



Red Hat OpenShift AI Cloud Service 1

Working with distributed workloads

Use distributed workloads for faster and more efficient data processing and model training

Red Hat OpenShift AI Cloud Service 1 Working with distributed workloads

Use distributed workloads for faster and more efficient data processing and model training

Legal Notice

Copyright © 2025 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Distributed workloads enable data scientists to use multiple cluster nodes in parallel for faster and more efficient data processing and model training. The CodeFlare framework simplifies task orchestration and monitoring, and offers seamless integration for automated resource scaling and optimal node utilization with advanced GPU support.

Table of Contents

| | |
|--|-----------|
| PREFACE | 4 |
| CHAPTER 1. OVERVIEW OF DISTRIBUTED WORKLOADS | 5 |
| 1.1. DISTRIBUTED WORKLOADS INFRASTRUCTURE | 5 |
| 1.2. TYPES OF DISTRIBUTED WORKLOADS | 6 |
| CHAPTER 2. PREPARING THE DISTRIBUTED TRAINING ENVIRONMENT | 7 |
| 2.1. CREATING A WORKBENCH FOR DISTRIBUTED TRAINING | 7 |
| 2.2. USING THE CLUSTER SERVER AND TOKEN TO AUTHENTICATE | 9 |
| 2.3. MANAGING CUSTOM TRAINING IMAGES | 10 |
| 2.3.1. About base training images | 10 |
| 2.3.2. Creating a custom training image | 11 |
| 2.3.3. Pushing an image to the integrated OpenShift image registry | 13 |
| CHAPTER 3. RUNNING RAY-BASED DISTRIBUTED WORKLOADS | 15 |
| 3.1. RUNNING DISTRIBUTED DATA SCIENCE WORKLOADS FROM JUPYTER NOTEBOOKS | 15 |
| 3.1.1. Downloading the demo Jupyter notebooks from the CodeFlare SDK | 15 |
| 3.1.2. Running the demo Jupyter notebooks from the CodeFlare SDK | 16 |
| 3.1.3. Managing Ray clusters from within a Jupyter notebook | 19 |
| 3.2. RUNNING DISTRIBUTED DATA SCIENCE WORKLOADS FROM DATA SCIENCE PIPELINES | 23 |
| CHAPTER 4. RUNNING TRAINING OPERATOR-BASED DISTRIBUTED TRAINING WORKLOADS | 27 |
| 4.1. USING THE KUBEFLOW TRAINING OPERATOR TO RUN DISTRIBUTED TRAINING WORKLOADS | 27 |
| 4.1.1. Creating a Training Operator PyTorch training script ConfigMap resource | 27 |
| 4.1.2. Creating a Training Operator PyTorchJob resource | 28 |
| 4.1.3. Creating a Training Operator PyTorchJob resource by using the CLI | 30 |
| 4.1.4. Example Training Operator PyTorch training scripts | 32 |
| 4.1.4.1. Example Training Operator PyTorch training script: NCCL | 32 |
| 4.1.4.2. Example Training Operator PyTorch training script: DDP | 33 |
| 4.1.4.3. Example Training Operator PyTorch training script: FSDP | 35 |
| 4.1.5. Example Dockerfile for a Training Operator PyTorch training script | 37 |
| 4.1.6. Example Training Operator PyTorchJob resource for multi-node training | 37 |
| 4.2. USING THE TRAINING OPERATOR SDK TO RUN DISTRIBUTED TRAINING WORKLOADS | 39 |
| 4.2.1. Configuring a training job by using the Training Operator SDK | 39 |
| 4.2.2. Running a training job by using the Training Operator SDK | 41 |
| 4.2.3. TrainingClient API: Job-related methods | 43 |
| 4.3. FINE-TUNING A MODEL BY USING KUBEFLOW TRAINING | 44 |
| 4.3.1. Configuring the fine-tuning job | 45 |
| 4.3.2. Running the fine-tuning job | 51 |
| 4.3.3. Deleting the fine-tuning job | 53 |
| 4.4. CREATING A MULTI-NODE PYTORCH TRAINING JOB WITH RDMA | 54 |
| 4.5. EXAMPLE TRAINING OPERATOR PYTORCHJOB RESOURCE CONFIGURED TO RUN WITH RDMA | 58 |
| CHAPTER 5. MONITORING DISTRIBUTED WORKLOADS | 60 |
| 5.1. VIEWING PROJECT METRICS FOR DISTRIBUTED WORKLOADS | 60 |
| 5.2. VIEWING THE STATUS OF DISTRIBUTED WORKLOADS | 61 |
| 5.3. VIEWING KUEUE ALERTS FOR DISTRIBUTED WORKLOADS | 62 |
| CHAPTER 6. TROUBLESHOOTING COMMON PROBLEMS WITH DISTRIBUTED WORKLOADS FOR USERS | 64 |
| 6.1. MY RAY CLUSTER IS IN A SUSPENDED STATE | 64 |
| 6.2. MY RAY CLUSTER IS IN A FAILED STATE | 65 |
| 6.3. I SEE A "FAILED TO CALL WEBHOOK" ERROR MESSAGE FOR THE CODEFLARE OPERATOR | 65 |

| | |
|--|----|
| 6.4. I SEE A "FAILED TO CALL WEBHOOK" ERROR MESSAGE FOR KUEUE | 65 |
| 6.5. MY RAY CLUSTER DOES NOT START | 66 |
| 6.6. I SEE A "DEFAULT LOCAL QUEUE NOT FOUND" ERROR MESSAGE | 67 |
| 6.7. I SEE A "LOCAL_QUEUE PROVIDED DOES NOT EXIST" ERROR MESSAGE | 67 |
| 6.8. I CANNOT CREATE A RAY CLUSTER OR SUBMIT JOBS | 68 |
| 6.9. MY POD PROVISIONED BY KUEUE IS TERMINATED BEFORE MY IMAGE IS PULLED | 68 |

PREFACE

To train complex machine-learning models or process data more quickly, you can use the distributed workloads feature to run your jobs on multiple OpenShift worker nodes in parallel. This approach significantly reduces the task completion time, and enables the use of larger datasets and more complex models.

CHAPTER 1. OVERVIEW OF DISTRIBUTED WORKLOADS

You can use the distributed workloads feature to queue, scale, and manage the resources required to run data science workloads across multiple nodes in an OpenShift cluster simultaneously. Typically, data science workloads include several types of artificial intelligence (AI) workloads, including machine learning (ML) and Python workloads.

Distributed workloads provide the following benefits:

- You can iterate faster and experiment more frequently because of the reduced processing time.
- You can use larger datasets, which can lead to more accurate models.
- You can use complex models that could not be trained on a single node.
- You can submit distributed workloads at any time, and the system then schedules the distributed workload when the required resources are available.

1.1. DISTRIBUTED WORKLOADS INFRASTRUCTURE

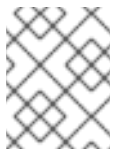
The distributed workloads infrastructure includes the following components:

CodeFlare Operator

Secures deployed Ray clusters and grants access to their URLs

CodeFlare SDK

Defines and controls the remote distributed compute jobs and infrastructure for any Python-based environment



NOTE

The CodeFlare SDK is not installed as part of OpenShift AI, but it is included in some of the workbench images provided by OpenShift AI.

Kubeflow Training Operator

Provides fine-tuning and scalable distributed training of ML models created with different ML frameworks such as PyTorch

Kubeflow Training Operator Python Software Development Kit (Training Operator SDK)

Simplifies the creation of distributed training and fine-tuning jobs



NOTE

The Training Operator SDK is not installed as part of OpenShift AI, but it is included in some of the workbench images provided by OpenShift AI.

KubeRay

Manages remote Ray clusters on OpenShift for running distributed compute workloads

Kueue

Manages quotas and how distributed workloads consume them, and manages the queueing of distributed workloads with respect to quotas

For information about installing these components, see [Installing the distributed workloads components](#).

1.2. TYPES OF DISTRIBUTED WORKLOADS

Depending on which type of distributed workloads you want to run, you must enable different OpenShift AI components:

- Ray-based distributed workloads: Enable the **codeflare**, **kueue**, and **ray** components.
- Training Operator-based distributed workloads: Enable the **trainingoperator** and **kueue** components.

For both Ray-based and Training Operator-based distributed workloads, you can use Kueue and supported accelerators:

- Use Kueue to manage the resources for the distributed workload.
- Use CUDA training images for NVIDIA GPUs, and ROCm-based training images for AMD GPUs.

For more information about supported accelerators, see the [Red Hat OpenShift AI: Supported Configurations](#) Knowledgebase article

You can run distributed workloads from data science pipelines, from Jupyter notebooks, or from Microsoft Visual Studio Code files.



NOTE

Data science pipelines workloads are not managed by the distributed workloads feature, and are not included in the distributed workloads metrics.

CHAPTER 2. PREPARING THE DISTRIBUTED TRAINING ENVIRONMENT

Before you run a distributed training or tuning job, prepare your training environment as follows:

- Create a workbench with the appropriate workbench image. Review the list of packages in each workbench image to find the most suitable image for your distributed training workload.
- Ensure that you have the credentials to authenticate to the OpenShift cluster.
- Select a suitable training image. Choose from the list of base training images provided with Red Hat OpenShift AI, or create a custom training image.

For information about the workbench images and training images provided with Red Hat OpenShift AI, and their preinstalled packages, see the [Red Hat OpenShift AI: Supported Configurations](#) Knowledgebase article.

2.1. CREATING A WORKBENCH FOR DISTRIBUTED TRAINING

Create a workbench with the appropriate resources to run a distributed training or tuning job.

Prerequisites

- You can access an OpenShift cluster that has sufficient worker nodes with supported accelerators to run your training or tuning job.
- Your cluster administrator has configured the cluster as follows:
 - Installed Red Hat OpenShift AI with the required distributed training components, as described in [Installing the distributed workloads components](#).
 - Configured the distributed training resources, as described in [Managing distributed workloads](#).
 - Configured supported accelerators, as described in [Working with accelerators](#).

Procedure

1. Log in to the Red Hat OpenShift AI web console.
2. If you want to add the workbench to an existing project, open the project and proceed to the next step.
If you want to add the workbench to a new project, create the project as follows:
 - a. In the left navigation pane, click **Data science projects**, and click **Create project**.
 - b. Enter a project name, and optionally a description, and click **Create**. The project details page opens, with the **Overview** tab selected by default.
3. Create a workbench as follows:
 - a. On the project details page, click the **Workbench** tab, and click **Create workbench**.
 - b. Enter a workbench name, and optionally a description.

- c. In the **Workbench image** section, from the **Image selection** list, select the appropriate image for your training or tuning job. If project-scoped images exist, the **Image selection** list includes subheadings to distinguish between global images and project-scoped images. For example, to run the example fine-tuning job described in [Fine-tuning a model by using Kubeflow Training](#), select **PyTorch**.
- d. In the **Deployment size** section, select one of the following options, depending on whether the hardware profiles feature is enabled.



IMPORTANT

The hardware profiles feature is currently available in Red Hat OpenShift AI as a Technology Preview feature. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [Technology Preview Features Support Scope](#).

- If the hardware profiles feature is not enabled:
 - i. From the **Container size** list, select the appropriate size for the size of the model that you want to train or tune.
For example, to run the example fine-tuning job described in [Fine-tuning a model by using Kubeflow Training](#), select **Medium**.
 - ii. From the **Accelerator** list, select a suitable accelerator profile for your workbench. If project-scoped accelerator profiles exist, the **Accelerator** list includes subheadings to distinguish between global accelerator profiles and project-scoped accelerator profiles.
- If the hardware profiles feature is enabled:
 - i. From the **Hardware profile** list, select a suitable hardware profile for your workbench.
If project-scoped hardware profiles exist, the **Hardware profile** list includes subheadings to distinguish between global hardware profiles and project-scoped hardware profiles.

The hardware profile specifies the number of CPUs and the amount of memory allocated to the container, setting the guaranteed minimum (request) and maximum (limit) for both.
 - ii. If you want to change the default values, click **Customize resource requests and limit** and enter new minimum (request) and maximum (limit) values.



IMPORTANT

By default, the hardware profiles feature is not enabled: hardware profiles are not shown in the dashboard navigation menu or elsewhere in the user interface. In addition, user interface components associated with the deprecated accelerator profiles functionality are still displayed. To show the **Settings → Hardware profiles** option in the dashboard navigation menu, and the user interface components associated with hardware profiles, set the **disableHardwareProfiles** value to **false** in the **OdhDashboardConfig** custom resource (CR) in OpenShift. For more information about setting dashboard configuration options, see [Customizing the dashboard](#).

- e. In the **Cluster storage** section, click either **Attach existing storage** or **Create storage** to specify the storage details so that you can share data between the workbench and the training or tuning runs.
For example, to run the example fine-tuning job described in [Fine-tuning a model by using KubeFlow Training](#), specify a storage class with ReadWriteMany (RWX) capability.
- f. Review the storage configuration and click **Create workbench**.

