volumeHandle: beegfs://192.168.3.100/datasets/imagenet/
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: imagenet
  namespace: netapp
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 150Gi
  storageClassName: ""
  volumeName: imagenet

```

Subsequent sections will cover how users of an MLOps platform can gain access to these PVCs.

Enabling users to dynamically provision BeeGFS storage on-demand

It is likely that users will want to dynamically provision storage whenever they need it; for example, when creating Jupyter notebook servers with persistent storage for the user's workspace, or other data volumes they might need. In Kubernetes, this is achieved by configuring a [storage class](#) used to dynamically provision PVs whenever users submit new PVCs. The storage class is defined as a Kubernetes manifest file. The key parameters are described below:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-scratch
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: beegfs.csi.netapp.com
parameters:
  sysMgmtHost: 192.168.3.100
  volDirBasePath: k8s/dyn/fast
  # Optionally configure the default stripePattern parameters.
  stripePattern/storagePoolID: "3"
  # stripePattern/chunkSize: 512k
  # stripePattern/numTargets: "4"
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: false

```

You can set the `name` based on your preferred way to differentiate multiple storage classes available in the same cluster, for example, bronze, silver, gold, or archive. Your users can then select the class that is most suitable for them. Optionally use an annotation to make this the default storage class in the cluster. The `provisioner` must be `beegfs.csi.netapp.com`.

Note: Kubernetes does not prevent more than one default storage class. To change which class is the default, remove the annotation from the current default.

Under `parameters`, the `sysMgmtHost` indicates the IP or host name of the BeeGFS management server for the desired file system. The `volDirBasePath` indicates the parent directory where new subdirectories are created for each PVC, with user access restricted to the directory representing their PVC. The driver automatically creates this directory (and any parent directories) if it does not exist.

Note: Multiple Kubernetes clusters sharing the same `volDirBasePath` is not a recommended configuration. If multiple clusters need to use the same BeeGFS cluster, rename the `k8s` portion of the path to reflect the cluster name.

To control which BeeGFS storage pool is used, specify `stripePattern/StoragePoolID` (this example uses the EF600 storage pool from above). To further optimize BeeGFS storage classes for [specific workloads](#), use the `stripePattern/chunkSize` and `stripePattern/numTargets` parameters. Otherwise, these values are inherited from the parent directory.

Setting a `reclaimPolicy` of `Delete` ensures that if the PVC is deleted, the PV and its underlying BeeGFS directory will also be deleted. Although this setting allows automatic cleanup of unneeded storage space, users should be made aware that their files are deleted if they delete their PVC. For more information and alternative settings, see the [Kubernetes documentation](#). In the 1.0 version of the NetApp BeeGFS CSI driver, you can set `volumeBindingMode` to `Immediate` because the driver is not topology aware, and you should set `allowVolumeExpansion` to `false` because the driver has no concept of capacity. If you are using a different version, see the [BeeGFS CSI driver documentation](#).

When ready, apply the new storage class.

```
jmccormi-mac-0:storage-classes jmccormi$ kubectl apply -f sc-fast-scratch.yml
storageclass.storage.k8s.io/fast-scratch created
```

To verify that the storage class was successfully created and set, run the following commands:

```
jmccormi-mac-0:storage-classes jmccormi$ kubectl get sc
NAME                PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
archive             beegfs.csi.netapp.com   Delete          Immediate            false                  5d
fast-scratch (default) beegfs.csi.netapp.com   Delete          Immediate            false                  4s
```

Kubeflow

Overview

Kubeflow is a complete ML toolkit platform for Kubernetes that includes functionality enabling data science and MLOps, including:

- Rapid experimentation using Jupyter notebooks
- Tuning model hyperparameters using Katib
- Continuous integration and deployment (CI/CD) for ML using Kubeflow Pipelines

Users are also able to leverage storage and GPUs provided by the underlying Kubernetes infrastructure where needed. For a complete list of Kubeflow's functionality, visit the [Kubeflow](#) website. If you already have Kubeflow deployed, you can skip to the section "Using BeeGFS with Jupyter notebooks as the workspace for data science."

Note: To avoid broken links, most links will point at the documentation for the version of Kubeflow we're currently using (v1.2). The latest Kubeflow documentation can be accessed [here](#), or after navigating to each link by clicking the version drop down at the top of the page.

Deployment

Kubeflow can be deployed [on-premises or in the cloud](#), which makes it an ideal option for customers implementing a hybrid-cloud model. Kubeflow is constantly being improved and updated; therefore, instead of listing the step-by-step deployment steps, this section provides a high-level overview of the deployment process and links to the most recent documentation. Storage-specific configurations or caveats are also noted.

Before you deploy Kubeflow, set a default storage class in Kubernetes. This setting is already specified in the BeeGFS storage class manifest example shown above. If you're not sure whether a default storage class is set, run: `kubectl get sc` and verify that `(default)` is listed by one (and only one) storage class. If needed, [patch an existing storage class](#) to make it the default, then verify.

```
kubectl patch storageclass <STORAGE_CLASS> -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
kubectl get sc
```

Note: In our environment, Kubeflow was entirely backed by BeeGFS, including a MySQL Database and MinIO object storage server.

When you're ready to deploy Kubeflow, NetApp recommends using the Kubeflow deployment tool provided by [NVIDIA DeepOps](#), particularly if you are planning to use NVIDIA GPUs. Alternatively, you can deploy Kubeflow manually by using the `kfctl` tool (follow the instructions in the [official documentation](#)). Either way, there are two primary deployment methods for existing Kubernetes clusters:

- [Vanilla Kubeflow deployment](#)
- [Multi-user auth-enabled \(Dex\)](#)

By default, both options deploy Istio (a service mesh), which can be disabled if Istio is already installed. In our environment, we chose the multiuser auth-enabled option using Dex to provide LDAP/AD integration (follow-on sections of this document reflect this). If you choose this option, make sure you refer to the [Kubeflow instructions](#) for setting up authentication post-deployment.

Notes on deploying Kubeflow

The following sections are meant to compliment the Kubeflow documentation and provide additional guidance on deploying Kubeflow v1.2 (based on our experience). For current recommendations and best practices, always refer to the latest Kubeflow documentation.

Deploying Kubeflow in an air-gapped environment

As of Kubeflow v1.2, there is minimal documentation on how to deploy Kubeflow in an air-gapped environment, although the process is fairly straightforward. The deployment process does require an internal Docker registry mirror that can pull the images Kubeflow requires from gcr.io, quay.io, and docker.io. If you have this setup, follow these steps on a server/local machine with `kubectl` access to the Kubernetes cluster you wish to deploy Kubeflow to:

1. Download the [version of the kfctl](#) tool that corresponds with the version of Kubeflow you wish to deploy to a new `$HOME/kf-install` directory.
Note: You can also add this binary to a directory specified by `$PATH` negating the need to specify the path to run `kfctl` in subsequent steps.
2. Download the [Kubeflow manifests](#) for the version of Kubeflow you want to deploy to `$HOME/kf-install/` directory.
3. Extract the manifests to a new `$HOME/kf-install/<KF-DEPLOYMENT>/manifests` directory where `<KF-DEPLOYMENT>` is the name of the Kubeflow deployment (this is arbitrary).
4. For all files in the `manifests` directory, find and replace the external registries `docker.io`, `gcr.io`, `quay.io` with the name of your internal registry (for example: `docker.repo.eng.netapp.com`).
5. From the `manifests` directory, under `kfdef/`, copy either the `kfctl_istio_dex.yaml` or `kfctl_k8s_istio.yaml` configuration YAML file (depending on whether you plan to use Dex for authentication) to `kf-install/<KF-DEPLOYMENT>/<FILE>.yaml`.
6. At the bottom of the configuration YAML file, update the `uri` to point the full path to `kf-install/<KF-DEPLOYMENT>/manifests` and the `version` to reflect the manifest version.

For example:

```
repos:
- name: manifests
  uri: /home/user/kf-install/kf-ntap/manifests
  version: v1.2.0
```

7. To deploy Kubeflow, with the working directory set to `kf-install/<KF-DEPLOYMENT>`, run: `kf-install/kfctl apply -V -f <KF_CONFIG_FILE>.yaml` where `<KF_CONFIG_FILE>` is the name of the configuration file you copied from `kfdef`.

For example:

```
user@k8s-01:~/kf-install/kf-ntap$ $HOME/kf-install/kfctl apply -V -f kfctl_istio_dex.yaml
```


This is an example of the final directory structure:

```
user@k8s-01:~/kf-install$ tree
.
├── kfctl
├── kfctl_v1.2.0-0-gbc038f9_linux.tar.gz
├── kf-ntap
│   ├── kfctl_istio_dex.v1.2.0.yaml
│   ├── kustomize
│   │   └── application
│   │       └── kustomization.yaml
│   └── [...]
└── manifests
    └── [...]
```

Note: The `kustomize` directory is created by `kfctl`.

Enabling access to Kubeflow

After you successfully deploy Kubeflow, some additional network configurations are required to access the Kubeflow dashboard. Depending on how you deployed Kubeflow, see the following sections:

- [Vanilla Kubeflow deployment](#)
- [Multi-user auth-enabled \(Dex\)](#)

In both cases, properly exposing Kubeflow to the outside world requires a load balancer. If needed, see the Kubernetes deployment section for more details. If you are using the multi-user auth-enabled documentation for exposing Kubeflow, there are a few additional steps. Update the Kubeflow Istio Gateway to expose port 443 (HTTPS) and ideally make port 80 redirect to 443. The default example uses a wildcard (*) for the host; however, if you have (or plan to have) multiple applications running in the same cluster, you can also specify a host name when editing the object as follows:

```
kubectl edit -n kubeflow gateways.networking.istio.io kubeflow-gateway

[...]

spec:
  selector:
    istio: ingressgateway
  servers:
  - hosts:
    - 'kubeflow.netapp.com' # Update as needed.
    port:
      name: http
      number: 80
      protocol: HTTP
      # Upgrade HTTP to HTTPS
      tls:
        httpsRedirect: true
  - hosts:
    - 'kubeflow.netapp.com' # Update as needed.
    port:
      name: https
      number: 443
      protocol: HTTPS
      tls:
        mode: SIMPLE
        privateKey: /etc/istio/ingressgateway-certs/tls.key
        serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
```

Complete the remaining steps to [expose using a LoadBalancer](#), then you can access the Kubeflow dashboard at the host name (for example, <https://kubeflow.netapp.com>).

Managing multiple users, projects, and teams in Kubeflow

By default, when you first log on, you are prompted to create a namespace, which defaults to your username (for example, joe-mccormick). You can then add other users (as desired) to your namespace for collaboration. You can also create namespaces for specific projects or teams by configuring [static users](#) in Kubeflow (technically as part of configuring Dex). In our environment, we used the default admin user to create a `netapp` namespace and then we added other users.

Note: Automatic namespace creation can [optionally be disabled](#).

Namespaces in Kubeflow are powerful because they restrict access to Jupyter notebook servers, pipelines, and other Kubeflow features to the users in that namespace. Because each Kubeflow namespace correlates with a Kubernetes namespace, they also grant selective access to Kubernetes objects such as PVCs. This means administrators can use static PVCs to expose directories in BeeGFS to specific users who require access. It also means that if a user requests on-demand storage space (such as dynamically provisioned PVCs using a storage class), other users in the same namespace can immediately share access.

Administrators can also restrict the types of resources that users are able to request within their namespaces. For example, an administrator might want to prevent users from reserving GPUs in a private namespace when creating Jupyter notebook servers, and only allow GPU reservations in a shared namespace, such as `netapp`, to promote collaboration and provide better visibility into reservations of these shared resources. For existing users, this can easily be done by editing their profile with `kubectl edit profile <namespace>`, which opens the preferred editor (Vim on many systems) and changes can be made and immediately saved.

```
apiVersion: kubeflow.org/v1
kind: Profile
metadata:
  [...]
spec:
  owner:
    kind: User
    name: Joe.McCormick@netapp.com
  resourceQuotaSpec:
    hard:
      cpu: "2"
      memory: 2Gi
      requests.nvidia.com/gpu: "0"
      persistentvolumeclaims: "1"
      requests.storage: "5Gi"
```

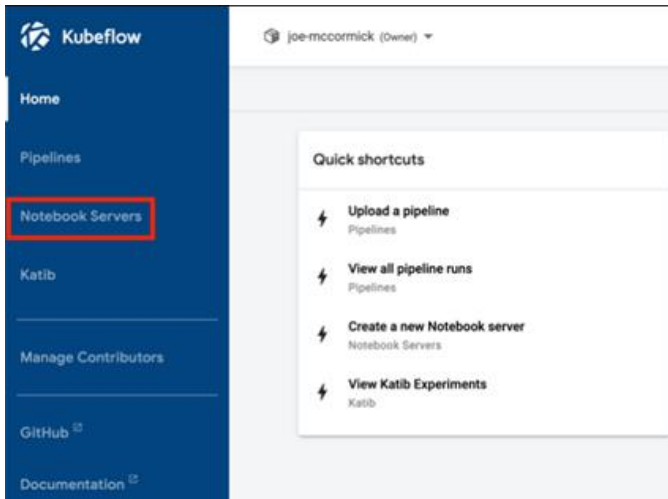
In a production environment, NetApp recommends that user profiles be saved as Kubernetes manifest files stored in version control and applied by using Kubernetes [declarative object configuration](#) management. This recommendation better supports Kubeflow's ability to allow [manual profile creation](#), which accelerates the onboarding of new users.

Note: For details on multitenancy, see the [Kubeflow documentation](#). There is a one-to-one correspondence of profiles with Kubernetes namespaces, so the terms are often used interchangeably in Kubeflow documentation.

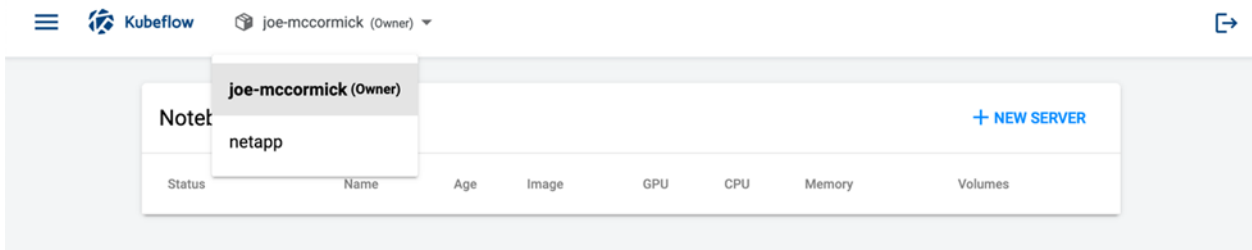
Using BeeGFS with Jupyter notebooks as the workspace for data science

This section describes how BeeGFS can be used to back the working environment used by a team of data scientists. In this scenario, a new team member wants to set up a Jupyter notebook server in the NetApp namespace.

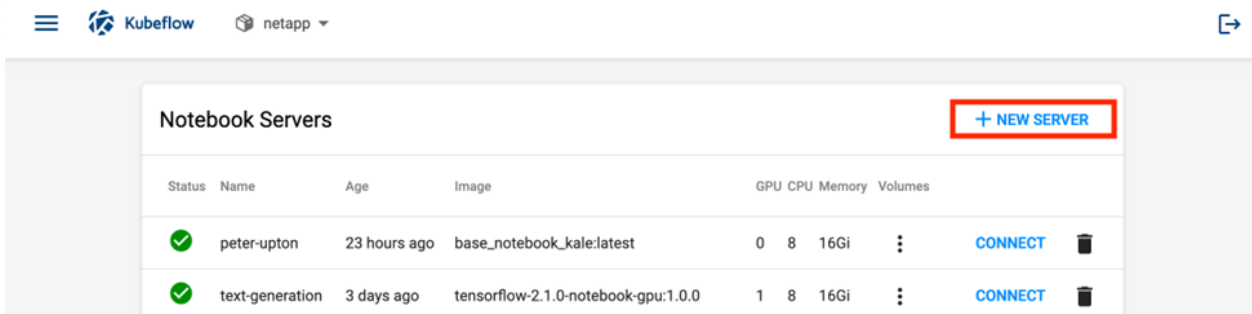
1. After logging into Kubeflow, select `Notebook Servers` from the left menu (clicking the hamburger button at the top left if needed to expose the navigation).




2. Because the administrator already added them to the NetApp namespace, select `netapp` from the drop-down menu.



3. Any existing notebooks for the team are displayed. Click New Server to create a Jupyter notebook server.



4. Provide the desired primary configuration for the Jupyter notebook server.

 **Name**


Specify the name of the Notebook Server and the Namespace it will belong to.

Name

joe-mccormick

Namespace

netapp


 **Image**

A starter Jupyter Docker Image with a baseline deployment and typical ML packages.

☒ Custom Image

Custom Image

docker.repo.eng.netapp.com/peteru/base_notebook_kale:latest

 **CPU / RAM**


Specify the total amount of CPU and RAM reserved by your Notebook Server. For CPU-intensive workloads, you can choose more than 1 CPU (e.g. 1.5).

CPU

8

Memory

32.0Gi

 **GPUs**


Specify the number and Vendor of GPUs that will be assigned to the Notebook Server's Container.

Number of GPUs

1

GPU Vendor

NVIDIA

 **Workspace Volume**

These notebook servers are based on a Jupyter Docker image; but users can optionally provide a custom image. For security purposes, our data science environment is air-gapped. In other words, any internet access is provided through proxies. To ease onboarding of new users, custom Docker images are preconfigured to work inside this environment with common dependencies already in place. For example, setting up a custom Python pip configuration to use the internal mirror, adding configuration to allow `pip install` commands to work normally, or preinstalling NVIDIA GPU libraries. In our environment users can still pull any Docker image from various internet registries including Docker Hub, Google Cloud, Quay, and NVIDIA NGC.

As part of this workflow, you can specify CPU/memory requirements and request GPU reservations. Our environment includes NVIDIA A100 GPUs, so using the [multi-instance GPU](#) (MIG) feature each GPU can be partitioned into as many as seven instances. This feature allows administrators to allocate each data scientist part of a GPU, accelerating initial exploration in their Jupyter notebooks before scaling to full production training. Behind the scenes, Kubeflow works with Kubernetes to ensure the Jupyter notebook is scheduled to a node with available GPUs.

Scroll down to specify your desired storage configuration (Figure 3).

Figure 3) Jupyter notebook server creation (storage options).

The screenshot shows the 'Workspace Volume' section with a sub-header 'Configure the Volume to be mounted as your personal Workspace.' Below this is a checkbox 'Don't use Persistent Storage for User's home' which is unchecked. There are five input fields: 'Type' (New), 'Name' (workspace-joe-mccormick), 'Size' (10Gi), 'Mode' (ReadWriteOnce), and 'Mount Point' (/home/jovyan). Below this is the 'Data Volumes' section with a sub-header 'Configure the Volumes to be mounted as your Datasets.' There is a '+ ADD VOLUME' button. Below the button are three rows of volume configuration, each with 'Type', 'Name', and 'Mount Point' fields, and a red trash icon to the right. The first row has Type: Existing, Name: shared-scratch, Mount Point: /yan/shared-scratch. The second row has Type: Existing, Name: raw-datasets, Mount Point: ovyan/raw-datasets. The third row has Type: Existing, Name: imagenet, Mount Point: ne/jovyan/imagenet.

In this example, the user is requesting space in BeeGFS for the workspace's home directory that will be dynamically provisioned by the default storage class that we set up earlier. By default, this space persists even if the Jupyter notebook server is deleted, meaning users can easily migrate to a new notebook server configuration if needed. The user is also requesting access to the shared spaces we set up earlier for collaboration, and the ImageNet volume.

Scroll down further to see the options to specify an Affinity, Tolerations, and extra layers of Configuration that can be applied (Figure 4). In our environment, these options are not currently being used. At the bottom of the page, there is an option to enable shared memory. This option allows libraries such as PyTorch to use shared memory for multiprocessing. When you are ready, click **Launch** to create the new notebook server.