Verification

On the **Workbenches** tab, the status changes from **Starting** to **Running**.

Additional resources

- [Creating a data science project](#)
- [Creating a workbench and selecting an IDE](#)
- [Working in your data science IDE](#)
- [Red Hat OpenShift AI: Supported Configurations](#) Knowledgebase article

2.2. USING THE CLUSTER SERVER AND TOKEN TO AUTHENTICATE


To interact with the OpenShift cluster, you must authenticate to the OpenShift API by specifying the cluster server and token. You can find these values from the OpenShift Console.

Prerequisites

- You can access the OpenShift Console.

Procedure

1. Log in to the OpenShift Console.

In the OpenShift AI top navigation bar, click the application launcher icon () and then click **OpenShift Console**.

2. In the upper-right corner of the OpenShift Console, click your user name and click **Copy login command**.

3. In the new tab that opens, log in as the user whose credentials you want to use.
4. Click **Display Token**.
5. In the **Log in with this token** section, find the required values as follows:
 - The **token** value is the text after the **--token=** prefix.
 - The **server** value is the text after the **--server=** prefix.



NOTE

The **token** and **server** values are security credentials, treat them with care.

- Do not save the token and server details in a notebook file.
- Do not store the token and server details in Git.

The token expires after 24 hours.

6. You can use the token and server details to authenticate in various ways, as shown in the following examples:
 - You can specify the values in a notebook cell:

```
api_server = "<server>"
token = "<token>"
```

- You can log in to the OpenShift command-line interface (CLI) by copying the entire **Log in with this token** command and pasting the command in a terminal window.

```
$ oc login --token=<token> --server=<server>
```

2.3. MANAGING CUSTOM TRAINING IMAGES

To run distributed training jobs, you can use one of the base training images that are provided with OpenShift AI, or you can create your own custom training images. You can optionally push your custom training images to the integrated OpenShift image registry, to make your images available to other users.

2.3.1. About base training images

The base training images for distributed workloads are optimized with the tools and libraries that you need to run distributed training jobs. You can use the provided base images, or you can create custom images that are specific to your needs.

For information about Red Hat support of training images and packages, see [Red Hat OpenShift AI: Supported Configurations](#).

The following table lists the training images that are installed with Red Hat OpenShift AI by default. These images are AMD64 images, which might not work on other architectures.

Table 2.1. Default training base images

| Image type | Description |
|------------|--|
| Ray CUDA | If you are working with compute-intensive models and you want to accelerate the training job with NVIDIA GPU support, you can use the Ray Compute Unified Device Architecture (CUDA) base image to gain access to the NVIDIA CUDA Toolkit. Using this toolkit, you can accelerate your work by using libraries and tools that are optimized for NVIDIA GPUs. |
| Ray ROCm | If you are working with compute-intensive models and you want to accelerate the training job with AMD GPU support, you can use the Ray ROCm base image to gain access to the AMD ROCm software stack. Using this software stack, you can accelerate your work by using libraries and tools that are optimized for AMD GPUs. |
| KFTO CUDA | If you are working with compute-intensive models and you want to accelerate the training job with NVIDIA GPU support, you can use the Kubeflow Training Operator CUDA base image to gain access to the NVIDIA CUDA Toolkit. Using this toolkit, you can accelerate your work by using libraries and tools that are optimized for NVIDIA GPUs. |
| KFTO ROCm | If you are working with compute-intensive models and you want to accelerate the training job with AMD GPU support, you can use the Kubeflow Training Operator ROCm base image to gain access to the AMD ROCm software stack. Using this software stack, you can accelerate your work by using libraries and tools that are optimized for AMD GPUs. |

If the preinstalled packages that are provided in these images are not sufficient for your use case, you have the following options:

- Install additional libraries after launching a default image. This option is good if you want to add libraries on an ad hoc basis as you run training jobs. However, it can be challenging to manage the dependencies of installed libraries.
- Create a custom image that includes the additional libraries or packages. For more information, see [Creating a custom training image](#).

2.3.2. Creating a custom training image

You can create a custom training image by adding packages to a base training image.

Prerequisites

- You can access the training image that you have chosen to use as the base for your custom image.
Select the image based on the *image type* (for example, Ray or Kubeflow Training Operator), the *accelerator framework* (for example, CUDA for NVIDIA GPUs, or ROCm for AMD GPUs), and the *Python version* (for example, 3.9 or 3.11).

The following table shows some example base training images:

Table 2.2. Example base training images

| Image type | Accelerator framework | Python version | Example base training image | Preinstalled packages |
|------------|-----------------------|----------------|--|---------------------------------------|
| Ray | CUDA | 3.9 | ray:2.35.0-py39-cu121 | Ray 2.35.0, Python 3.9, CUDA 12.1 |
| Ray | CUDA | 3.11 | ray:2.47.1-py311-cu121 | Ray 2.47.1, Python 3.11, CUDA 12.1 |
| Ray | ROCm | 3.9 | ray:2.35.0-py39-rocm62 | Ray 2.35.0, Python 3.9, ROCm 6.2 |
| Ray | ROCm | 3.11 | ray:2.47.1-py311-rocm62 | Ray 2.47.1, Python 3.11, ROCm 6.2 |
| KF TO | CUDA | 3.11 | training:py311-cuda124-torch251 | Python 3.11, CUDA 12.4, PyTorch 2.5.1 |
| KF TO | ROCm | 3.11 | training:py311-rocm62-torch251 | Python 3.11, ROCm 6.2, PyTorch 2.5.1 |

For a complete list of the OpenShift AI base training images and their preinstalled packages, see [Supported Configurations](#).

- You have Podman installed in your local environment, and you can access a container registry. For more information about Podman and container registries, see [Building, running, and managing containers](#).

Procedure

1. In a terminal window, create a directory for your work, and change to that directory.
2. Set the **IMG** environment variable to the name of your custom image. In the example commands in this section, **my_training_image** is the name of the custom image.

```
export IMG=my_training_image
```

3. Create a file named **Dockerfile** with the following content:
 - a. Use the **FROM** instruction to specify the location of a suitable base training image. In the following command, replace **<base-training-image>** with the name of your chosen base training image:

```
FROM quay.io/modh/<base-training-image>
```

Examples:

```
FROM quay.io/modh/ray:2.47.1-py311-cu121
```



```
FROM quay.io/modh/training:py311-rocm62-torch251
```

- b. Use the **RUN** instruction to install additional packages. You can also add comments to the Dockerfile by prefixing each comment line with a number sign (**#**).

The following example shows how to install a specific version of the Python PyTorch package:

```
# Install PyTorch
RUN python3 -m pip install torch==2.5.1
```

4. Build the image file. Use the **-t** option with the **podman build** command to create an image tag that specifies the custom image name and version, to make it easier to reference and manage the image:

```
podman build -t <custom-image-name>:_<version>_ -f Dockerfile
```

Example:

```
podman build -t ${IMG}:0.0.1 -f Dockerfile
```

The build output indicates when the build process is complete.

5. Display a list of your images:

```
podman images
```

If your new image was created successfully, it is included in the list of images.

6. Push the image to your container registry:

```
podman push ${IMG}:0.0.1
```

7. Optional: Make your new image available to other users, as described in [Pushing an image to the integrated OpenShift image registry](#).

2.3.3. Pushing an image to the integrated OpenShift image registry

To make an image available to other users in your OpenShift cluster, you can push the image to the *integrated OpenShift image registry*, a built-in container image registry.

For more information about the integrated OpenShift image registry, see [Integrated OpenShift image registry](#).

Prerequisites

- Your cluster administrator has exposed the integrated image registry, as described in [Exposing the registry](#).
- You have Podman installed in your local environment.
For more information about Podman and container registries, see [Building, running, and managing containers](#).

Procedure

1. In a terminal window, log in to the OpenShift CLI as shown in the following example:

```
$ oc login <openshift_cluster_url> -u <admin_username> -p <password>
```

2. Set the **IMG** environment variable to the name of your image. In the example commands in this section, **my_training_image** is the name of the image.

```
export IMG=my_training_image
```

3. Log in to the integrated image registry:

```
podman login -u $(oc whoami) -p $(oc whoami -t) $(oc registry info)
```

4. Tag the image for the integrated image registry:

```
podman tag ${IMG} $(oc registry info)/$(oc project -q)/${IMG}
```

5. Push the image to the integrated image registry:

```
podman push $(oc registry info)/$(oc project -q)/${IMG}
```

6. Retrieve the image repository location for the tag that you want:

```
oc get is ${IMG} -o jsonpath='{.status.tags[?(@.tag=="<TAG>")].items[0].dockerImageReference}'
```

Any user can now use your image by specifying this retrieved image location value in the **image** parameter of a Ray cluster or training job.

CHAPTER 3. RUNNING RAY-BASED DISTRIBUTED WORKLOADS

In OpenShift AI, you can run a Ray-based distributed workload from a Jupyter notebook or from a pipeline.

You can run Ray-based distributed workloads in a disconnected environment if you can access all of the required software from that environment. For example, you must be able to access a Ray cluster image, and the data sets and Python dependencies used by the workload, from the disconnected environment.

3.1. RUNNING DISTRIBUTED DATA SCIENCE WORKLOADS FROM JUPYTER NOTEBOOKS

To run a distributed workload from a Jupyter notebook, you must configure a Ray cluster. You must also provide environment-specific information such as cluster authentication details.

The examples in this section refer to the JupyterLab integrated development environment (IDE).

3.1.1. Downloading the demo Jupyter notebooks from the CodeFlare SDK

The demo Jupyter notebooks from the CodeFlare SDK provide guidelines on how to use the CodeFlare stack in your Jupyter notebooks. Download the demo Jupyter notebooks so that you can learn how to run Jupyter notebooks locally.

Prerequisites

- You can access a data science cluster that is configured to run distributed workloads as described in [Managing distributed workloads](#).
- You can access a data science project that contains a workbench, and the workbench is running a default workbench image that contains the CodeFlare SDK, for example, the **Standard Data Science** notebook. For information about projects and workbenches, see [Working on data science projects](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have logged in to Red Hat OpenShift AI, started your workbench, and logged in to JupyterLab.

Procedure

1. In the JupyterLab interface, click **File > New > Notebook**. Specify your preferred Python version, and then click **Select**.
A new Jupyter notebook file is created with the **.ipynb** file name extension.
2. Add the following code to a cell in the new notebook:

Code to download the demo Jupyter notebooks

```
from codeflare_sdk import copy_demo_nbs
copy_demo_nbs()
```

3. Select the cell, and click **Run > Run selected cell**
After a few seconds, the **copy_demo_nbs()** function copies the demo Jupyter notebooks that are packaged with the currently installed version of the CodeFlare SDK, and clones them into the **demo-notebooks** folder.
4. In the left navigation pane, right-click the new notebook and click **Delete**.
5. Click **Delete** to confirm.

Verification

Locate the downloaded demo Jupyter notebooks in the JupyterLab interface, as follows:

1. In the left navigation pane, double-click **demo-notebooks**.
2. Double-click **additional-demos** and verify that the folder contains several demo Jupyter notebooks.
3. Click **demo-notebooks**.
4. Double-click **guided-demos** and verify that the folder contains several demo Jupyter notebooks.

You can run these demo Jupyter notebooks as described in [Running the demo Jupyter notebooks from the CodeFlare SDK](#).

3.1.2. Running the demo Jupyter notebooks from the CodeFlare SDK

To run the demo Jupyter notebooks from the CodeFlare SDK, you must provide environment-specific information.

In the examples in this procedure, you edit the demo Jupyter notebooks in JupyterLab to provide the required information, and then run the Jupyter notebooks.

Prerequisites

- You can access a data science cluster that is configured to run distributed workloads as described in [Managing distributed workloads](#).
- You can access the following software from your data science cluster:
 - A Ray cluster image that is compatible with your hardware architecture
 - The data sets and models to be used by the workload
 - The Python dependencies for the workload, either in a Ray image or in your own Python Package Index (PyPI) server
- You can access a data science project that contains a workbench, and the workbench is running a default workbench image that contains the CodeFlare SDK, for example, the **Standard Data Science** workbench. For information about projects and workbenches, see [Working on data science projects](#).

- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have logged in to Red Hat OpenShift AI, started your workbench, and logged in to JupyterLab.
- You have downloaded the demo Jupyter notebooks provided by the CodeFlare SDK, as described in [Downloading the demo Jupyter notebooks from the CodeFlare SDK](#).

Procedure

1. Check whether your cluster administrator has defined a *default* local queue for the Ray cluster. You can use the **codeflare_sdk.list_local_queues()** function to view all local queues in your current namespace, and the resource flavors associated with each local queue.

Alternatively, you can use the OpenShift web console as follows:

- a. In the OpenShift web console, select your project from the **Project** list.
- b. Click **Search**, and from the **Resources** list, select **LocalQueue** to show the list of local queues for your project.
If no local queue is listed, contact your cluster administrator.
- c. Review the details of each local queue:
 - i. Click the local queue name.
 - ii. Click the **YAML** tab, and review the **metadata.annotations** section.
If the **kueue.x-k8s.io/default-queue** annotation is set to **'true'**, the queue is configured as the default local queue.