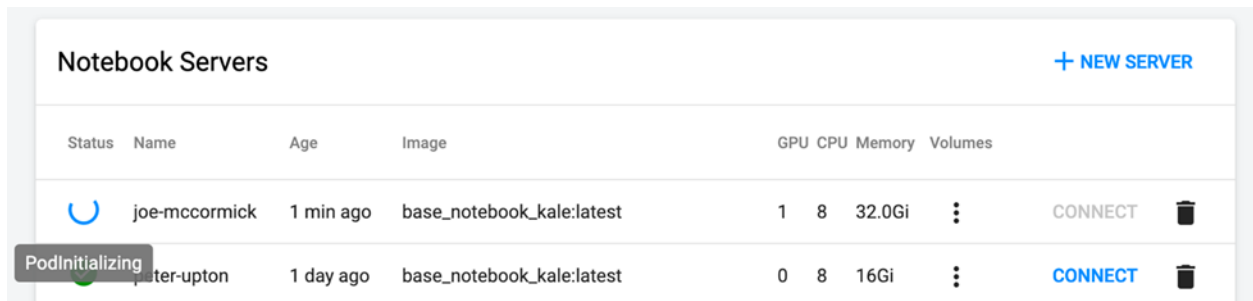
Figure 4) Jupyter notebook server creation (miscellaneous settings).

The screenshot shows the 'Miscellaneous Settings' section with a sub-header 'Other possible settings to be applied to the Notebook Server.' Below this is a toggle switch 'Enable Shared Memory' which is turned on. At the bottom are two buttons: 'LAUNCH' and 'CANCEL'.

Note: For the latest details about setting up Jupyter notebooks, see the [Kubeflow documentation](#).

A spinning icon is displayed as Kubernetes works to create the requested resources (Figure 5). Hover over the Loading icon to see the current status of the request.

Figure 5) Jupyter notebook server provisioning in progress.



Notebook Servers								+ NEW SERVER
Status	Name	Age	Image	GPU	CPU	Memory	Volumes	
	joe-mccormick	1 min ago	base_notebook_kale:latest	1	8	32.0Gi		CONNECT
	joe-mccormick	1 day ago	base_notebook_kale:latest	0	8	16Gi		CONNECT

When the provisioning is complete, a green checkbox is displayed. Click Connect.

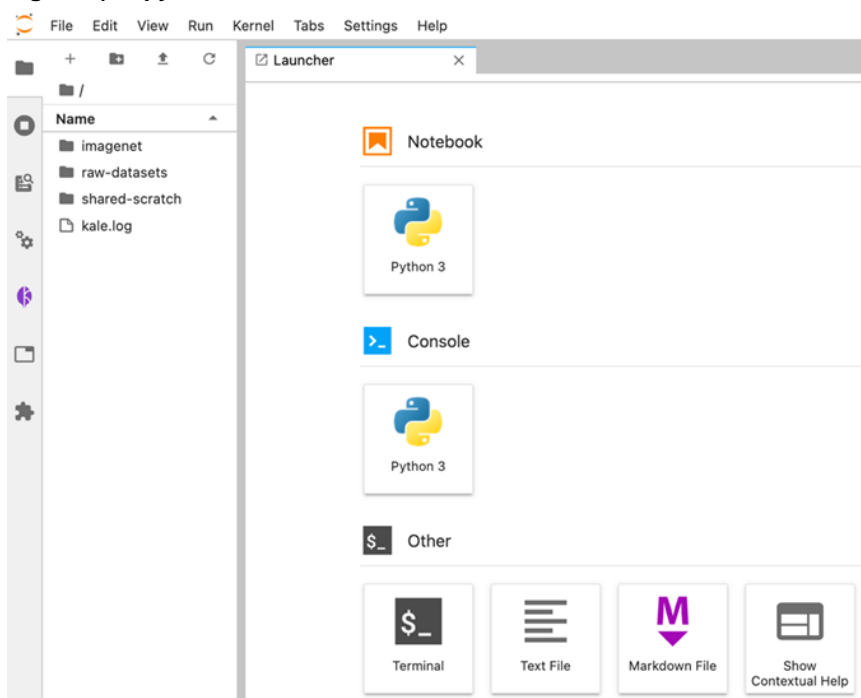


	joe-mccormick	3 mins ago	base_notebook_kale:0.1	1	8	32.0Gi		CONNECT
--	---------------	------------	------------------------	---	---	--------	--	-------------------------

Note: Clicking on Connect results in a `No Healthy Upstream` error. In some instances, waiting a few minutes allows the Jupyter notebook to finish starting. If the issue persists, then further troubleshooting might be required. Run `kubectl describe` and `kubectl logs` on the Pod in question, or refer to your observability platform (such as Prometheus). Common issues include [improperly configured custom Docker images](#) that don't properly start Jupyter after `docker run`.

At this time, you should have access to a Jupyter notebook.

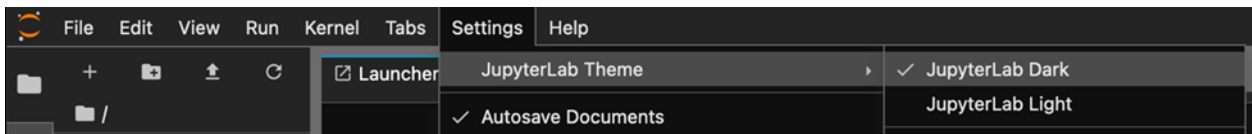
Figure 6) Jupyter notebook server launcher.



In our environment, we typically based our Jupyter notebook servers on the notebook image provided by the Kubeflow Automated Pipelines Engine ([KALE](#)) project, which includes the Kale JupyterLab extension and the recommended version of JupyterLab (which might look different from what you're using). Kale provides more seamless integration with Kubeflow Pipelines, so starting with that notebook container can make it easier to transition from development to production.

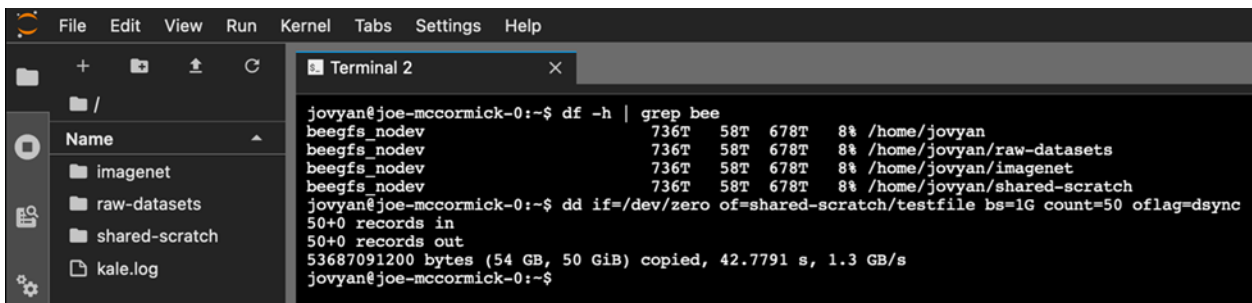
If desired, set up dark mode by selecting: Settings > JupyterLab Theme > JupyterLab Dark.

Figure 7) JupyterLab theme.



At this point, you have a complete development environment. The data volumes that were requested from the server creation page are conveniently mounted to the workspace for easy access. Behind the scenes, BeeGFS is providing enough capacity and performance to comfortably support Jupyter. You can verify this by using a terminal, as shown in Figure 8.

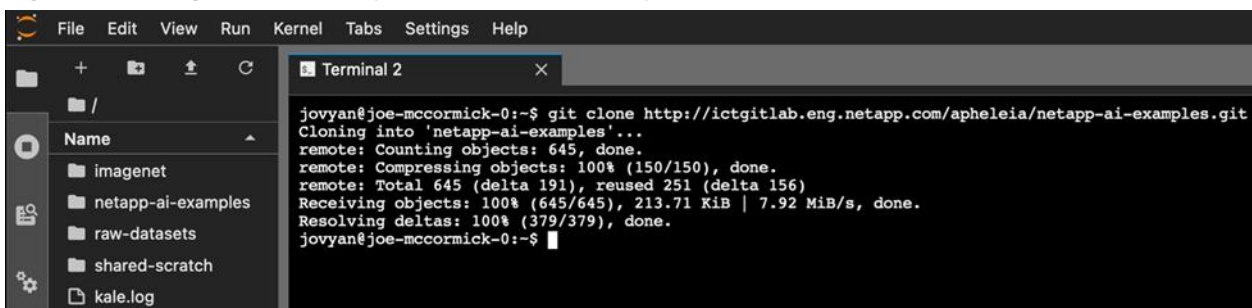
Figure 8) Verifying BeeGFS capacity and performance from a terminal in Jupyter.



Note: Demonstrating performance is not the intent of this report. The above test is meant to provide a basic idea of what a user can expect from BeeGFS, even with a single-threaded application. This test is not optimized or even capable of achieving the maximum theoretical performance of the BeeGFS filesystem.

The examples in this report are stored in a Git repository. They demonstrate one workflow that the data scientists can use to version and collaborate their work. Figure 9 shows how to clone a Git repository from a Jupyter terminal.

Figure 9) Cloning a Git repository from a terminal in Jupyter.



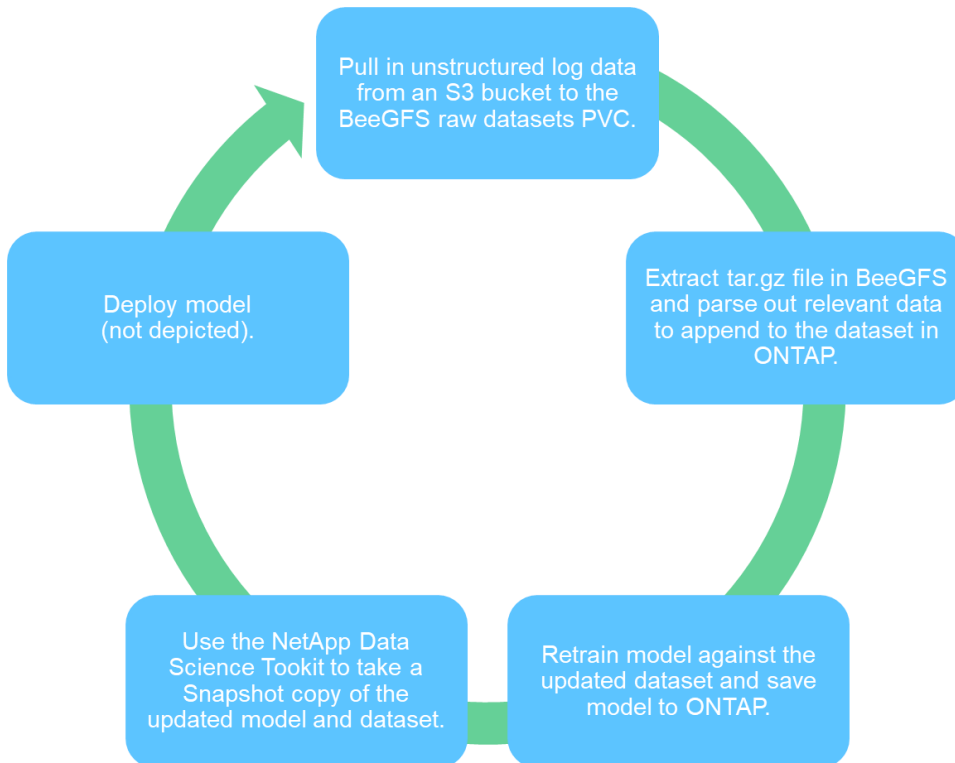
At this point, we finished setting up an environment that a team member can use for experimentation. They now have access to plenty of scratch space and the existing datasets. The following sections demonstrate how to use this environment to develop Kubeflow Pipelines when you're ready to move from model development to production.

Building Kubeflow Pipelines using NetApp storage

Overview

This section describes how to use NetApp and [Kubeflow Pipelines](#) to support production MLOps workflows. Although this scenario is based on the requirements of an actual internal project, each component can be easily interchanged or swapped depending on the individual project requirements. Figure 10 is the example workflow we'll be demonstrating.

Figure 10) Example Kubeflow Pipelines workflow.



Note: The actual data science code is omitted to allow the reader to focus on understanding how NetApp solutions wrap around the data science to deliver a complete MLOps solution.

The pipeline is developed in a Jupyter notebook that interacts with the Kubeflow Pipelines service to upload and execute the pipeline for testing. When the pipeline is complete, it will be uploaded to the Kubeflow Pipelines UI, which allows it to be reused to facilitate multiple ML workflows.

Prerequisites

As of Kubeflow v1.2, if you deploy using the multiuser authentication enabled setup, additional configurations are required to enable access to the Kubeflow Pipelines API from within a Jupyter notebook.

Note: This setup is not required in future versions of Kubeflow. For the latest updates on related enhancements, see this [GitHub issue](#).

Set up [Istio Role-Based Access Control](#) (RBAC) to allow notebook servers in a specific namespace to access the `ml-pipeline` service. Create a YAML file and replace `<NAMESPACE>` with the following text as needed. Then apply with `kubectl apply -f <FILE>`.


```

apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: bind-ml-pipeline-nb-<NAMESPACE>
  namespace: kubeflow
spec:
  roleRef:
    kind: ServiceRole
    name: ml-pipeline-services
  subjects:
  - properties:
    source.principal: cluster.local/ns/<NAMESPACE>/sa/default-editor

```

For each notebook in that namespace, you need to interact with the Kubeflow Pipelines API, therefore, you need to set up an Envoy filter to inject the `kubeflow-userid` header so the ML Pipeline API server can validate the incoming request. Create the following YAML file and update the placeholder text, as described below:

```

apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: add-header
  namespace: <NAMESPACE>
spec:
  configPatches:
  - applyTo: VIRTUAL_HOST
    match:
      context: SIDECAR_OUTBOUND
      routeConfiguration:
        vhost:
          name: ml-pipeline.kubeflow.svc.cluster.local:8888
          route:
            name: default
    patch:
      operation: MERGE
      value:
        request_headers_to_add:
        - append: true
          header:
            key: kubeflow-userid
            value: <USERNAME@DOMAIN>
  workloadSelector:
    labels:
      notebook-name: "<NOTEBOOK-NAME>"

```

Update these placeholders with the appropriate values for your environment:

- `<NAMESPACE>` to match the namespace the notebook server is in.
- `<USERNAME@DOMAIN>` to reflect the email of the user that created the namespace.
 - If you're not sure, run the following command:

```
kubectl get profile <NAMESPACE> '-o=custom-columns=OWNER:spec.owner.name'
```

- `<NOTEBOOK-NAME>` to the name of the desired Jupyter notebook.

Lastly deploy the configuration by running `kubectl apply -f <FILE>`.

Add additional Envoy filters as needed for each notebook that requires access to the pipeline's API. To enable access for notebooks in other namespaces, add additional service role bindings and Envoy filters.

Kubeflow Pipeline components

Kubeflow Pipelines are made up of [components](#) that each perform a step in the ML workflow, for example, data preprocessing, data transformation, and training. You can build [lightweight components](#) by using Python functions that run in a Python container (with an option to specify a different base image) [or reusable components](#). Reusable components consist of a Docker image containing any custom

application and a YAML component definition file that describes how the application interacts with the Kubeflow Pipelines system.

Creating reusable components

In this example, we created multiple reusable components. The first component uses [s3cmd](#) to download data from an S3 source such as StorageGRID and save it at a specified output path. To create the component, we created a new directory `s3cmd` containing a single file named `Dockerfile`.

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y s3cmd && rm -rf /var/lib/apt/lists/*
```

Run `docker build . -t <tag>` and push the image to a Docker registry with `docker push <image>`. In the same directory, create a `component.yaml` file updating image to point at the appropriate registry/image:tag.

```
---
# For more examples see https://www.kubeflow.org/docs/pipelines/sdk/component-development/
name: Get data from S3 # optional field
description: A kubeflow component wrapper for the s3cmd tool's get option. # optional
inputs: # For additional details on each paramter see: https://s3tools.org/usage
- {name: bucket} # the bucket to use
- {name: endpoint, default: "s3.amazonaws.com", optional: true} # S3 endpoint.
- {name: keyid} # keyid string
- {name: object} # object to download
- {name: nossl, default: false, type: Boolean, optional: true}
- {name: secretkey} # secret key string
- {name: output_path} # path to save the pulled object to
implementation:
  container:
    image: docker.repo.eng.netapp.com/jmccormi/s3cmd:latest
    command:
      - s3cmd
      - get
      - --debug
      - concat: [s3://,{inputValue: bucket}, /, {inputValue: object}]
      - {inputValue: output_path}
      - concat: [--host-bucket=,{inputValue: endpoint}]
      - concat: [--host=,{inputValue: endpoint}]
      - concat: [--access_key=, {inputValue: keyid}]
      - concat: [--secret_key=,{inputValue: secretkey}]
      - concat: [--skip-existing]
      - if:
        cond: {inputValue: nossl}
        then: --no-ssl
```

The second component uses the [NetApp Data Science Toolkit for Kubernetes](#) to trigger Snapshot copies. For this to work, [Trident](#) must already be installed on your Kubernetes cluster and you must have completed the [setup required for on-demand volume snapshots](#). You also need to set up a cluster role and cluster role binding giving the default-editor service account used by Kubeflow permissions on storage resources including Snapshot copies. Create a YAML manifest file with the following data (replacing the namespace value with yours if needed), then apply by using `kubectl apply -f <file>`.

```
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: ntap-dsutil
rules:
- apiGroups: [""]
  resources: ["persistentvolumeclaims", "persistentvolumeclaims/status", "services"]
  verbs: ["get", "list", "create", "delete"]
- apiGroups: ["snapshot.storage.k8s.io"]
```

```

    resources: ["volumesnapshots", "volumesnapshots/status", "volumesnapshotcontents",
"volumesnapshotcontents/status"]
    verbs: ["get", "list", "create", "delete"]
- apiGroups: ["apps", "extensions"]
  resources: ["deployments", "deployments/scale", "deployments/status"]
  verbs: ["get", "list", "create", "delete", "patch", "update"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: ntap-dsutil
subjects:
- kind: ServiceAccount
  name: default-editor
  namespace: netapp # Replace with desired namespace
roleRef:
  kind: ClusterRole
  name: ntap-dsutil
  apiGroup: rbac.authorization.k8s.io

```

After you complete the initial prerequisites, create a new directory `ntap_dsutil_k8s_snap` for the component. Create a `Dockerfile` that installs the [prerequisites](#) for the Data Science Toolkit, clones in the latest version using Git, then ensures the required file is executable.

```

FROM python:3.7
RUN pip install ipython kubernetes pandas tabulate
CMD mkdir /netapp-data-science-toolkit
WORKDIR /netapp-data-science-toolkit
RUN git clone https://github.com/NetApp/netapp-data-science-toolkit.git .
RUN chmod +x Kubernetes/ntap_dsutil_k8s.py

```

Note: The `Dockerfile` and the following `component.yaml` file were tested with [v1.2](#) of the Data Science Toolkit. Future updates might require adjustments to both.

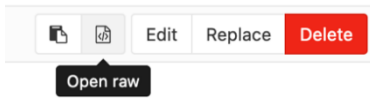
Run `docker build . -t <tag>` and push the image to a Docker registry with `docker push <image>`. Create the `component.yaml` file updating the value for the `image` parameter to point at the appropriate registry/image:tag.