NOTE

If your cluster administrator does not define a default local queue, you must specify a local queue in each Jupyter notebook.

2. In the JupyterLab interface, open the **demo-notebooks > guided-demos** folder.
3. Open all of the Jupyter notebooks by double-clicking each Jupyter notebook file.
Jupyter notebook files have the **.ipynb** file name extension.
4. In each Jupyter notebook, ensure that the **import** section imports the required components from the CodeFlare SDK, as follows:

Example import section

```
from codeflare_sdk import Cluster, ClusterConfiguration, TokenAuthentication
```

5. In each Jupyter notebook, update the **TokenAuthentication** section to provide the **token** and **server** details to authenticate to the OpenShift cluster by using the CodeFlare SDK.

For information about how to find the server and token details, see [Using the cluster server and token to authenticate](#).

6. Optional: If you want to use custom certificates, update the **TokenAuthentication** section to add the **ca_cert_path** parameter to specify the location of the custom certificates, as shown in the following example:

Example authentication section

```
auth = TokenAuthentication(
    token = "XXXXXX",
    server = "XXXXXX",
    skip_tls=False,
    ca_cert_path="/path/to/cert"
)
auth.login()
```

Alternatively, you can set the **CF_SDK_CA_CERT_PATH** environment variable to specify the location of the custom certificates.

7. In each Jupyter notebook, update the cluster configuration section as follows:
 - a. If the **namespace** value is specified, replace the example value with the name of your project.
If you omit this line, the Ray cluster is created in the current project.
 - b. If the **image** value is specified, replace the example value with a link to a suitable Ray cluster image. The Python version in the Ray cluster image must be the same as the Python version in the workbench.
If you omit this line, one of the following Ray cluster images is used by default, based on the Python version detected in the workbench:

- Python 3.9: **quay.io/modh/ray:2.35.0-py39-cu121**
- Python 3.11: **quay.io/modh/ray:2.47.1-py311-cu121**

The default Ray images are compatible with NVIDIA GPUs that are supported by the specified CUDA version. The default images are AMD64 images, which might not work on other architectures.

Additional ROCm-compatible Ray cluster images are available, which are compatible with AMD accelerators that are supported by the specified ROCm version. These images are AMD64 images, which might not work on other architectures.

For information about the latest available training images and their preinstalled packages, including the CUDA and ROCm versions, see [Red Hat OpenShift AI: Supported Configurations](#).

- c. If your cluster administrator has not configured a default local queue, specify the local queue for the Ray cluster, as shown in the following example:

Example local queue assignment

```
local_queue="your_local_queue_name"
```

- d. Optional: Assign a dictionary of **labels** parameters to the Ray cluster for identification and management purposes, as shown in the following example:

Example labels assignment

```
labels = {"exampleLabel1": "exampleLabel1Value", "exampleLabel2":
"exampleLabel2Value"}
```

8. In the **2_basic_interactive.ipynb** Jupyter notebook, ensure that the following Ray cluster authentication code is included after the Ray cluster creation section:

Ray cluster authentication code

```
from codeflare_sdk import generate_cert
generate_cert.generate_tls_cert(cluster.config.name, cluster.config.namespace)
generate_cert.export_env(cluster.config.name, cluster.config.namespace)
```



NOTE

Mutual Transport Layer Security (mTLS) is enabled by default in the CodeFlare component in OpenShift AI. You must include the Ray cluster authentication code to enable the Ray client that runs within a Jupyter notebook to connect to a secure Ray cluster that has mTLS enabled.

9. Run the Jupyter notebooks in the order indicated by the file-name prefix (**0_**, **1_**, and so on).
 - a. In each Jupyter notebook, run each cell in turn, and review the cell output.
 - b. If an error is shown, review the output to find information about the problem and the required corrective action. For example, replace any deprecated parameters as instructed. See also [Troubleshooting common problems with distributed workloads for users](#).
 - c. For more information about the interactive browser controls that you can use to simplify Ray cluster tasks when working within a Jupyter notebook, see [Managing Ray clusters from within a Jupyter notebook](#).

Verification

1. The Jupyter notebooks run to completion without errors.
2. In the Jupyter notebooks, the output from the **cluster.status()** function or **cluster.details()** function indicates that the Ray cluster is **Active**.

3.1.3. Managing Ray clusters from within a Jupyter notebook

You can use interactive browser controls to simplify Ray cluster tasks when working within a Jupyter notebook.

The interactive browser controls provide an alternative to the equivalent commands, but do not replace them. You can continue to manage the Ray clusters by running commands within the Jupyter notebook, for ease of use in scripts and pipelines.

Several different interactive browser controls are available:

- When you run a cell that provides the cluster configuration, the Jupyter notebook automatically shows the controls for starting or deleting the cluster.
- You can run the **view_clusters()** command to add controls that provide the following functionality:
 - View a list of the Ray clusters that you can access.
 - View cluster information, such as cluster status and allocated resources, for the selected Ray cluster. You can view this information from within the Jupyter notebook, without switching to the OpenShift console or the Ray dashboard.
 - Open the Ray dashboard directly from the Jupyter notebook, to view the submitted jobs.
 - Refresh the Ray cluster list and the cluster information for the selected cluster.

You can add these controls to existing Jupyter notebooks, or manage the Ray clusters from a separate Jupyter notebook.

The **3_widget_example.ipynb** demo Jupyter notebook shows all of the available interactive browser controls. In the example in this procedure, you create a new Jupyter notebook to manage the Ray clusters, similar to the example provided in the **3_widget_example.ipynb** demo Jupyter notebook.

Prerequisites

- You can access a data science cluster that is configured to run distributed workloads as described in [Managing distributed workloads](#).
- You can access the following software from your data science cluster:
 - A Ray cluster image that is compatible with your hardware architecture
 - The data sets and models to be used by the workload
 - The Python dependencies for the workload, either in a Ray image or in your own Python Package Index (PyPI) server
- You can access a data science project that contains a workbench, and the workbench is running a default workbench image that contains the CodeFlare SDK, for example, the **Standard Data Science** workbench. For information about projects and workbenches, see [Working on data science projects](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have logged in to Red Hat OpenShift AI, started your workbench, and logged in to JupyterLab.
- You have downloaded the demo Jupyter notebooks provided by the CodeFlare SDK, as described in [Downloading the demo Jupyter notebooks from the CodeFlare SDK](#).

Procedure

1. Run all of the demo Jupyter notebooks in the order indicated by the file-name prefix (**0_1_** and so on), as described in [Running the demo Jupyter notebooks from the CodeFlare SDK](#).
2. In each demo Jupyter notebook, when you run the cluster configuration step, the following interactive controls are automatically shown in the Jupyter notebook:
 - **Cluster Up:** You can click this button to start the Ray cluster. This button is equivalent to the **cluster.up()** command. When you click this button, a message indicates whether the cluster was successfully created.
 - **Cluster Down:** You can click this button to delete the Ray cluster. This button is equivalent to the **cluster.down()** command. The cluster is deleted immediately; you are not prompted to confirm the deletion. When you click this button, a message indicates whether the cluster was successfully deleted.
 - **Wait for Cluster:** You can select this option to specify that the notebook cell should wait for the Ray cluster dashboard to be ready before proceeding to the next step. This option is equivalent to the **cluster.wait_ready()** command.
3. In the JupyterLab interface, create a new Jupyter notebook to manage the Ray clusters, as follows:
 - a. Click **File > New > Notebook** Specify your preferred Python version, and then click **Select**. A new Jupyter notebook file is created with the **.ipynb** file name extension.
 - b. Add the following code to a cell in the new Jupyter notebook:

Code to import the required packages

```
from codeflare_sdk import TokenAuthentication, view_clusters
```

The **view_clusters** package provides the interactive browser controls for listing the clusters, showing the cluster details, opening the Ray dashboard, and refreshing the cluster data.

- c. Add a new notebook cell, and add the following code to the new cell:

Code to authenticate

```
auth = TokenAuthentication(
    token = "XXXXX",
    server = "XXXXX",
    skip_tls=False
)
auth.login()
```

For information about how to find the token and server values, see [Running the demo Jupyter notebooks from the CodeFlare SDK](#).

- d. Add a new notebook cell, and add the following code to the new cell:

Code to view clusters in the current project

```
view_clusters()
```

When you run the **view_clusters()** command with no arguments specified, you generate a list of all of the Ray clusters in the *current* project, and display information similar to the **cluster.details()** function.

If you have access to another project, you can list the Ray clusters in that project by specifying the project name as shown in the following example:

Code to view clusters in another project

```
view_clusters("my_second_project")
```

- e. Click **File > Save Notebook As**, enter **demo-notebooks/guided-demos/manage_ray_clusters.ipynb**, and click **Save**.
4. In the **demo-notebooks/guided-demos/manage_ray_clusters.ipynb** Jupyter notebook, select each cell in turn, and click **Run > Run selected cell**
5. When you run the cell with the **view_clusters()** function, the output depends on whether any Ray clusters exist.
If no Ray clusters exist, the following text is shown, where **_[project-name]** is the name of the target project:

```
No clusters found in the [project-name] namespace.
```

Otherwise, the Jupyter notebook shows the following information about the existing Ray clusters:

- **Select an existing cluster**
Under this heading, a toggle button is shown for each existing cluster. Click a cluster name to select the cluster. The cluster details section is updated to show details about the selected cluster; for example, cluster name, OpenShift AI project name, cluster resource information, and cluster status.
- **Delete cluster**
Click this button to delete the selected cluster. This button is equivalent to the **Cluster Down** button. The cluster is deleted immediately; you are not prompted to confirm the deletion. A message indicates whether the cluster was successfully deleted, and the corresponding button is no longer shown under the **Select an existing cluster** heading.
- **View Jobs**
Click this button to open the **Jobs** tab in the Ray dashboard for the selected cluster, and view details of the submitted jobs. The corresponding URL is shown in the Jupyter notebook.
- **Open Ray Dashboard**
Click this button to open the **Overview** tab in the Ray dashboard for the selected cluster. The corresponding URL is shown in the Jupyter notebook.
- **Refresh Data**
Click this button to refresh the list of Ray clusters, and the cluster details for the selected cluster, on demand. The cluster details are automatically refreshed when you select a cluster and when you delete the selected cluster.

Verification

1. The demo Jupyter notebooks run to completion without errors.
2. In the **manage_ray_clusters.ipynb** Jupyter notebook, the output from the **view_clusters()** function is correct.

3.2. RUNNING DISTRIBUTED DATA SCIENCE WORKLOADS FROM DATA SCIENCE PIPELINES

To run a distributed workload from a pipeline, you must first update the pipeline to include a link to your Ray cluster image.

Prerequisites

- You can access a data science cluster that is configured to run distributed workloads as described in [Managing distributed workloads](#).
- You can access the following software from your data science cluster:
 - A Ray cluster image that is compatible with your hardware architecture
 - The data sets and models to be used by the workload
 - The Python dependencies for the workload, either in a Ray image or in your own Python Package Index (PyPI) server
- You can access a data science project that contains a workbench, and the workbench is running a default workbench image that contains the CodeFlare SDK, for example, the **Standard Data Science** workbench. For information about projects and workbenches, see [Working on data science projects](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have access to S3-compatible object storage.
- You have logged in to Red Hat OpenShift AI.

Procedure

1. Create a connection to connect the object storage to your data science project, as described in [Adding a connection to your data science project](#).
2. Configure a pipeline server to use the connection, as described in [Configuring a pipeline server](#).
3. Create the data science pipeline as follows:
 - a. Install the **kfp** Python package, which is required for all pipelines:


```
$ pip install kfp
```
 - b. Install any other dependencies that are required for your pipeline.