```

---
name: Trigger Snapshot
description: A Kubeflow component wrapper for the NetApp Data Science Toolkit for Kubernetes
snapshot functionality.
inputs:
- {name: pvc}
- {name: namespace}
implementation:
  container: # Reference https://github.com/NetApp/netapp-data-science-
  toolkit/tree/main/Kubernetes#cli-create-volume-snapshot
    image: docker.repo.eng.netapp.com/jmccormi/ntap_data_science_toolkit:latest
    command:
    - python3
    - /root/ntap_dsutil_k8s.py
    args:
    - create
    - volume-snapshot
    - concat: [--pvc=,{inputValue: pvc}]
    - concat: [--namespace=,{inputValue: namespace}]

```

Push the `s3cmd` and `ntap_dsutil_k8s_snap` directories to a new or existing Git repository. You will need the URLs to the plaintext component files in a subsequent step. To obtain these URLs, click Open Raw in the web interface of your Git repository and copy the contents of your browser's address bar.



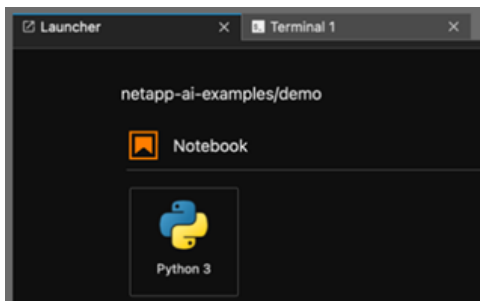
Note: It is possible to host the `component.yaml` files on any HTTP server, or locally on the notebook server, but hosting the files in a Git repository allows for versioning. For examples of additional Kubeflow components, see the [Kubeflow Pipelines Git repository](#).

Creating a Kubeflow Pipeline using a Jupyter notebook

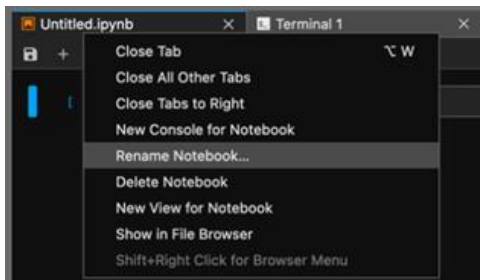
Creating the Jupyter notebook

To create the Jupyter notebook, complete the following steps:

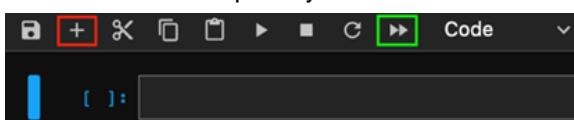
1. Connect to a Jupyter notebook server in a namespace where you want to create the new pipeline. This example uses the server created in the last section under the `netapp` namespace. From the launcher, click `Python 3` under the notebook section.



2. Right-click the notebook tab and select `Rename Notebook`. Enter the desired name and click `Rename`.



3. If you're unfamiliar with Jupyter, there is extensive [documentation](#) and tutorials online. For the purposes of this document, there are only two buttons you need to be familiar with:
 - The button outlined in red adds a new cell, which is a multiline text input field that defaults to Python 3 code (this can be changed using the drop-down menu). You can run the cells individually by using the play button or with `Shift + Enter`.
 - The button highlighted in green restarts the kernel clearing of any variables and prior output (this function requires you to confirm the restart) before rerunning the whole notebook.



4. For each text block in the following section, copy the contents to a new cell in the Jupyter notebook, unless otherwise noted.

Basic setup

1. In the first cell, start by ensuring that the Kubeflow Pipelines API is installed.

```
!pip install --user --upgrade kfp
```

Note: To execute a shell command instead of running the Python code, use “!” at the beginning of the line.

2. Restart the notebook kernel to ensure the pipeline’s API is available.

```
from IPython.core.display import HTML
HTML("<script>Jupyter.notebook.kernel.restart()</script>")
```

3. Set up the imports. Use `kfp` to include the [Kubeflow Pipelines SDK](#). `kfp.dsl` for the [domain-specific language](#) used to define and interact with pipelines and components, and lastly `kfp.components`, which includes classes and methods to interact with [pipeline components](#).

Import `datetime` and set up a `today` variable.

```
import kfp
import kfp.dsl as dsl
import kfp.components as comp
from datetime import date
today = date.today()
```

Defining function-based components

This section describes how to create Python functions that define lightweight components that are used as steps in the pipeline. Each component runs as a Docker container in a Kubernetes Pod; therefore, in order to load packages, complete one (or a combination) of the following tasks:

- Ensure the package is installed in the container image already.
- Define the package by using the `packages_to_install` parameter of the `kfp.components.create_component_from_func(func)` function.
- Install the package as part of the function, for example using the `subprocess` module to call `pip`.

For more information, see the [Kubeflow documentation](#).

Start by creating a function for data preparation. This function demonstrates use of the `subprocess` module to execute the `tar` command to extract data from a tarball at `raw_dataset_path` to `scratch_path`. This tarball contains unstructured data aggregated from multiple NetApp systems, so we must iterate through the data to extract a specific file for each system and add it to the cleaned data.

For demonstrative purposes, this implementation is not idempotent, meaning we don’t take into account whether data is already appended to the output file when the component is rerun. Also, to process gigabytes or terabytes of data, NetApp recommends that you use a reusable component and either write custom functionality in a faster language such as C, Go, or Rust, or use an existing application such as [NetApp XCP](#). However, for many cases, Python works well. The following example is a simple component:

```
def prepare_data(raw_dataset_path, scratch_path, output_dataset_path):

    import subprocess
    import os
    from pathlib import Path

    # Create a directory and extract the log collection:
    Path(os.path.dirname(scratch_path)).mkdir(parents=True, exist_ok=True)
    print(subprocess.run(['tar', 'vxf', raw_dataset_path, '-C', scratch_path]).stdout)
```

```
# Make sure we have a path to store the parsed logs:
Path(os.path.dirname(output_dataset_path)).mkdir(parents=True, exist_ok=True)

with open(output_dataset_path, 'w') as cleaned_data:
    for entry in Path(scratch_path).iterdir():
        if entry.is_dir and not entry.name.startswith('.'):
            try:
                with open(entry / 'state-capture-data.txt') as f:
                    cleaned_data.write(f.read())
            except Exception as e:
                print("Error '{}' reading filename '{}'.format(e, entry / 'state-capture-
data.txt'))
```

Note: The example uses BeeGFS to back the raw dataset and scratch directories. This should provide ample performance for any additional parsing or transformation needed for training.

Define a placeholder function where the code used to actually train the model can be inserted. The function has access to the cleaned dataset at `output_dataset` and outputs the trained model at `output_model`.

This example simply converts the dataset to uppercase then writes it to a file.

```
def train_model(output_dataset, output_model):
    from pathlib import Path
    import os

    Path(os.path.dirname(output_model)).mkdir(parents=True, exist_ok=True)
    with open(output_dataset) as t: # open input text file
        with open(output_model, 'w') as o: # open output text file
            for line in t: # for every line
                o.write(line.upper()) # convert to uppercase
```

Loading Kubeflow reusable and lightweight components

To use the reusable components in pipelines, load them from a local file using `load_component_from_file()` or a URL using `load_component_from_url()`. To use a function-based component, convert it with `func_to_container_op()`. In all three instances, this returns a factory function that you can call to construct a pipeline task (also known as a container operator or [ContainerOp](#)) from that component. In a new cell, add the following text. Replace `base_image` as needed to work in your environment and change the URLs to point to the `component.yaml` files in Git.

```
get_s3_op = comp.load_component_from_url("http://ictgitlab.eng.netapp.com/apheleia/kf-
components/raw/master/s3cmd/component.yaml")

prepare_data_op = comp.func_to_container_op(prepare_data, base_image=' \
    docker.repo.eng.netapp.com/python:3.7')
train_model_op = comp.func_to_container_op(train_model, base_image=' \
    docker.repo.eng.netapp.com/nvidia/tensorflow:21.02-tfl-py3')

trigger_snapshot_op = comp.load_component_from_url("http://ictgitlab.eng.netapp.com/apheleia/kf-
components/raw/master/ntap_dsutil_k8s_snap/component.yaml")
```

Note: For additional ways to create components, see the [Kubeflow Pipeline documentation](#).

Developing the pipeline

In Kubeflow, a [pipeline](#) describes an AI workflow including its various components (or steps) and how they relate to each other. Creating a pipeline involves defining a Python function that specifies any inputs needed to run the pipeline, along with the inputs and outputs of each component in the pipeline.

All code in this section should be inserted into the same cell in your Jupyter notebook. For illustrative purposes, this code is broken up so we can explain each piece.

1. Define the pipeline function signature by using an optional decorator to provide a name and description. The function parameters represent the inputs to the pipeline and are part of what makes it reusable.

```
@kfp.dsl.pipeline( # Optional Decorator
    name='S3-BeeGFS-ONTAP Pipeline',
    description='A pipeline demonstrating the power of Kubeflow pipelines.',
)
def s3_beeGfs_ontap_pipeline (
    namespace: str,          # The Kubernetes namespace where our pipeline is running.
    s3_endpoint: str,        # Hostname:port for the S3 endpoint containing raw data.
                             # Depending on the DNS setup an IP may be needed.
    s3_disable_ssl: bool,    # Optionally disable SSL.
    s3_keyid: str,           # S3 Access key (or username).
    s3_secretkey: str,       # S3 Secret key.
    s3_bucket: str,         # S3 Bucket to use.
    s3_object: str,         # The name of the object we want to download.
    download_dir: str,       # Directory where raw data should be downloaded.
    raw_dataset_pvc: str,    # The name of a PVC we can use to download and manipulate the dataset
    raw_dataset_mount: str,  # The path where our raw dataset PVC will be mounted.
    output_pvc: str,         # The name of a PVC where we should save the final dataset and model.
    output_mount: str,       # The path where our output PVC will be mounted.
    output_dataset: str,     # The path relative to output_mount to save the cleaned dataset.
    output_model: str,       # The path relative to output_mount to save the model.
):
```

Note: To simplify this example, the parameters are storage focused, but you can easily add parameters to further customize each step in the pipeline. For example, you can specify the container image or command to execute for the training step.

2. Define a few variables for convenience.

```
raw_dataset_path = "{}/{}/{}/".format(raw_dataset_mount, download_dir, s3_object)
scratch_path = "{}/{}/{}/{}/".format(raw_dataset_mount, download_dir, today.strftime("%b-%d-%Y"))
output_dataset_path = "{}/{}/{}/".format(output_mount, output_dataset)
output_model_path = "{}/{}/{}/".format(output_mount, output_model)
```

3. The next few steps will define the tasks you want to perform as steps in the pipeline. First, obtain new data from the S3 bucket and save it to a file-system-based PV in Kubernetes for fast access. The parameters used correspond with the parameters defined in the Get Data from S3 `component.yaml` file. The parameter values depend on the S3 provider (such as Amazon, MinIO, or StorageGRID). To attach a PV to the container operation, use the `add_pvolumes()` function. This step requires you to specify the mount point and provide a [PipelineVolume](#), which can reference either an existing PV claim or [one created](#) as part of executing the pipeline. This example presumes that there is an existing PVC that you want to use in the appropriate namespace to download the raw dataset.

```
get_s3_task = get_s3_op(
    keyid=s3_keyid,
    secretkey=s3_secretkey,
    endpoint=s3_endpoint,
    nossl=bool(s3_disable_ssl),
    bucket=s3_bucket,
    object=s3_object,
    output_path=raw_dataset_path # Path to save the raw data.
).add_pvolumes({raw_dataset_mount: dsl.PipelineVolume(pvc=raw_dataset_pvc)})
```

4. The next task prepares the data using the function you created in the previous step, notably mounting the PVC used to download the raw dataset alongside another PVC that stores any additions to the cleaned dataset used to train the model.

```
prepare_data_task = prepare_data_op(raw_dataset_path, scratch_path, output_dataset_path)\
    .add_pvolumes({raw_dataset_mount: dsl.PipelineVolume(pvc=raw_dataset_pvc), \
        output_mount: dsl.PipelineVolume(pvc=output_pvc)})
```

5. Define a single task representing the actual model training. In a real-world use case, this can span multiple tasks, each inputting and outputting data to subsequent steps in the pipeline, leveraging any combination of new or existing PVs based on training requirements. This task [depends on a GPU](#), so use `set_gpu_limit()` to request access to one.

```
train_model_task = train_model_op(output_dataset_path, output_model_path)\
    .add_pvolumes({output_mount: dsl.PipelineVolume(pvc=output_pvc)}) \
    .set_gpu_limit(1, vendor='nvidia')
```

6. This last task triggers a Snapshot copy of the volume on the ONTAP system that backs the PV claim used to store the model and dataset used to train it. As with the `get_s3_task`, the parameters correspond with the ones set in the Trigger Snapshot component .yaml file.

```
trigger_snapshot_task = trigger_snapshot_op(
    pvc="{}".format(output_pvc),
    namespace="{}".format(namespace),
)
```

7. Define the order in which the tasks should run (otherwise they'll attempt to run all at once).

```
prepare_data_task.after(get_s3_task)
train_model_task.after(prepare_data_task)
trigger_snapshot_task.after(train_model_task)
```

At this point, you have completed defining the pipeline.

Testing the pipeline

To test the pipeline, complete the following steps:

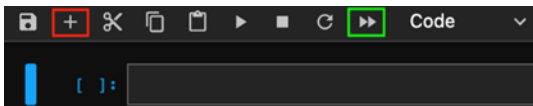
1. Import the Kubeflow Pipeline, then use the `create_run_from_pipeline_func()` [to compile the pipeline function and submit it for execution](#) on Kubeflow Pipelines. These parameters are described in detail above. You can now enter some test values. In a new cell, add the following text:

```
client = kfp.Client() # Create the Kubeflow Pipelines client.

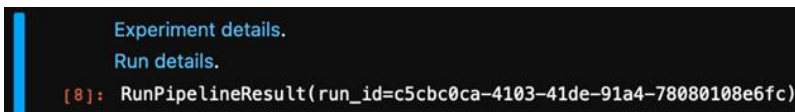
client.create_run_from_pipeline_func(s3_beegfs_ontap_pipeline, arguments={
    "namespace": "netapp",
    "project": "project_01",
    "s3_endpoint": "10.113.204.127:30979",
    "s3_disable_ssl": True,
    "s3_keyid": "keyid",
    "s3_secretkey": "secretkey",
    "s3_bucket": "logs",
    "s3_object": "daily-collection.tar.gz",
    "raw_dataset_pvc": "raw-datasets",
    "raw_dataset_mount": "/beegfs",
    "output_pvc": "project-01-model-dataset",
    "output_mount": "/ontap",
    "output_dataset": "dataset/parsed-logs.txt",
    "output_model": "model/dummy_model",
}, namespace="netapp")
```

Note: To simplify this example, we hardcoded most of the values inside the function call. Do not specify the `secretkey` to a production system here then accidentally commit it to Git. One option is to store it as a [Secret](#) in Kubernetes, then inject it into any component that needs it at runtime using an environmental variable.

2. To run the pipeline, click the button highlighted in green to restart the kernel and run the whole notebook.



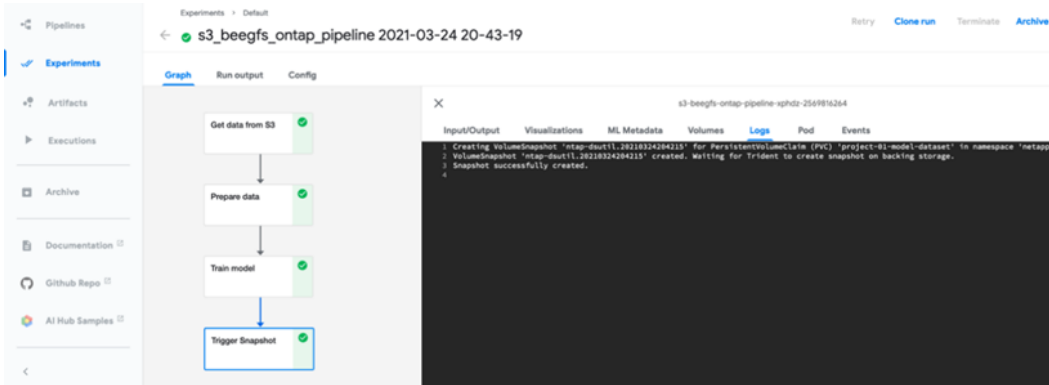
3. If everything goes as expected, under the last cell, two URLs are displayed. Click Run Details.



- A new browser tab/window opens with a graph showing nodes representing each stage in the pipeline, with additional nodes being added as the pipeline progresses.



- When the pipeline completes, a green checkbox is displayed at the top of the page. At any time, you can click the steps in the pipeline to view additional details including input/output, visualizations, logs, and so on. This is particularly valuable if you ever need to troubleshoot why a step in the pipeline failed, without resorting to `kubectl` or other mechanisms for getting at the pod's logs. The following output shows the successful pipeline run with the `Trigger Snapshot` step selected:



Uploading the pipeline for reuse

Although running the pipeline from the Jupyter notebook is an excellent way to test it during development, in the long-term, you should upload it to the [Kubeflow Pipelines UI](#). This allows you to reuse the pipeline to support multiple projects where you can trigger a one-off run or schedule reoccurring runs.

To upload a pipeline for reuse, complete the following steps:

- [Compile](#) the pipeline into a single YAML file that is uploaded to the Kubeflow Pipelines service. Although there are a few ways to do this, the easiest way is to add the following cell to your notebook to create the YAML file and compile and upload it all at once:

```
kfp.compiler.Compiler().compile(s3_beeGfs_ontap_pipeline, 's3_beeGfs_ontap_pipeline.yaml')
client.upload_pipeline('s3_beeGfs_ontap_pipeline.yaml', \
                      pipeline_name="S3-BeeGFS-ONTAP Pipeline", \
                      description="A pipeline demonstrating the power of Kubeflow pipelines.",)
```

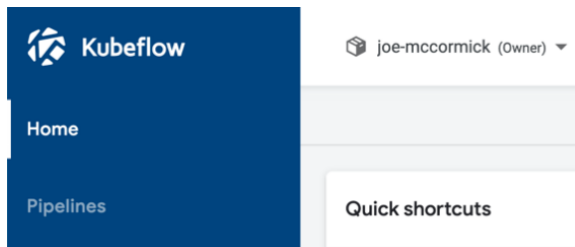
- A link is displayed as well as additional output confirming the pipeline was created successfully.

```
[32]: kfp.compiler.Compiler().compile(s3_beeGfs_ontap_pipeline, 's3_beeGfs_ontap_pipeline.yaml')
      client.upload_pipeline('s3_beeGfs_ontap_pipeline.yaml', \
                           pipeline_name="S3-BeeGFS-ONTAP Pipeline", \
                           description="A pipeline demonstrating the power of Kubeflow pipelines.",)

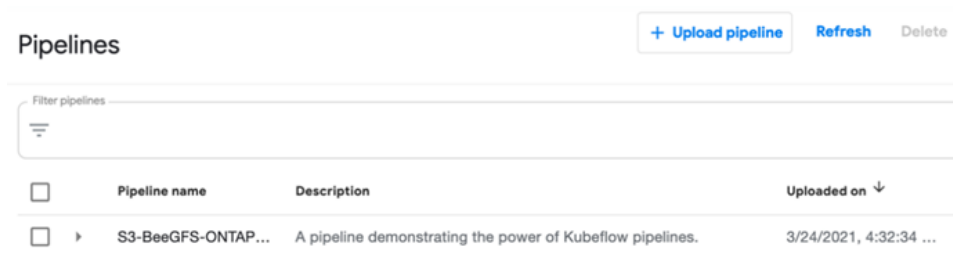
Pipeline details.
[32]: {'created_at': datetime.datetime(2021, 3, 24, 21, 32, 34, tzinfo=tzlocal()),
      'default_version': {'code_source_url': None,
```

Note: Clicking `Pipeline details` allows you to verify that the pipeline was created successfully, but it only provides limited access to the Kubeflow Pipelines interface. For full functionality, navigate to Pipelines through the Kubeflow Dashboard.

3. Navigate to the Kubeflow Dashboard then click Pipelines:



4. The new pipeline is displayed in the list of pipelines. Click the pipeline to see additional details.

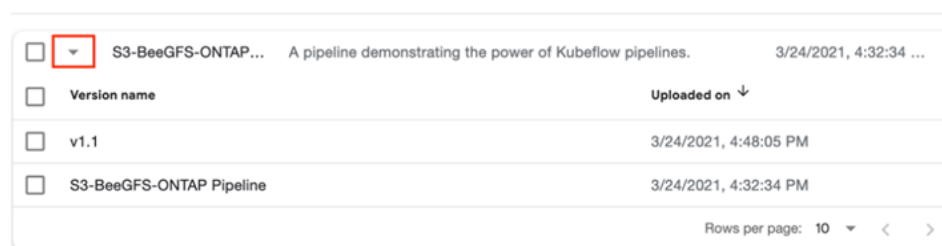


5. After the initial upload, to make changes, either delete the pipeline from the UI or, ideally, use the `upload_pipeline_version()` function instead of `upload_pipeline()`. Modify the original cell and substitute `pipeline_version_name` with the desired version.