- c. Build your data science pipeline in Python code.
For example, create a file named **compile_example.py** with the following content.

```

from kfp import dsl

@dsl.component(
    base_image="registry.redhat.io/ubi9/python-311:latest",
    packages_to_install=['codeflare-sdk']
)

def ray_fn():
    import ray 1
    from codeflare_sdk import Cluster, ClusterConfiguration, generate_cert 2

    cluster = Cluster( 3
        ClusterConfiguration(
            namespace="my_project", 4
            name="raytest",
            num_workers=1,
            head_cpu_requests="500m",
            head_cpu_limits="500m",
            worker_memory_requests=1,
            worker_memory_limits=1,
            worker_extended_resource_requests={"nvidia.com/gpu": 1}, 5
            image="quay.io/modh/ray:2.47.1-py311-cu121", 6
            local_queue="local_queue_name", 7
        )
    )

    print(cluster.status())
    cluster.up() 8
    cluster.wait_ready() 9
    print(cluster.status())
    print(cluster.details())

    ray_dashboard_uri = cluster.cluster_dashboard_uri()
    ray_cluster_uri = cluster.cluster_uri()
    print(ray_dashboard_uri, ray_cluster_uri)

    # Enable Ray client to connect to secure Ray cluster that has mTLS enabled
    generate_cert.generate_tls_cert(cluster.config.name, cluster.config.namespace) 10
    generate_cert.export_env(cluster.config.name, cluster.config.namespace)

    ray.init(address=ray_cluster_uri)
    print("Ray cluster is up and running: ", ray.is_initialized())

@ray.remote
def train_fn(): 11

```

```

    # complex training function
    return 100

result = ray.get(train_fn.remote())
assert 100 == result
ray.shutdown()
cluster.down() 12
auth.logout()
return result

@dsl.pipeline( 13
    name="Ray Simple Example",
    description="Ray Simple Example",
)

def ray_integration():
    ray_fn()

if __name__ == '__main__': 14
    from kfp.compiler import Compiler
    Compiler().compile(ray_integration, 'compiled-example.yaml')

```

- 1 Imports Ray.
- 2 Imports packages from the CodeFlare SDK to define the cluster functions.
- 3 Specifies the Ray cluster configuration: replace these example values with the values for your Ray cluster.
- 4 Optional: Specifies the project where the Ray cluster is created. Replace the example value with the name of your project. If you omit this line, the Ray cluster is created in the current project.
- 5 Optional: Specifies the requested accelerators for the Ray cluster (in this example, 1 NVIDIA GPU). If you do not use NVIDIA GPUs, replace **nvidia.com/gpu** with the correct value for your accelerator; for example, specify **amd.com/gpu** for AMD GPUs. If no accelerators are required, set the value to 0 or omit the line.
- 6 Specifies the location of the Ray cluster image. The Python version in the Ray cluster image must be the same as the Python version in the workbench. If you omit this line, one of the default CUDA-compatible Ray cluster images is used, based on the Python version detected in the workbench. The default Ray images are AMD64 images, which might not work on other architectures. If you are running this code in a disconnected environment, replace the default value with the location for your environment. For information about the latest available training images and their preinstalled packages, see [Red Hat OpenShift AI: Supported Configurations](#).
- 7 Specifies the local queue to which the Ray cluster will be submitted. If a default local queue is configured, you can omit this line.
- 8 Creates a Ray cluster by using the specified image and configuration.

- 9 Waits until the Ray cluster is ready before proceeding.
- 10 Enables the Ray client to connect to a secure Ray cluster that has mutual Transport Layer Security (mTLS) enabled. mTLS is enabled by default in the CodeFlare component in OpenShift AI.
- 11 Replace the example details in this section with the details for your workload.
- 12 Removes the Ray cluster when your workload is finished.
- 13 Replace the example name and description with the values for your workload.
- 14 Compiles the Python code and saves the output in a YAML file.

d. Compile the Python file (in this example, the **compile_example.py** file):

```
$ python compile_example.py
```

This command creates a YAML file (in this example, **compiled-example.yaml**), which you can import in the next step.

4. Import your data science pipeline, as described in [Importing a data science pipeline](#).
5. Schedule the pipeline run, as described in [Scheduling a pipeline run](#).
6. When the pipeline run is complete, confirm that it is included in the list of triggered pipeline runs, as described in [Viewing the details of a pipeline run](#).

Verification

The YAML file is created and the pipeline run completes without errors.

You can view the run details, as described in [Viewing the details of a pipeline run](#).

Additional resources

- [Working with data science pipelines](#)
- [Ray Clusters documentation](#)

CHAPTER 4. RUNNING TRAINING OPERATOR-BASED DISTRIBUTED TRAINING WORKLOADS

To reduce the time needed to train a Large Language Model (LLM), you can run the training job in parallel. In Red Hat OpenShift AI, the Kubeflow Training Operator and Kubeflow Training Operator Python Software Development Kit (Training Operator SDK) simplify the job configuration.

You can use the Training Operator and the Training Operator SDK to configure a training job in a variety of ways. For example, you can use multiple nodes and multiple GPUs per node, fine-tune a model, or configure a training job to use Remote Direct Memory Access (RDMA).

4.1. USING THE KUBEFLOW TRAINING OPERATOR TO RUN DISTRIBUTED TRAINING WORKLOADS

You can use the Training Operator **PyTorchJob** API to configure a **PyTorchJob** resource so that the training job runs on multiple nodes with multiple GPUs.

You can store the training script in a **ConfigMap** resource, or include it in a custom container image.

4.1.1. Creating a Training Operator PyTorch training script ConfigMap resource

You can create a **ConfigMap** resource to store the Training Operator PyTorch training script.



NOTE

Alternatively, you can use the [example Dockerfile](#) to include the training script in a custom container image, as described in [Creating a custom training image](#).

Prerequisites

- Your cluster administrator has installed Red Hat OpenShift AI with the required distributed training components as described in [Installing the distributed workloads components](#).
- You can access the OpenShift Console for the cluster where OpenShift AI is installed.

Procedure

1. Log in to the OpenShift Console.
2. Create a **ConfigMap** resource, as follows:
 - a. In the **Administrator** perspective, click **Workloads** → **ConfigMaps**.
 - b. From the **Project** list, select your project.
 - c. Click **Create ConfigMap**.
 - d. In the **Configure via** section, select the **YAML view** option.
The **Create ConfigMap** page opens, with default YAML code automatically added.
3. Replace the default YAML code with your training-script code.
For example training scripts, see [Example Training Operator PyTorch training scripts](#).
4. Click **Create**.

Verification

1. In the OpenShift Console, in the **Administrator** perspective, click **Workloads → ConfigMaps**.
2. From the **Project** list, select your project.
3. Click your ConfigMap resource to display the training script details.

4.1.2. Creating a Training Operator PyTorchJob resource

You can create a **PyTorchJob** resource to run the Training Operator PyTorch training script.

Prerequisites

- You can access an OpenShift cluster that has multiple worker nodes with supported NVIDIA GPUs or AMD GPUs.
- Your cluster administrator has configured the cluster as follows:
 - Installed Red Hat OpenShift AI with the required distributed training components, as described in [Installing the distributed workloads components](#).
 - Configured the distributed training resources, as described in [Managing distributed workloads](#).
- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.

Procedure

1. Log in to the OpenShift Console.
2. Create a **PyTorchJob** resource, as follows:
 - a. In the **Administrator** perspective, click **Home → Search**.
 - b. From the **Project** list, select your project.
 - c. Click the **Resources** list, and in the search field, start typing **PyTorchJob**.
 - d. Select **PyTorchJob**, and click **Create PyTorchJob**.
The **Create PyTorchJob** page opens, with default YAML code automatically added.
3. Update the metadata to replace the **name** and **namespace** values with the values for your environment, as shown in the following example:

```
metadata:
  name: pytorch-multi-node-job
  namespace: test-namespace
```


4. Configure the master node, as shown in the following example:

```
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: OnFailure
      template:
        metadata:
          labels:
            app: pytorch-multi-node-job
```

- a. In the **replicas** entry, specify **1**. Only one master node is needed.
- b. To use a ConfigMap resource to provide the training script for the PyTorchJob pods, add the ConfigMap volume mount information, as shown in the following example:

Adding the training script from a ConfigMap resource

```
Spec:
  pytorchReplicaSpecs:
    Master:
      ...
      template:
        spec:
          containers:
            - name: pytorch
              image: quay.io/modh/training:py311-cuda124-torch251
              command: ["python", "/workspace/scripts/train.py"]
              volumeMounts:
                - name: training-script-volume
                  mountPath: /workspace
          volumes:
            - name: training-script-volume
              configMap:
                name: training-script-configmap
```

- c. Add the appropriate resource constraints for your environment, as shown in the following example:

Adding the resource constraints

```
SSpec:
  pytorchReplicaSpecs:
    Master:
      ...
      template:
        spec:
          containers: ...
          resources:
            requests:
              cpu: "4"
              memory: "8Gi"
              nvidia.com/gpu: 2 # To use GPUs (Optional)
          limits:
```



```
cpu: "4"
memory: "8Gi"
nvidia.com/gpu: 2
```

5. Make similar edits in the **Worker** section of the **PyTorchJob** resource.
 - a. Update the **replicas** entry to specify the number of worker nodes.

For a complete example **PyTorchJob** resource, see [Example Training Operator PyTorchJob resource for multi-node training](#).

6. Click **Create**.

Verification

1. In the OpenShift Console, open the **Administrator** perspective.
2. From the **Project** list, select your project.
3. Click **Home** → **Search** → **PyTorchJob** and verify that the job was created.
4. Click **Workloads** → **Pods** and verify that requested head pod and worker pods are running.

4.1.3. Creating a Training Operator PyTorchJob resource by using the CLI

You can use the OpenShift command-line interface (CLI) to create a **PyTorchJob** resource to run the Training Operator PyTorch training script.

Prerequisites

- You can access an OpenShift cluster that has multiple worker nodes with supported NVIDIA GPUs or AMD GPUs.
- Your cluster administrator has configured the cluster as follows:
 - Installed Red Hat OpenShift AI with the required distributed training components, as described in [Installing the distributed workloads components](#).
 - Configured the distributed training resources, as described in [Managing distributed workloads](#).
- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have downloaded and installed the OpenShift command-line interface (CLI), as described in [Installing the OpenShift CLI \(OpenShift Dedicated\)](#) or [Installing the OpenShift CLI \(Red Hat OpenShift Service on AWS\)](#).

Procedure

1. Log in to the OpenShift CLI, as follows:

Logging into the OpenShift CLI

```
oc login --token=<token> --server=<server>
```

For information about how to find the server and token details, see [Using the cluster server and token to authenticate](#).

2. Create a file named **train.py** and populate it with your training script, as follows:

Creating the training script

```
cat <<EOF > train.py
<paste your content here>
EOF
```

Replace *<paste your content here>* with your training script content.

For example training scripts, see [Example Training Operator PyTorch training scripts](#).

3. Create a **ConfigMap** resource to store the training script, as follows:

Creating the ConfigMap resource

```
oc create configmap training-script-configmap --from-file=train.py -n <your-namespace>
```

Replace *<your-namespace>* with the name of your project.

4. Create a file named **pytorchjob.yaml** to define the distributed training job setup, as follows:

Defining the distributed training job

```
cat <<EOF > pytorchjob.py
<paste your content here>
EOF
```

Replace *<paste your content here>* with your training job content.

For an example training job, see [Example Training Operator PyTorchJob resource for multi-node training](#).

5. Create the distributed training job, as follows:

Creating the distributed training job

```
oc apply -f pytorchjob.yaml
```

Verification

1. Monitor the running distributed training job, as follows:

Monitoring the distributed training job

```
oc get pytorchjobs -n <your-namespace>
```

Replace `<your-namespace>` with the name of your project.

2. Check the pod logs, as follows:

Checking the pod logs

```
oc logs <pod-name> -n <your-namespace>
```

Replace `<your-namespace>` with the name of your project.

3. When you want to delete the job, run the following command:

Deleting the job

```
oc delete pytorchjobs/pytorch-multi-node-job -n <your-namespace>
```

Replace `<your-namespace>` with the name of your project.

4.1.4. Example Training Operator PyTorch training scripts

The following examples show how to configure a PyTorch training script for NVIDIA Collective Communications Library (NCCL), Distributed Data Parallel (DDP), and Fully Sharded Data Parallel (FSDP) training jobs.



NOTE

If you have the required resources, you can run the example code without editing it.

Alternatively, you can modify the example code to specify the appropriate configuration for your training job.

4.1.4.1. Example Training Operator PyTorch training script: NCCL

This NVIDIA Collective Communications Library (NCCL) example returns the rank and tensor value for each accelerator.

```
import os
import torch
import torch.distributed as dist

def main():
    # Select backend dynamically: nccl for GPU, gloo for CPU
    backend = "nccl" if torch.cuda.is_available() else "gloo"

    # Initialize the process group
    dist.init_process_group(backend)

    # Get rank and world size
    rank = dist.get_rank()
    world_size = dist.get_world_size()
```

```

# Select device dynamically
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Running on rank {rank} out of {world_size} using {device} with backend {backend}.")

# Initialize tensor on the selected device
tensor = torch.zeros(1, device=device)

if rank == 0:
    tensor += 1
    for i in range(1, world_size):
        dist.send(tensor, dst=i)
else:
    dist.recv(tensor, src=0)

print(f"Rank {rank}: Tensor value {tensor.item()} on {device}")

if name == "main":
    main()

```

The **backend** value is automatically set to one of the following values:

- **nccl**: Uses NVIDIA Collective Communications Library (NCCL) for NVIDIA GPUs or ROCm Communication Collectives Library (RCCL) for AMD GPUs
- **gloo**: Uses Gloo for CPUs



NOTE

Specify **backend="nccl"** for both NVIDIA GPUs and AMD GPUs.

For AMD GPUs, even though the **backend** value is set to **nccl**, the ROCm environment uses RCCL for communication.

4.1.4.2. Example Training Operator PyTorch training script: DDP

This example shows how to configure a training script for a Distributed Data Parallel (DDP) training job.

```

import os
import sys
import torch
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch import nn, optim

# Enable verbose logging
os.environ["TORCH_DISTRIBUTED_DEBUG"] = "INFO"

def setup_ddp():
    """Initialize the distributed process group dynamically."""
    backend = "nccl" if torch.cuda.is_available() else "gloo"
    dist.init_process_group(backend=backend)
    local_rank = int(os.environ["LOCAL_RANK"])
    world_size = dist.get_world_size()

```

```

# Ensure correct device is set
device = torch.device(f"cuda:{local_rank}" if torch.cuda.is_available() else "cpu")
torch.cuda.set_device(local_rank) if torch.cuda.is_available() else None

print(f"[Rank {local_rank}] Initialized with backend={backend}, world_size={world_size}")
sys.stdout.flush() # Ensure logs are visible in Kubernetes
return local_rank, world_size, device

def cleanup():
    """Clean up the distributed process group."""
    dist.destroy_process_group()

class SimpleModel(nn.Module):
    """A simple model with multiple layers."""
    def init(self):
        super(SimpleModel, self).init()
        self.layer1 = nn.Linear(1024, 512)
        self.layer2 = nn.Linear(512, 256)
        self.layer3 = nn.Linear(256, 128)
        self.layer4 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = torch.relu(self.layer3(x))
        x = torch.relu(self.layer4(x))
        return self.output(x)

def log_ddp_parameters(model, rank):
    """Log model parameter count for DDP."""
    num_params = sum(p.numel() for p in model.parameters())
    print(f"[Rank {rank}] Model has {num_params} parameters (replicated across all ranks)")
    sys.stdout.flush()

def log_memory_usage(rank):
    """Log GPU memory usage if CUDA is available."""
    if torch.cuda.is_available():
        torch.cuda.synchronize()
        print(f"[Rank {rank}] GPU Memory Allocated: {torch.cuda.memory_allocated() / 1e6} MB")
        print(f"[Rank {rank}] GPU Memory Reserved: {torch.cuda.memory_reserved() / 1e6} MB")
        sys.stdout.flush()

def main():
    local_rank, world_size, device = setup_ddp()

    # Initialize model and wrap with DDP
    model = SimpleModel().to(device)
    model = DDP(model, device_ids=[local_rank] if torch.cuda.is_available() else None)

    print(f"[Rank {local_rank}] DDP Initialized")
    log_ddp_parameters(model, local_rank)
    log_memory_usage(local_rank)

    # Optimizer and criterion
    optimizer = optim.Adam(model.parameters(), lr=0.001)

```

```

criterion = nn.MSELoss()

# Dummy dataset (adjust for real-world use case)
x = torch.randn(32, 1024).to(device)
y = torch.randn(32, 1).to(device)

# Training loop
for epoch in range(5):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    outputs = model(x)
    loss = criterion(outputs, y)

    # Backward pass
    loss.backward()
    optimizer.step()

    print(f"[Rank {local_rank}] Epoch {epoch}, Loss: {loss.item()}")
    log_memory_usage(local_rank) # Track memory usage

    sys.stdout.flush() # Ensure logs appear in real-time

cleanup()

if name == "main":
    main()

```

4.1.4.3. Example Training Operator PyTorch training script: FSDP

This example shows how to configure a training script for a Fully Sharded Data Parallel (FSDP) training job.

```

import os
import sys
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP, CPUOffload
from torch.distributed.fsdp.wrap import always_wrap_policy
from torch import nn, optim

# Enable verbose logging for debugging
os.environ["TORCH_DISTRIBUTED_DEBUG"] = "INFO" # Enables detailed FSDP logs

def setup_ddp():
    """Initialize the distributed process group dynamically."""
    backend = "nccl" if torch.cuda.is_available() else "gloo"
    dist.init_process_group(backend=backend)
    local_rank = int(os.environ["LOCAL_RANK"])
    world_size = dist.get_world_size()

    # Ensure the correct device is set
    device = torch.device(f"cuda:{local_rank}" if torch.cuda.is_available() else "cpu")
    torch.cuda.set_device(local_rank) if torch.cuda.is_available() else None

```

```

print(f"[Rank {local_rank}] Initialized with backend={backend}, world_size={world_size}")
sys.stdout.flush() # Ensure logs are visible in Kubernetes
return local_rank, world_size, device

def cleanup():
    """Clean up the distributed process group."""
    dist.destroy_process_group()

class SimpleModel(nn.Module):
    """A simple model with multiple layers."""
    def init(self):
        super(SimpleModel, self).init()
        self.layer1 = nn.Linear(1024, 512)
        self.layer2 = nn.Linear(512, 256)
        self.layer3 = nn.Linear(256, 128)
        self.layer4 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = torch.relu(self.layer3(x))
        x = torch.relu(self.layer4(x))
        return self.output(x)

def log_fsdp_parameters(model, rank):
    """Log FSDP parameters and sharding strategy."""
    num_params = sum(p.numel() for p in model.parameters())
    print(f"[Rank {rank}] Model has {num_params} parameters (sharded across {dist.get_world_size()} workers)")
    sys.stdout.flush()

def log_memory_usage(rank):
    """Log GPU memory usage if CUDA is available."""
    if torch.cuda.is_available():
        torch.cuda.synchronize()
        print(f"[Rank {rank}] GPU Memory Allocated: {torch.cuda.memory_allocated() / 1e6} MB")
        print(f"[Rank {rank}] GPU Memory Reserved: {torch.cuda.memory_reserved() / 1e6} MB")
        sys.stdout.flush()

def main():
    local_rank, world_size, device = setup_ddp()

    # Initialize model and wrap with FSDP
    model = SimpleModel().to(device)
    model = FSDP(
        model,
        cpu_offload=CPUOffload(offload_params=not torch.cuda.is_available()), # Offload if no GPU
        auto_wrap_policy=always_wrap_policy, # Wrap all layers automatically
    )

    print(f"[Rank {local_rank}] FSDP Initialized")
    log_fsdp_parameters(model, local_rank)
    log_memory_usage(local_rank)

```



```

# Optimizer and criterion
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()

# Dummy dataset (adjust for real-world use case)
x = torch.randn(32, 1024).to(device)
y = torch.randn(32, 1).to(device)

# Training loop
for epoch in range(5):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    outputs = model(x)
    loss = criterion(outputs, y)

    # Backward pass
    loss.backward()
    optimizer.step()

    print(f"[Rank {local_rank}] Epoch {epoch}, Loss: {loss.item()}")
    log_memory_usage(local_rank) # Track memory usage

    sys.stdout.flush() # Ensure logs appear in real-time

cleanup()

if name == "main":
    main()

```

4.1.5. Example Dockerfile for a Training Operator PyTorch training script

You can use this example Dockerfile to include the training script in a custom training image.

```

FROM quay.io/modh/training:py311-cuda124-torch251
WORKDIR /workspace
COPY train.py /workspace/train.py
CMD ["python", "train.py"]

```

This example copies the training script to the default PyTorch image, and runs the script.

For more information about how to use this Dockerfile to include the training script in a custom container image, see [Creating a custom training image](#).

4.1.6. Example Training Operator PyTorchJob resource for multi-node training

This example shows how to create a Training Operator PyTorch training job that runs on multiple nodes with multiple GPUs.

```

apiVersion: kubeflow.org/v1
kind: PyTorchJob
metadata:
  name: pytorch-multi-node-job
  namespace: test-namespace

```

```
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: OnFailure
      template:
        metadata:
          labels:
            app: pytorch-multi-node-job
        spec:
          containers:
            - name: pytorch
              image: quay.io/modh/training:py311-cuda124-torch251
              imagePullPolicy: IfNotPresent
              command: ["torchrun", "/workspace/train.py"]
              volumeMounts:
                - name: training-script-volume
                  mountPath: /workspace
          resources:
            requests:
              cpu: "4"
              memory: "8Gi"
              nvidia.com/gpu: "2"
            limits:
              cpu: "4"
              memory: "8Gi"
              nvidia.com/gpu: "2"
          volumes:
            - name: training-script-volume
              configMap:
                name: training-script-configmap
    Worker:
      replicas: 1
      restartPolicy: OnFailure
      template:
        metadata:
          labels:
            app: pytorch-multi-node-job
        spec:
          containers:
            - name: pytorch
              image: quay.io/modh/training:py311-cuda124-torch251
              imagePullPolicy: IfNotPresent
              command: ["torchrun", "/workspace/train.py"]
              volumeMounts:
                - name: training-script-volume
                  mountPath: /workspace
          resources:
            requests:
              cpu: "4"
              memory: "8Gi"
              nvidia.com/gpu: "2"
            limits:
              cpu: "4"
              memory: "8Gi"
              nvidia.com/gpu: "2"
```

```
volumes:
- name: training-script-volume
  configMap:
    name: training-script-configmap
```

4.2. USING THE TRAINING OPERATOR SDK TO RUN DISTRIBUTED TRAINING WORKLOADS

You can use the Training Operator SDK to configure a distributed training job to run on multiple nodes with multiple accelerators per node.

You can configure the **PyTorchJob** resource so that the training job runs on multiple nodes with multiple GPUs.

4.2.1. Configuring a training job by using the Training Operator SDK

Before you can run a job to train a model, you must configure the training job. You must set the training parameters, define the training function, and configure the Training Operator SDK.



NOTE

The code in this procedure specifies how to configure an example training job. If you have the specified resources, you can run the example code without editing it.

Alternatively, you can modify the example code to specify the appropriate configuration for your training job.

Prerequisites

- You can access an OpenShift cluster that has sufficient worker nodes with supported accelerators to run your training or tuning job.
- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.

Procedure

1. Open the workbench, as follows:
 - a. Log in to the Red Hat OpenShift AI web console.
 - b. Click **Data science projects** and click your project.
 - c. Click the **Workbenches** tab.
 - d. If your workbench is not already running, start the workbench.
 - e. Click the **Open** link to open the IDE in a new window.

2. Click **File → New → Notebook**
3. Create the training function as shown in the following example:
 - a. Create a cell with the following content:

Example training function

```
def train_func():
    import os
    import torch
    import torch.distributed as dist

    # Select backend dynamically: nccl for GPU, gloo for CPU
    backend = "nccl" if torch.cuda.is_available() else "gloo"

    # Initialize the process group
    dist.init_process_group(backend)

    # Get rank and world size
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # Select device dynamically
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Log rank initialization
    print(f"Rank {rank}/{world_size} initialized with backend {backend} on device {device}.")

    # Initialize tensor on the selected device
    tensor = torch.zeros(1, device=device)

    if rank == 0:
        tensor += 1
        for i in range(1, world_size):
            dist.send(tensor, dst=i)
    else:
        dist.recv(tensor, src=0)

    print(f"Rank {rank}: Tensor value {tensor.item()} on {device}")

    # Cleanup
    dist.destroy_process_group()
```



NOTE

For this example training job, you do not need to install any additional packages or set any training parameters.

For more information about how to add additional packages and set the training parameters, see [Configuring the fine-tuning job](#).

- b. Optional: Edit the content to specify the appropriate values for your environment.

- c. Run the cell to create the training function.
4. Configure the Training Operator SDK client authentication as follows:
 - a. Create a cell with the following content:

Example Training Operator SDK client authentication

```
from kubernetes import client
from kubeflow.training import TrainingClient
from kubeflow.training.models import V1Volume, V1VolumeMount,
V1PersistentVolumeClaimVolumeSource

api_server = "<API_SERVER>"
token = "<TOKEN>"

configuration = client.Configuration()
configuration.host = api_server
configuration.api_key = {"authorization": f"Bearer {token}"}
# Un-comment if your cluster API server uses a self-signed certificate or an un-trusted
CA
#configuration.verify_ssl = False
api_client = client.ApiClient(configuration)
client = TrainingClient(client_configuration=api_client.configuration)
```

- b. Edit the **api_server** and **token** parameters to enter the values to authenticate to your OpenShift cluster.
For information on how to find the server and token details, see [Using the cluster server and token to authenticate](#).
- c. Run the cell to configure the Training Operator SDK client authentication.
5. Click **File > Save Notebook As**, enter an appropriate file name, and click **Save**.

Verification

1. All cells run successfully.

4.2.2. Running a training job by using the Training Operator SDK

When you run a training job to tune a model, you must specify the resources needed, and provide any authorization credentials required.



NOTE

The code in this procedure specifies how to run the example training job. If you have the specified resources, you can run the example code without editing it.

Alternatively, you can modify the example code to specify the appropriate details for your training job.

Prerequisites

- You can access an OpenShift cluster that has sufficient worker nodes with supported accelerators to run your training or tuning job.

- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- Optional: If you want to enforce the use of local queues, the local-queue labeling policy is enabled for your project namespace.
For more information about enforcing the use of local queues, see [Enforcing the local-queue labeling policy for all projects](#) or [Enforcing the local-queue labeling policy for some projects only](#).

If the local-queue labeling policy is enabled for your namespace, you have created resource flavor, cluster queue, and local queue Kueue objects for your data science project. For more information about creating these objects, see [Configuring quota management for distributed workloads](#).