```
kfp.compiler.Compiler().compile(s3_bee_gfs_ontap_pipeline, 's3_bee_gfs_ontap_pipeline.yaml')
client.upload_pipeline_version('s3_bee_gfs_ontap_pipeline.yaml', \
    pipeline_name="S3-BeeGFS-ONTAP Pipeline", \
    pipeline_version_name="v1.1",)
```

6. Return to the Kubeflow Pipelines UI and click the button outlined in red to see all available versions.

Figure 11) Viewing versions of a Kubeflow Pipeline.

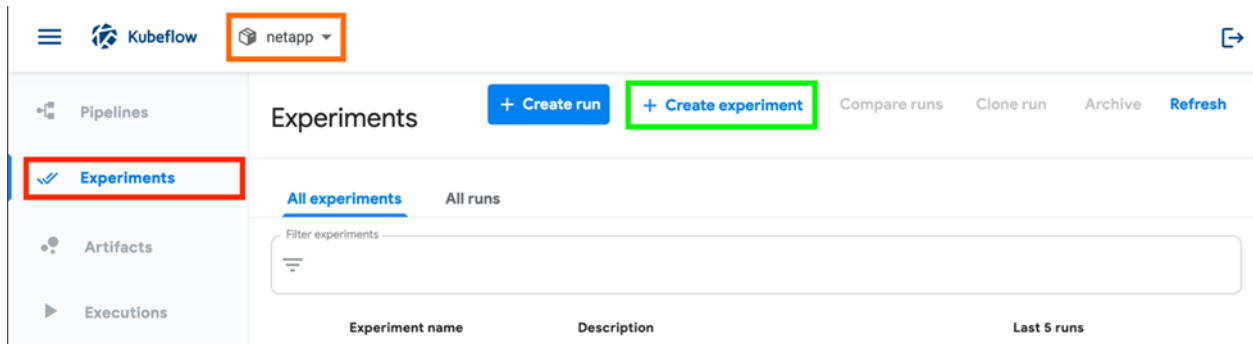


Using the Pipeline

To use the pipeline to support a project, start by creating an [experiment](#). Experiments are essentially a space to contain the history of one or more pipelines and their associated runs. This example creates an experiment and set up a reoccurring run to facilitate the following scenario:

Outside the pipeline, there is a daily process to collect unstructured logs from multiple NetApp storage systems, bundling them into a `tar.gz` file that is uploaded to an S3 bucket for long-term retention. This pipeline is needed to facilitate the following data management tasks for this experiment:

- On a daily basis, download and extract the latest log bundle to the `raw-datasets` PV backed by the BeeGFS set up in the section “Setting up BeeGFS for data science.”
Note: This step also makes the raw data accessible to anyone who wants to experiment with it in Jupyter, or potentially use it as a dependency in other pipelines.
 - Parse only what is needed to train this particular model from the raw dataset and add it to the cleaned dataset in an existing PV backed by ONTAP.
Note: This step prevents storing anything in ONTAP that is not needed while keeping highly efficient Snapshot copies of how the model and dataset have evolved over time.
7. From the Kubeflow Dashboard, click Pipelines, then click Experiments (highlighted in red). Select the desired namespace (highlighted in orange), then click Create Experiment (highlighted in green).



8. On the next page, fill in the details about the experiment, then click Next.

← New experiment

Experiment details

Think of an Experiment as a space that contains the history of all pipelines and their associated runs

Experiment name*

Project 01

Description (optional)

An example experiment representing a ML project.

Next Cancel

9. The next page prompts you to start a [run](#), which defines how a pipeline should be executed either on a one-off or reoccurring basis. Although you can skip this define the runs later, create the run using your pipeline. Select the pipeline you created earlier (with the desired version), then enter a name and description.

Run details

Pipeline *
S3-BeeGFS-ONTAP Pipeline [Choose](#)

Pipeline Version *
v1.1 [Choose](#)

Run name *
Train Dummy Log Model

Description (optional)
Demonstration on how to create a reoccurring run that refreshes a dataset and retrain a model.

This run will be associated with the following experiment

Experiment *
Project 01 [Choose](#)

This run will use the following Kubernetes service account. [?](#)

Service Account (Optional)

10. Scroll down, and under Run Type, select the Recurring option. Enter the [run trigger](#) parameters as desired. In this example, the pipeline is run on a daily basis.

Run Type

☐ One-off ☒ Recurring

Run trigger

Choose a method by which new runs will be triggered

Trigger type *
Periodic ▼

Maximum concurrent runs *
1

☐ Has start date

☐ Has end date

☐ Catchup [?](#)

Run every 1 Days ▼

11. At the bottom of the page, specify the run parameters (this example uses the same values as during earlier testing). Click Start.

Run parameters

Specify parameters required by the pipeline

namespace	netapp
s3_endpoint	10.113.204.127:30979
s3_disable_ssl	True
s3_keyid	keyid
s3_secretkey	secretkey
s3_bucket	logs
s3_object	daily-collection.tar.gz
download_dir	daily-log-bundles
raw_dataset_pvc	raw-datasets
raw_dataset_mount	/beegfs
output_pvc	project-01-model-dataset
output_mount	/ontap
output_dataset	dataset/parsed-logs.txt
output_model	model/dummy_model

[Start](#) [Cancel](#)

12. The subsequent page shows the reoccurring run has been configured. To view details, click Manage.

Experiments

[Refresh](#) [Archive](#)

[← Project 01](#)

Recurring run configs

1 active

[Manage](#)

Experiment description

An example experiment representing a ML p...

Runs

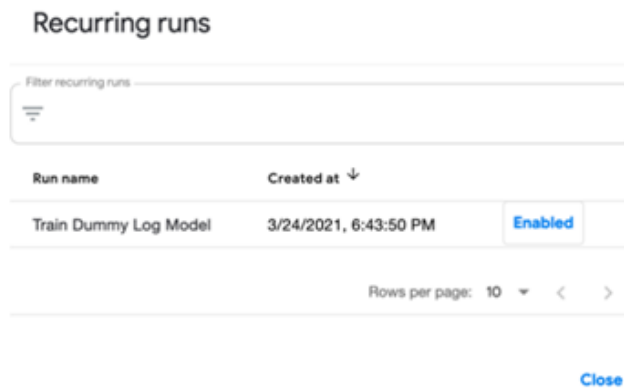
[+ Create run](#) [+ Create recurring run](#) [Compare runs](#) [Clone run](#) [Archive](#)

Filter runs

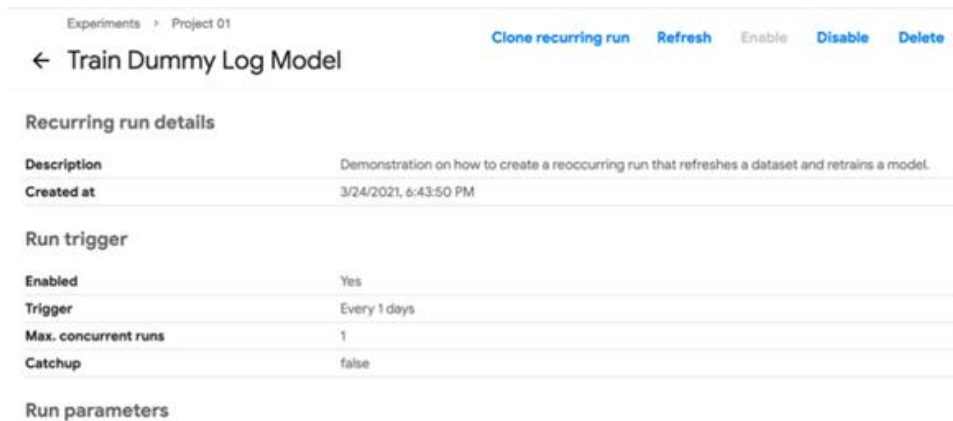
<input type="checkbox"/>	Run name	Status	Duration...	Pipeline Version	Recurring ...	Start time ↓
No available runs found for this experiment.						

Rows per page: 10 < >

13. The Recurring Runs page allows us to disable the run if desired, or access additional options by clicking on the run name.



14. On this page, you can clone the run or delete it.



15. Click ← to return to the Experiment page, then select Create Run. You can use the same run parameters as the reoccurring run, but configure this as a one-off run.

Run details

Pipeline *
S3-BeeGFS-ONTAP Pipeline [Choose](#)

Pipeline Version *
v1.1 [Choose](#)

Run name *
One-Off Training Run

Description (optional)
Demonstration on how to create a one-off run that refreshes a dataset and retrain a model.

This run will be associated with the following experiment

Experiment *
Project 01 [Choose](#)

This run will use the following Kubernetes service account. [?](#)

Service Account (Optional)

Run Type

☒ One-off ☐ Recurring

16. Click Start on the following page. The new run will be displayed.

← Project 01

Recurring run configs

1 active

[Manage](#)

Experiment description

An example experiment representing a ML p...

[Refresh](#) [Archive](#)

[Runs](#)

[+ Create run](#) [+ Create recurring run](#) [Compare runs](#) [Clone run](#) [Archive](#)

Filter runs

☐ Run name Status Duration... Pipeline Version Recurring ... Start time ↓

☐ One-Off Training Run ? - v1.1 - 3/24/2021, 6:57:50 ...

17. When the run successfully completes, the status switches to a green checkbox. To see the detailed graph and the run output, click the run name.

Experiments > Project 01

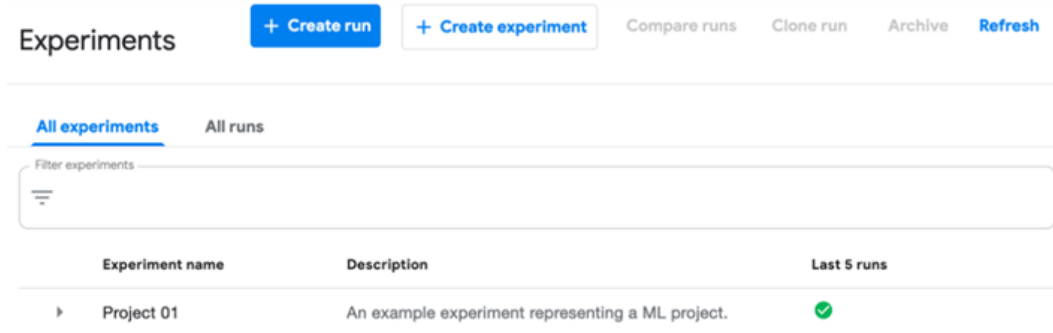
← ☒ One-Off Training Run

[Graph](#) [Run output](#) [Config](#)

Get data from S3

☒

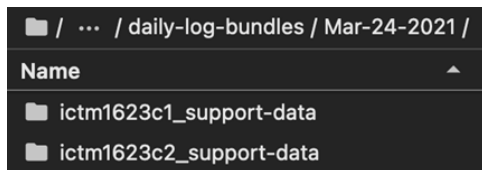
18. In the left navigation pane, click Experiments. From here, you can quickly view the status of the last five runs of your experiments.



Next steps

At this point, you can set up additional experiments using this pipeline, or set up this experiment to use other pipelines. Data scientists can also access the daily raw data collected and organized by the pipeline directly from their Jupyter notebooks, as shown in Figure 12.

Figure 12) Viewing raw data from a Jupyter notebook server.

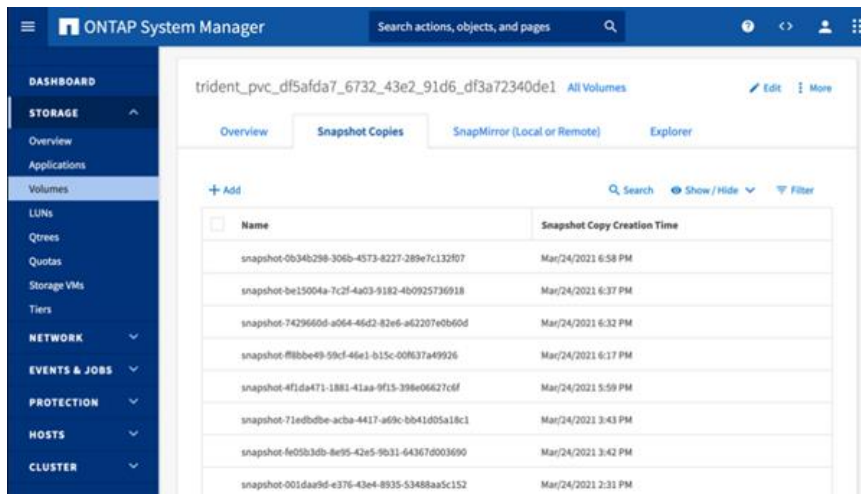


Other users can also browse to the directory in BeeGFS if they have access from another application.