- You have access to a model.
- You have access to data that you can use to train the model.
- You have configured the training job as described in [Configuring a training job by using the Training Operator SDK](#).

Procedure

1. Open the workbench, as follows:
 - a. Log in to the Red Hat OpenShift AI web console.
 - b. Click **Data science projects** and click your project.
 - c. Click the **Workbenches** tab. If your workbench is not already running, start the workbench.
 - d. Click the **Open** link to open the IDE in a new window.
2. Click **File → Open**, and open the Jupyter notebook that you used to configure the training job.
3. Create a cell to run the job, and add the following content:

```
from kubernetes import client

# Start PyTorchJob with 2 Workers and 2 GPU per Worker (multi-node, multi-worker job).
client.create_job(
    name="pytorch-ddp",
    train_func=train_func,
    base_image="quay.io/modh/training:py311-cuda124-torch251",
    num_workers=2,
    resources_per_worker={"nvidia.com/gpu": "2"},
    packages_to_install=["torchvision==0.19.0"],
    env_vars={"NCCL_DEBUG": "INFO", "TORCH_DISTRIBUTED_DEBUG": "DETAIL"},
```

```
labels={"key": "value"},
annotations={"key": "value"}
)
```

4. Edit the content to specify the appropriate values for your environment, as follows:
 - a. Edit the **num_workers** value to specify the number of worker nodes.
 - b. Update the **resources_per_worker** values according to the job requirements and the resources available.
 - c. The example provided is for NVIDIA GPUs. If you use AMD accelerators, make the following additional changes:
 - In the **resources_per_worker** entry, change **nvidia.com/gpu** to **amd.com/gpu**
 - Change the **base_image** value to **quay.io/modh/training:py311-rocm62-torch251**
 - Remove the **NCCL_DEBUG** entry
 - If the local-queue labeling policy is enabled, add the **kueue.x-k8s.io/queue-name: <local-queue-name>** label in the **labels** field:

```
client.create_job(
    name="pytorch-ddp"
    ...
    labels={"kueue.x-k8s.io/queue-name": "<local-queue-name>"}
    ...
)
```



NOTE

This example does not specify the **job_kind** parameter. If the **job_kind** value is not explicitly set, the **TrainingClient** API automatically sets the **job_kind** value to **PyTorchJob**.

5. Run the cell to run the job.

Verification

View the progress of the job as follows:

1. Create a cell with the following content:

```
client.get_job_logs(
    name="pytorch-ddp",
    job_kind="PyTorchJob",
    follow=True,
)
```

2. Run the cell to view the job progress.

4.2.3. TrainingClient API: Job-related methods

Use these methods to find job-related information.

List all training job resources

```
client.list_jobs(namespace="<namespace>", job_kind="PyTorchJob")
```

Get information about a specified training job

```
client.get_job(name="<PyTorchJob-name>", namespace="<namespace>", job_kind="PyTorchJob")
```

Get pod names for the training job

```
client.get_job_pod_names(name="<PyTorchJob-name>", namespace="<namespace>")
```

Get the logs from the training job

```
client.get_job_logs(name="<PyTorchJob-name>", namespace="<namespace>",  
job_kind="PyTorchJob")
```

Delete the training job

```
client.delete_job(name="<PyTorchJob-name>", namespace="<namespace>",  
job_kind="PyTorchJob")
```



NOTE

The **train** method from the **TrainingClient** API provides a higher-level API to fine-tune LLMs with PyTorchJobs. The **train** method is Developer Preview software, and depends on the **huggingface** Python package, which you must install manually in your environment before running it. For more information about the **train** method, see the [Kubeflow documentation](#).



IMPORTANT

Developer Preview features are not supported by Red Hat in any way and are not functionally complete or production-ready. Do not use Developer Preview features for production or business-critical workloads. Developer Preview features provide early access to functionality in advance of possible inclusion in a Red Hat product offering. Customers can use these features to test functionality and provide feedback during the development process. Developer Preview features might not have any documentation, are subject to change or removal at any time, and have received limited testing. Red Hat might provide ways to submit feedback on Developer Preview features without an associated SLA.

For more information about the support scope of Red Hat Developer Preview features, see [Developer Preview Support Scope](#).

4.3. FINE-TUNING A MODEL BY USING KUBEFLOW TRAINING

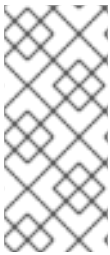
Supervised fine-tuning (SFT) is the process of customizing a Large Language Model (LLM) for a specific task by using labelled data. In this example, you use the Kubeflow Training Operator and

Kubeflow Training Operator Python Software Development Kit (Training Operator SDK) to supervise fine-tune an LLM in Red Hat OpenShift AI, by using the Hugging Face SFT Trainer.

Optionally, you can use Low-Rank Adaptation (LoRA) to efficiently fine-tune large language models. LORA optimizes computational requirements and reduces memory footprint, enabling you to fine-tune on consumer-grade GPUs. With SFT, you can combine PyTorch Fully Sharded Data Parallel (FSDP) and LoRA to enable scalable, cost-effective model training and inference, enhancing the flexibility and performance of AI workloads within OpenShift environments.

4.3.1. Configuring the fine-tuning job

Before you can use a training job to fine-tune a model, you must configure the training job. You must set the training parameters, define the training function, and configure the Training Operator SDK.



NOTE

The code in this procedure specifies how to configure an example fine-tuning job. If you have the specified resources, you can run the example code without editing it.

Alternatively, you can modify the example code to specify the appropriate configuration for your fine-tuning job.

Prerequisites

- You can access an OpenShift cluster that has sufficient worker nodes with supported accelerators to run your training or tuning job.
The example fine-tuning job requires 8 worker nodes, where each worker node has 64 GiB memory, 4 CPUs, and 1 NVIDIA GPU.
- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You can access a dynamic storage provisioner that supports ReadWriteMany (RWX) Persistent Volume Claim (PVC) provisioning, such as [Red Hat OpenShift Data Foundation](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.

Procedure

1. Open the workbench, as follows:
 - a. Log in to the Red Hat OpenShift AI web console.
 - b. Click **Data science projects** and click your project.
 - c. Click the **Workbenches** tab.
 - d. Ensure that the workbench uses a storage class with RWX capability.
 - e. If your workbench is not already running, start the workbench.

- f. Click the **Open** link to open the IDE in a new window.
2. Click **File → New → Notebook**
3. Install any additional packages that are needed to run the training or tuning job.
 - a. In a notebook cell, add the code to install the additional packages, as follows:

Code to install dependencies

```
# Install the yamlmagic package
!pip install yamlmagic
%load_ext yamlmagic

!pip install git+https://github.com/kubeflow/trainer.git@release-1.9#subdirectory=sdk/python
```

- b. Select the cell, and click **Run > Run selected cell**
The additional packages are installed.
4. Set the training parameters as follows:
 - a. Create a cell with the following content:

```
%%yaml parameters

# Model
model_name_or_path: Meta-Llama/Meta-Llama-3.1-8B-Instruct
model_revision: main
torch_dtype: bfloat16
attn_implementation: flash_attention_2

# PEFT / LoRA
use_peft: true
lora_r: 16
lora_alpha: 8
lora_dropout: 0.05
lora_target_modules: ["q_proj", "v_proj", "k_proj", "o_proj", "gate_proj", "up_proj",
"down_proj"]
lora_modules_to_save: []
init_lora_weights: true

# Quantization / BitsAndBytes
load_in_4bit: false           # use 4 bit precision for the base model (only with LoRA)
load_in_8bit: false          # use 8 bit precision for the base model (only with LoRA)

# Datasets
dataset_name: gsm8k           # id or path to the dataset
dataset_config: main          # name of the dataset configuration
dataset_train_split: train     # dataset split to use for training
dataset_test_split: test      # dataset split to use for evaluation
dataset_text_field: text       # name of the text field of the dataset
dataset_kwargs:
  add_special_tokens: false     # template with special tokens
  append_concat_token: false    # add additional separator token
```

```

# SFT
max_seq_length: 1024          # max sequence length for model and packing of the
dataset                       dataset
dataset_batch_size: 1000      # samples to tokenize per batch
packing: false
use_liger: false

# Training
num_train_epochs: 10          # number of training epochs

per_device_train_batch_size: 32 # batch size per device during training
per_device_eval_batch_size: 32  # batch size for evaluation
auto_find_batch_size: false     # find a batch size that fits into memory
                                automatically
eval_strategy: epoch           # evaluate every epoch

bf16: true                    # use bf16 16-bit (mixed) precision
tf32: false                   # use tf32 precision

learning_rate: 1.0e-4         # initial learning rate
warmup_steps: 10              # steps for a linear warmup from 0 to `learning_rate`
lr_scheduler_type: inverse_sqrt # learning rate scheduler (see
transformers.SchedulerType)

optim: adamw_torch_fused      # optimizer (see transformers.OptimizerNames)
max_grad_norm: 1.0            # max gradient norm
seed: 42

gradient_accumulation_steps: 1 # number of steps before performing a
backward/update pass
gradient_checkpointing: false   # use gradient checkpointing to save memory
gradient_checkpointing_kwargs:
  use_reentrant: false

# FSDP
fsdp: "full_shard auto_wrap offload" # remove offload if enough GPU memory
fsdp_config:
  activation_checkpointing: true
  cpu_ram_efficient_loading: false
  sync_module_states: true
  use_orig_params: true
  limit_all_gathers: false

# Checkpointing
save_strategy: epoch          # save checkpoint every epoch
save_total_limit: 1           # limit the total amount of checkpoints
resume_from_checkpoint: false  # load the last checkpoint in output_dir and
resume from it

# Logging
log_level: warning            # logging level (see transformers.logging)
logging_strategy: steps
logging_steps: 1              # log every N steps
report_to:

```

```
- tensorboard                                # report metrics to tensorboard
```

```
output_dir: /mnt/shared/Meta-Llama-3.1-8B-Instruct
```

- b. Optional: If you specify a different model or dataset, edit the parameters to suit your model, dataset, and resources. If necessary, update the previous cell to specify the dependencies for your training or tuning job.
 - c. Run the cell to set the training parameters.
5. Create the training function as follows:
- a. Create a cell with the following content:

```
def main(parameters):
    import random

    from datasets import load_dataset
    from transformers import (
        AutoTokenizer,
        set_seed,
    )

    from trl import (
        ModelConfig,
        ScriptArguments,
        SFTConfig,
        SFTTrainer,
        TrlParser,
        get_peft_config,
        get_quantization_config,
        get_kbit_device_map,
    )

    parser = TrlParser((ScriptArguments, SFTConfig, ModelConfig))
    script_args, training_args, model_args = parser.parse_dict(parameters)

    # Set seed for reproducibility
    set_seed(training_args.seed)

    # Model and tokenizer
    quantization_config = get_quantization_config(model_args)
    model_kwargs = dict(
        revision=model_args.model_revision,
        trust_remote_code=model_args.trust_remote_code,
        attn_implementation=model_args.attn_implementation,
        torch_dtype=model_args.torch_dtype,
        use_cache=False if training_args.gradient_checkpointing or
            training_args.fsdp_config.get("activation_checkpointing",
                False) else True,
        device_map=get_kbit_device_map() if quantization_config is not None else None,
        quantization_config=quantization_config,
    )
    training_args.model_init_kwargs = model_kwargs
    tokenizer = AutoTokenizer.from_pretrained(
        model_args.model_name_or_path,
```

```

trust_remote_code=model_args.trust_remote_code, use_fast=True
)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# You can override the template here according to your use case
# tokenizer.chat_template = ...

# Datasets
train_dataset = load_dataset(
    path=script_args.dataset_name,
    name=script_args.dataset_config,
    split=script_args.dataset_train_split,
)
test_dataset = None
if training_args.eval_strategy != "no":
    test_dataset = load_dataset(
        path=script_args.dataset_name,
        name=script_args.dataset_config,
        split=script_args.dataset_test_split,
    )

# Templatize datasets
def template_dataset(sample):
    # return{"text": tokenizer.apply_chat_template(examples["messages"],
tokenizer=False)}
    messages = [
        {"role": "user", "content": sample[question]},
        {"role": "assistant", "content": sample[answer]},
    ]
    return {"text": tokenizer.apply_chat_template(messages, tokenize=False)}

train_dataset = train_dataset.map(template_dataset, remove_columns=["question",
"answer"])
if training_args.eval_strategy != "no":
    # test_dataset = test_dataset.map(template_dataset, remove_columns=
["messages"])
    test_dataset = test_dataset.map(template_dataset, remove_columns=["question",
"answer"])

# Check random samples
with training_args.main_process_first(
    desc="Log few samples from the training set"
):
    for index in random.sample(range(len(train_dataset)), 2):
        print(train_dataset[index]["text"])

# Training
trainer = SFTTrainer(
    model=model_args.model_name_or_path,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    peft_config=get_peft_config(model_args),
    tokenizer=tokenizer,
)

```

```

    if trainer.accelerator.is_main_process and hasattr(trainer.model,
"print_trainable_parameters"):
        trainer.model.print_trainable_parameters()

    checkpoint = None
    if training_args.resume_from_checkpoint is not None:
        checkpoint = training_args.resume_from_checkpoint

    trainer.train(resume_from_checkpoint=checkpoint)

    trainer.save_model(training_args.output_dir)

    with training_args.main_process_first(desc="Training completed"):
        print(f"Training completed, model checkpoint written to {training_args.output_dir}")

```

- b. Optional: If you specify a different model or dataset, edit the **tokenizer.chat_template** parameter to specify the appropriate value for your model and dataset.
 - c. Run the cell to create the training function.
6. Configure the Training Operator SDK client authentication as follows:
 - a. Create a cell with the following content:

```

from kubernetes import client
from kubeflow.training import TrainingClient
from kubeflow.training.models import V1Volume, V1VolumeMount,
V1PersistentVolumeClaimVolumeSource

api_server = "<API_SERVER>"
token = "<TOKEN>"

configuration = client.Configuration()
configuration.host = api_server
configuration.api_key = {"authorization": f"Bearer {token}"}
# Un-comment if your cluster API server uses a self-signed certificate or an un-trusted
CA
#configuration.verify_ssl = False
api_client = client.ApiClient(configuration)
client = TrainingClient(client_configuration=api_client.configuration)

```

- b. Edit the **api_server** and **token** parameters to enter the values to authenticate to your OpenShift cluster.
For information about how to find the server and token details, see [Using the cluster server and token to authenticate](#).
 - c. Run the cell to configure the Training Operator SDK client authentication.
7. Click **File > Save Notebook As**, enter an appropriate file name, and click **Save**.