```
jmccormi@ictm1625h1b Mar-24-2021]$ pwd
/mnt/beegfs/k8s/namespaces/netapp/raw-datasets/daily-log-bundles/Mar-24-2021
[jmccormi@ictm1625h1b Mar-24-2021]$ ll
total 1
drwxr-xr-x 2 502 ftp 116 Mar 22 14:28 ictm1623c1_support-data
drwxr-xr-x 2 502 ftp 112 Mar 22 14:28 ictm1623c2_support-data
[...]
```

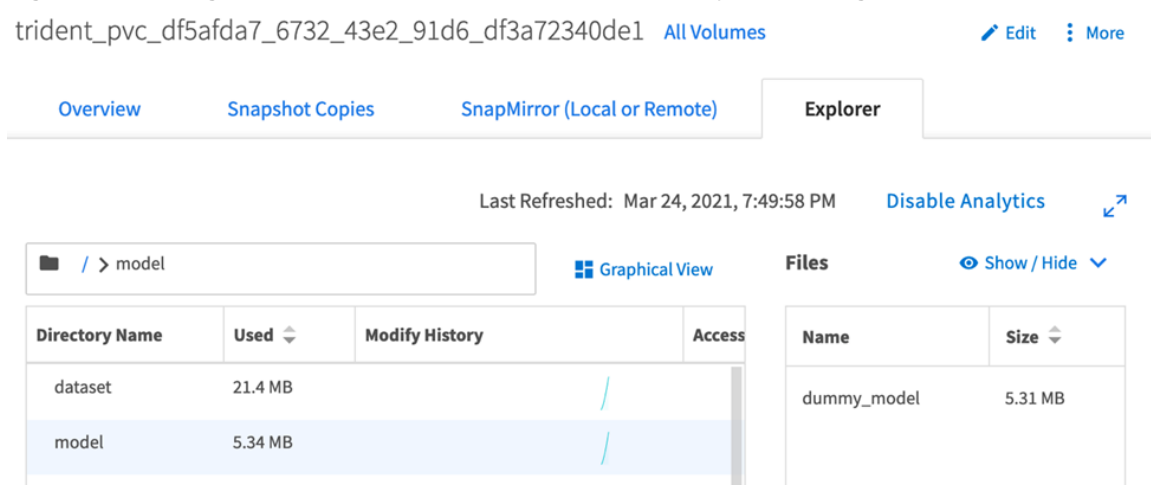
Use ONTAP System Manager to view the volume used to store the final dataset and model and verify the Snapshot copies are being created.

Figure 13) Viewing Snapshot copies in ONTAP System Manager.



You can also verify the contents of the volume directly from System Manager.

Figure 14) Viewing the cleaned dataset and model in ONTAP System Manager.



You're now ready to deploy the model for inferencing in production. Although the exact deployment steps depend on the specific use case, the [data fabric powered by NetApp](#) enables you to deploy the model anywhere in the world across edge, private cloud, and public cloud sites. Kubeflow also includes the [KFServing](#) toolkit for model deployment and serving. This toolkit can integrate with a variety of [other tools](#) that serve ML models in production.

Summary

This section demonstrated how to use Kubeflow Pipelines to orchestrate data movement between S3, parallel file systems, and NFS storage. It also showed you how to use components in Kubeflow to create custom reusable functionality that can be integrated as part of larger data pipelines. This functionality allows you to create workflows that meet the requirements of very simple to complex ML and DL projects.

Conclusion

BeeGFS allows you to provide data scientists the workspace needed to quickly explore and experiment with large amounts of raw unstructured data, while still using familiar development environments such as Jupyter. Leveraging integration with MLOps tools such as Kubeflow, you can design pipelines that automatically refresh the data available to scientists and facilitate retraining of models in production. The example provided demonstrated how you can rely on the entire NetApp portfolio to solve complex challenges around data orchestration:

- Open S3 object stores such as StorageGRID provide web-scale data ingest and access along with control over data placement and replication, providing an effective and affordable option to manage rapidly growing unstructured datasets.
- BeeGFS accelerates day-to-day work, serving as a high-speed workspace for experimentation and providing inexpensive scratch space for data cleaning and preparation. BeeGFS is also well suited for large-scale distributed training where many GPU nodes all need concurrent access to the same files (both large and small).
- ONTAP provides highly efficient versioning of your datasets and models for quick reproducibility, along with the ability to quickly deploy models or share datasets anywhere in the world.

You might have noticed that this document rarely mentioned performance. In our experience, the challenge with AI workloads is not speeds and feeds, but rather the nuances of orchestrating complex data workflows that might need to span the edge, core, and cloud. However, NetApp has partnered with NVIDIA to test and prove that [BeeGFS](#) and [ONTAP](#) are both capable of meeting even the most demanding AI workloads. This provides reassurances that future performance or capacity requirements will never slow you down.

Although you can easily change or leverage additional MLOps platforms, it's beneficial to be more conservative when selecting a storage vendor. The NetApp portfolio provides a wide range of options that work together to ensure the data you need is where you need it, when you need it.

Where to find additional information

To learn more about the information that is described in this document, review the following documents and/or websites:

- BeeGFS Documentation
<https://doc.beegfs.io/latest/>
- BeeGFS CSI Driver Documentation
<https://github.com/NetApp/beegfs-csi-driver/>
- Kubeflow Documentation:
<https://www.kubeflow.org/docs/>
- Kubernetes Documentation:
<https://kubernetes.io/docs/home/>
- NetApp AI Solutions
<https://www.netapp.com/artificial-intelligence/>
- NetApp Product Documentation
<https://www.netapp.com/support-and-training/documentation/>

Version history

Version	Date	Document version history
Version 1.0	May 2021	Initial release.

Refer to the [Interoperability Matrix Tool \(IMT\)](#) on the NetApp Support site to validate that the exact product and feature versions described in this document are supported for your specific environment. The NetApp IMT defines the product components and versions that can be used to construct configurations that are supported by NetApp. Specific results depend on each customer's installation in accordance with published specifications.

Copyright information

Copyright © 2021 NetApp, Inc. All Rights Reserved. Printed in the U.S. No part of this document covered by copyright may be reproduced in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an electronic retrieval system—without prior written permission of the copyright owner.

Software derived from copyrighted NetApp material is subject to the following license and disclaimer:

THIS SOFTWARE IS PROVIDED BY NETAPP "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL NETAPP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NetApp reserves the right to change any products described herein at any time, and without notice. NetApp assumes no responsibility or liability arising from the use of products described herein, except as expressly agreed to in writing by NetApp. The use or purchase of this product does not convey a license under any patent rights, trademark rights, or any other intellectual property rights of NetApp.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

Data contained herein pertains to a commercial item (as defined in FAR 2.101) and is proprietary to NetApp, Inc. The U.S. Government has a non-exclusive, non-transferrable, non-sublicensable, worldwide, limited irrevocable license to use the Data only in connection with and in support of the U.S. Government contract under which the Data was delivered. Except as provided herein, the Data may not be used, disclosed, reproduced, modified, performed, or displayed without the prior written approval of NetApp, Inc. United States Government license rights for the Department of Defense are limited to those rights identified in DFARS clause 252.227-7015(b).

Trademark information

NETAPP, the NETAPP logo, and the marks listed at <http://www.netapp.com/TM> are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.

TR-4890-0521