Verification

1. All cells run successfully.

4.3.2. Running the fine-tuning job

When you run a training job to tune a model, you must specify the resources needed, and provide any authorization credentials required.



NOTE

The code in this procedure specifies how to run the example fine-tuning job. If you have the specified resources, you can run the example code without editing it.

Alternatively, you can modify the example code to specify the appropriate details for your fine-tuning job.

Prerequisites

- You can access an OpenShift cluster that has sufficient worker nodes with supported accelerators to run your training or tuning job.
The example fine-tuning job requires 8 worker nodes, where each worker node has 64 GiB memory, 4 CPUs, and 1 NVIDIA GPU.
- You can access a workbench that is suitable for distributed training, as described in [Creating a workbench for distributed training](#).
- You have administrator access for the data science project.
 - If you created the project, you automatically have administrator access.
 - If you did not create the project, your cluster administrator must give you administrator access.
- You have access to a model.
- You have access to data that you can use to train the model.
- You have configured the fine-tuning job as described in [Configuring the fine-tuning job](#).
- You can access a dynamic storage provisioner that supports ReadWriteMany (RWX) Persistent Volume Claim (PVC) provisioning, such as [Red Hat OpenShift Data Foundation](#).
- A **PersistentVolumeClaim** resource named **shared** with RWX access mode is attached to your workbench.
- You have a Hugging Face account and access token. For more information, search for "user access tokens" in the [Hugging Face documentation](#).

Procedure

1. Open the workbench, as follows:
 - a. Log in to the Red Hat OpenShift AI web console.
 - b. Click **Data science projects** and click your project.
 - c. Click the **Workbenches** tab. If your workbench is not already running, start the workbench.
 - d. Click the **Open** link to open the IDE in a new window.

2. Click **File → Open**, and open the Jupyter notebook that you used to configure the fine-tuning job.
3. Create a cell to run the job, and add the following content:

```
client.create_job(
    job_kind="PyTorchJob",
    name="sft",
    train_func=main,
    num_workers=8,
    num_procs_per_worker="1",
    resources_per_worker={
        "nvidia.com/gpu": 1,
        "memory": "64Gi",
        "cpu": 4,
    },
    base_image="quay.io/modh/training:py311-cuda124-torch251",
    env_vars={
        # Hugging Face
        "HF_HOME": "/mnt/shared/.cache",
        "HF_TOKEN": "",
        # CUDA
        "PYTORCH_CUDA_ALLOC_CONF": "expandable_segments:True",
        # NCCL
        "NCCL_DEBUG": "INFO",
        "NCCL_ENABLE_DMABUF_SUPPORT": "1",
    },
    packages_to_install=[
        "tensorboard",
    ],
    parameters=parameters,
    volumes=[
        V1Volume(name="shared",

persistent_volume_claim=V1PersistentVolumeClaimVolumeSource(claim_name="shared")),
    ],
    volume_mounts=[
        V1VolumeMount(name="shared", mount_path="/mnt/shared"),
    ],
)
```

4. Edit the **HF_TOKEN** value to specify your Hugging Face access token.
Optional: If you specify a different model, and your model is not a gated model from the Hugging Face Hub, remove the **HF_HOME** and **HF_TOKEN** entries.
5. Optional: Edit the other content to specify the appropriate values for your environment, as follows:
 - a. Edit the **num_workers** value to specify the number of worker nodes.
 - b. Update the **resources_per_worker** values according to the job requirements and the resources available.
 - c. The example provided is for NVIDIA GPUs. If you use AMD accelerators, make the following additional changes:

- In the **resources_per_worker** entry, change **nvidia.com/gpu** to **amd.com/gpu**
 - Change the **base_image** value to **quay.io/modh/training:py311-rocm62-torch251**
 - Remove the **CUDA** and **NCCL** entries
- d. If the RWX **PersistentVolumeClaim** resource that is attached to your workbench has a different name instead of **shared**, update the following values to replace **shared** with your PVC name:
- In this cell, update the **HF_HOME** value.
 - In this cell, in the **volumes** entry, update the PVC details:
 - In the **V1Volume** entry, update the **name** and **claim_name** values.
 - In the **volume_mounts** entry, update the **name** and **mount_path** values.
 - In the cell where you set the training parameters, update the **output_dir** value. For more information about setting the training parameters, see [Configuring the fine-tuning job](#).
6. Run the cell to run the job.

Verification

View the progress of the job as follows:

1. Create a cell with the following content:

```
client.get_job_logs(
    name="sft",
    job_kind="PyTorchJob",
    follow=True,
)
```

2. Run the cell to view the job progress.

4.3.3. Deleting the fine-tuning job

When you no longer need the fine-tuning job, delete the job to release the resources.



NOTE

The code in this procedure specifies how to delete the example fine-tuning job. If you created the example fine-tuning job named **sft**, you can run the example code without editing it.

Alternatively, you can modify this example code to specify the name of your fine-tuning job.

Prerequisites

- You have created a fine-tuning job as described in [Running the fine-tuning job](#).

Procedure

Procedure

1. Open the workbench, as follows:
 - a. Log in to the Red Hat OpenShift AI web console.
 - b. Click **Data science projects** and click your project.
 - c. Click the **Workbenches** tab. If your workbench is not already running, start the workbench.
 - d. Click the **Open** link to open the IDE in a new window.
2. Click **File → Open**, and open the Jupyter notebook that you used to configure and run the example fine-tuning job.
3. Create a cell with the following content:

```
client.delete_job(name="sft")
```
4. Optional: If you want to delete a different job, edit the content to replace **sft** with the name of your job.
5. Run the cell to delete the job.

Verification

1. In the OpenShift Console, in the **Administrator** perspective, click **Workloads → Jobs**
2. From the **Project** list, select your project.
3. Verify that the specified job is not listed.

4.4. CREATING A MULTI-NODE PYTORCH TRAINING JOB WITH RDMA

NVIDIA GPUDirect RDMA uses Remote Direct Memory Access (RDMA) to provide direct GPU interconnect, enabling peripheral devices to access NVIDIA GPU memory in remote systems directly. RDMA improves the training job performance because it eliminates the overhead of using the operating system CPUs and memory. Running a training job on multiple nodes using multiple GPUs can significantly reduce the completion time.

In Red Hat OpenShift AI, NVIDIA GPUs can communicate directly by using GPUDirect RDMA across the following types of network:

- Ethernet: RDMA over Converged Ethernet (RoCE)
- InfiniBand

Before you create a PyTorch training job in a cluster configured for RDMA, you must configure the job to use the high-speed network interfaces.

Prerequisites

- You can access an OpenShift cluster that has multiple worker nodes with supported NVIDIA GPUs.
- Your cluster administrator has configured the cluster as follows:

- Installed Red Hat OpenShift AI with the required distributed training components, as described in [Installing the distributed workloads components](#).
- Configured the distributed training resources, as described in [Managing distributed workloads](#).
- Configured the cluster for RDMA, as described in [Configuring a cluster for RDMA](#).

Procedure

1. Log in to the OpenShift Console.
2. Create a **PyTorchJob** resource, as follows:
 - a. In the **Administrator** perspective, click **Home → Search**.
 - b. From the **Project** list, select your project.
 - c. Click the **Resources** list, and in the search field, start typing **PyTorchJob**.
 - d. Select **PyTorchJob**, and click **Create PyTorchJob**.
The **Create PyTorchJob** page opens, with default YAML code automatically added.
3. Attach the high-speed network interface to the **PyTorchJob** pods, as follows:
 - a. Edit the **PyTorchJob** resource YAML code to include an annotation that adds the pod to an additional network, as shown in the following example:

Example annotation to attach network interface to pod

```
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            k8s.v1.cni.cncf.io/networks: "example-net"
```

- b. Replace the example network name **example-net** with the appropriate value for your configuration.
4. Configure the job to use NVIDIA Collective Communications Library (NCCL) interfaces, as follows:
 - a. Edit the **PyTorchJob** resource YAML code to add the following environment variables:

Example environment variables

```
spec:
  containers:
    - command:
      - /bin/bash
      - -C
      - "your container command"
```

```
env:
  - name: NCCL_SOCKET_IFNAME
    value: "net1"
  - name: NCCL_IB_HCA
    value: "mlx5_1"
```

- b. Replace the example environment-variable values with the appropriate values for your configuration:
 - i. Set the ***NCCL_SOCKET_IFNAME*** environment variable to specify the IP interface to use for communication.
 - ii. [Optional] To explicitly specify the Host Channel Adapter (HCA) that NCCL should use, set the ***NCCL_IB_HCA*** environment variable.
5. Specify the base training image name, as follows:
 - a. Edit the **PyTorchJob** resource YAML code to add the following text:

Example base training image

```
image: quay.io/modh/training:py311-cuda124-torch251
```

- b. If you want to use a different base training image, replace the image name accordingly. For a list of supported training images, see [Red Hat OpenShift AI: Supported Configurations](#).
6. Specify the requests and limits for the network interface resources.
The name of the resource varies, depending on the NVIDIA Network Operator configuration. The resource name might depend on the deployment mode, and is specified in the **NicClusterPolicy** resource.



NOTE

You must use the resource name that matches your configuration. The name must correspond to the value advertised by the NVIDIA Network Operator on the cluster nodes.

The following example is for RDMA over Converged Ethernet (RoCE), where the Ethernet RDMA devices are using the RDMA shared device mode.

- a. Review the **NicClusterPolicy** resource to identify the **resourceName** value.

Example NicClusterPolicy

```
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
spec:
  rdmaSharedDevicePlugin:
    config: |
      {
        "configList": [
          {
            "resourceName": "rdma_shared_device_eth",
            "rdmaHcaMax": 63,
```

```

    "selectors": {
      "ifNames": ["ens8f0np0"]
    }
  ]
}

```

In this example **NicClusterPolicy** resource, the **resourceName** value is **rdma_shared_device_eth**.

- b. Edit the **PyTorchJob** resource YAML code to add the following text:

Example requests and limits for the network interface resources

```

resources:
  limits:
    nvidia.com/gpu: "1"
    rdma/rdma_shared_device_eth: "1"
  requests:
    nvidia.com/gpu: "1"
    rdma/rdma_shared_device_eth: "1"

```

- c. In the **limits** and **requests** sections, replace the resource name with the resource name from your **NicClusterPolicy** resource (in this example, **rdma_shared_device_eth**).
- d. Replace the specified value **1** with the number that you require. Ensure that the specified amount is available on your OpenShift cluster.
7. Repeat the above steps to make the same edits in the **Worker** section of the **PyTorchJob** YAML code.
8. Click **Create**.

You have created a multi-node PyTorch training job that is configured to run with RDMA.

You can see the entire YAML code for this example **PyTorchJob** resource in the [Example Training Operator PyTorchJob resource configured to run with RDMA](#).

Verification

1. In the OpenShift Console, open the **Administrator** perspective.
2. From the **Project** list, select your project.
3. Click **Home → Search → PyTorchJob** and verify that the job was created.
4. Click **Workloads → Pods** and verify that requested head pod and worker pods are running.

Additional resources

- [Attaching a pod to a secondary network](#) in the OpenShift documentation
- [NCCL environment variables](#) in the NVIDIA documentation
- [NVIDIA Network Operator deployment examples](#) in the NVIDIA documentation

- [NCCL Troubleshooting](#) in the NVIDIA documentation

4.5. EXAMPLE TRAINING OPERATOR PYTORCHJOB RESOURCE CONFIGURED TO RUN WITH RDMA

This example shows how to create a Training Operator PyTorch training job that is configured to run with Remote Direct Memory Access (RDMA).

```
apiVersion: kubeflow.org/v1
kind: PyTorchJob
metadata:
  name: job
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            k8s.v1.cni.cncf.io/networks: "example-net"
        spec:
          containers:
            - command:
                - /bin/bash
                - -c
                - "your container command"
              env:
                - name: NCCL_SOCKET_IFNAME
                  value: "net1"
                - name: NCCL_IB_HCA
                  value: "mlx5_1"
              image: quay.io/modh/training:py311-cuda124-torch251
              name: pytorch
            resources:
              limits:
                nvidia.com/gpu: "1"
                rdma/rdma_shared_device_eth: "1"
              requests:
                nvidia.com/gpu: "1"
                rdma/rdma_shared_device_eth: "1"
    Worker:
      replicas: 3
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            k8s.v1.cni.cncf.io/networks: "example-net"
        spec:
          containers:
            - command:
                - /bin/bash
                - -c
                - "your container command"
              env:
```

```
- name: NCCL_SOCKET_IFNAME
  value: "net1"
- name: NCCL_IB_HCA
  value: "mlx5_1"
image: quay.io/modh/training:py311-cuda124-torch251
name: pytorch
resources:
  limits:
    nvidia.com/gpu: "1"
    rdma/rdma_shared_device_eth: "1"
  requests:
    nvidia.com/gpu: "1"
    rdma/rdma_shared_device_eth: "1"
```

CHAPTER 5. MONITORING DISTRIBUTED WORKLOADS

In OpenShift AI, you can view project metrics for distributed workloads, and view the status of all distributed workloads in the selected project. You can use these metrics to monitor the resources used by distributed workloads, assess whether project resources are allocated correctly, track the progress of distributed workloads, and identify corrective action when necessary.



NOTE

Data science pipelines workloads are not managed by the distributed workloads feature, and are not included in the distributed workloads metrics.

5.1. VIEWING PROJECT METRICS FOR DISTRIBUTED WORKLOADS

In OpenShift AI, you can view the following project metrics for distributed workloads:

- **CPU** – The number of CPU cores that are currently being used by all distributed workloads in the selected project.
- **Memory** – The amount of memory in gibibytes (GiB) that is currently being used by all distributed workloads in the selected project.

You can use these metrics to monitor the resources used by the distributed workloads, and assess whether project resources are allocated correctly.

Prerequisites

- You have installed Red Hat OpenShift AI.
- On the OpenShift cluster where OpenShift AI is installed, user workload monitoring is enabled.
- You have logged in to Red Hat OpenShift AI.
- Your data science project contains distributed workloads.

Procedure

1. In the OpenShift AI left navigation pane, click **Distributed workloads**.
2. From the **Project** list, select the project that contains the distributed workloads that you want to monitor.
3. Click the **Project metrics** tab.
4. Optional: From the **Refresh interval** list, select a value to specify how frequently the graphs on the metrics page are refreshed to show the latest data.
You can select one of these values: **15 seconds**, **30 seconds**, **1 minute**, **5 minutes**, **15 minutes**, **30 minutes**, **1 hour**, **2 hours**, or **1 day**.
5. In the **Requested resources** section, review the **CPU** and **Memory** graphs to identify the resources requested by distributed workloads as follows:
 - Requested by the selected project

- Requested by all projects, including the selected project and projects that you cannot access
- Total shared quota for all projects, as provided by the cluster queue

For each resource type (**CPU** and **Memory**), subtract the **Requested by all projects** value from the **Total shared quota** value to calculate how much of that resource quota has not been requested and is available for all projects.

6. Scroll down to the **Top resource-consuming distributed workloads** section to review the following graphs:
 - Top 5 distributed workloads that are consuming the most CPU resources
 - Top 5 distributed workloads that are consuming the most memory

You can also identify how much CPU or memory is used in each case.

7. Scroll down to view the **Distributed workload resource metrics** table, which lists all of the distributed workloads in the selected project, and indicates the current resource usage and the status of each distributed workload.
In each table entry, progress bars indicate how much of the requested CPU and memory is currently being used by this distributed workload. To see numeric values for the actual usage and requested usage for CPU (measured in cores) and memory (measured in GiB), hover the cursor over each progress bar. Compare the actual usage with the requested usage to assess the distributed workload configuration. If necessary, reconfigure the distributed workload to reduce or increase the requested resources.

Verification

On the **Project metrics** tab, the graphs and table provide resource-usage data for the distributed workloads in the selected project.

5.2. VIEWING THE STATUS OF DISTRIBUTED WORKLOADS

In OpenShift AI, you can view the status of all distributed workloads in the selected project. You can track the progress of the distributed workloads, and identify corrective action when necessary.

Prerequisites

- You have installed Red Hat OpenShift AI.
- On the OpenShift cluster where OpenShift AI is installed, user workload monitoring is enabled.
- You have logged in to Red Hat OpenShift AI.
- Your data science project contains distributed workloads.

Procedure

1. In the OpenShift AI left navigation pane, click **Distributed workloads**.
2. From the **Project** list, select the project that contains the distributed workloads that you want to monitor.
3. Click the **Distributed workload status** tab.

4. Optional: From the **Refresh interval** list, select a value to specify how frequently the graphs on the metrics page are refreshed to show the latest data.
You can select one of these values: **15 seconds**, **30 seconds**, **1 minute**, **5 minutes**, **15 minutes**, **30 minutes**, **1 hour**, **2 hours**, or **1 day**.
5. In the **Status overview** section, review a summary of the status of all distributed workloads in the selected project.
The status can be **Pending**, **Inadmissible**, **Admitted**, **Running**, **Evicted**, **Succeeded**, or **Failed**.
6. Scroll down to view the **Distributed workloads** table, which lists all of the distributed workloads in the selected project. The table provides the priority, status, creation date, and latest message for each distributed workload.
The latest message provides more information about the current status of the distributed workload. Review the latest message to identify any corrective action needed. For example, a distributed workload might be **Inadmissible** because the requested resources exceed the available resources. In such cases, you can either reconfigure the distributed workload to reduce the requested resources, or reconfigure the cluster queue for the project to increase the resource quota.

Verification

On the **Distributed workload status** tab, the graph provides a summarized view of the status of all distributed workloads in the selected project, and the table provides more details about the status of each distributed workload.

5.3. VIEWING KUEUE ALERTS FOR DISTRIBUTED WORKLOADS

In OpenShift AI, you can view Kueue alerts for your cluster. Each alert provides a link to a *runbook*. The runbook provides instructions on how to resolve the situation that triggered the alert.

Prerequisites

- You have logged in to OpenShift with the **cluster-admin** role.
- You can access a data science cluster that is configured to run distributed workloads as described in [Managing distributed workloads](#).
- You can access a data science project that contains a workbench, and the workbench is running a default workbench image that contains the CodeFlare SDK, for example, the **Standard Data Science** workbench. For information about projects and workbenches, see [Working on data science projects](#).
- You have logged in to Red Hat OpenShift AI.
- Your data science project contains distributed workloads.

Procedure

1. In the OpenShift console, in the **Administrator** perspective, click **Observe → Alerting**.
2. Click the **Alerting rules** tab to view a list of alerting rules for default and user-defined projects.
 - The **Severity** column indicates whether the alert is informational, a warning, or critical.
 - The **Alert state** column indicates whether a rule is currently firing.

3. Click the name of an alerting rule to see more details, such as the condition that triggers the alert. The following table summarizes the alerting rules for Kueue resources.

Table 5.1. Alerting rules for Kueue resources

| Severity | Name | Alert condition |
|----------|--|--|
| Critical | KueuePodDown | The Kueue pod is not ready for a period of 5 minutes. |
| Info | LowClusterQueueResourceUsage | Resource usage in the cluster queue is below 20% of its nominal quota for more than 1 day. Resource usage refers to any resources listed in the cluster queue, such as CPU, memory, and so on. |
| Info | ResourceReservationExceedsQuota | Resource reservation is 10 times the available quota in the cluster queue. Resource reservation refers to any resources listed in the cluster queue, such as CPU, memory, and so on. |
| Info | PendingWorkloadPods | A pod has been in a Pending state for more than 3 days. |

4. If the **Alert state** of an alerting rule is set to **Firing**, complete the following steps:
 - a. Click **Observe → Alerting** and then click the **Alerts** tab.
 - b. Click each alert for the firing rule, to see more details. Note that a separate alert is fired for each resource type affected by the alerting rule.
 - c. On the alert details page, in the **Runbook** section, click the link to open a GitHub page that provides troubleshooting information.
 - d. Complete the runbook steps to identify the cause of the alert and resolve the situation.

Verification

After you resolve the cause of the alert, the alerting rule stops firing.

CHAPTER 6. TROUBLESHOOTING COMMON PROBLEMS WITH DISTRIBUTED WORKLOADS FOR USERS

If you are experiencing errors in Red Hat OpenShift AI relating to distributed workloads, read this section to understand what could be causing the problem, and how to resolve the problem.

If the problem is not documented here or in the release notes, contact Red Hat Support.

6.1. MY RAY CLUSTER IS IN A SUSPENDED STATE

Problem

The resource quota specified in the cluster queue configuration might be insufficient, or the resource flavor might not yet be created.

Diagnosis

The Ray cluster head pod or worker pods remain in a suspended state.

Resolution

1. In the OpenShift console, select your project from the **Project** list.
2. Check the workload resource:
 - a. Click **Search**, and from the **Resources** list, select **Workload**.
 - b. Select the workload resource that is created with the Ray cluster resource, and click the **YAML** tab.
 - c. Check the text in the **status.conditions.message** field, which provides the reason for the suspended state, as shown in the following example:

```
status:
conditions:
  - lastTransitionTime: '2024-05-29T13:05:09Z'
    message: 'couldn't assign flavors to pod set small-group-jobtest12: insufficient quota
for nvidia.com/gpu in flavor default-flavor in ClusterQueue'
```
3. Check the Ray cluster resource:
 - a. Click **Search**, and from the **Resources** list, select **RayCluster**.
 - b. Select the Ray cluster resource, and click the **YAML** tab.
 - c. Check the text in the **status.conditions.message** field.
4. Check the cluster queue resource:
 - a. Click **Search**, and from the **Resources** list, select **ClusterQueue**.
 - b. Check your cluster queue configuration to ensure that the resources that you requested are within the limits defined for the project.
 - c. Either reduce your requested resources, or contact your administrator to request more resources.

6.2. MY RAY CLUSTER IS IN A FAILED STATE

Problem

You might have insufficient resources.

Diagnosis

The Ray cluster head pod or worker pods are not running. When a Ray cluster is created, it initially enters a **failed** state. This failed state usually resolves after the reconciliation process completes and the Ray cluster pods are running.

Resolution

If the failed state persists, complete the following steps:

1. In the OpenShift console, select your project from the **Project** list.
2. Click **Search**, and from the **Resources** list, select **Pod**.
3. Click your pod name to open the pod details page.
4. Click the **Events** tab, and review the pod events to identify the cause of the problem.
5. If you cannot resolve the problem, contact your administrator to request assistance.

6.3. I SEE A "FAILED TO CALL WEBHOOK" ERROR MESSAGE FOR THE CODEFLARE OPERATOR

Problem

After you run the **cluster.up()** command, the following error is shown:

```
ApiException: (500)
Reason: Internal Server Error
HTTP response body: {"kind":"Status","apiVersion":"v1","metadata":
{"status":"Failure","message":"Internal error occurred: failed calling webhook
\"mraycluster.ray.openshift.ai\": failed to call webhook: Post \"https://codeflare-operator-webhook-
service.redhat-ods-applications.svc:443/mutate-ray-io-v1-raycluster?timeout=10s\": no endpoints
available for service \"codeflare-operator-webhook-service\"\",\"reason\":\"InternalError\",\"details\":
{\"causes\": [{\"message\":\"failed calling webhook \"mraycluster.ray.openshift.ai\": failed to call webhook:
Post \"https://codeflare-operator-webhook-service.redhat-ods-applications.svc:443/mutate-ray-io-v1-
raycluster?timeout=10s\": no endpoints available for service \"codeflare-operator-webhook-
service\""}]},\"code\":500}
```

Diagnosis

The CodeFlare Operator pod might not be running.

Resolution

Contact your administrator to request assistance.

6.4. I SEE A "FAILED TO CALL WEBHOOK" ERROR MESSAGE FOR KUEUE

Problem

After you run the **cluster.up()** command, the following error is shown:

```
ApiException: (500)
Reason: Internal Server Error
HTTP response body: {"kind":"Status","apiVersion":"v1","metadata":
{},"status":"Failure","message":"Internal error occurred: failed calling webhook \"mraycluster.kb.io\":
failed to call webhook: Post \"https://kueue-webhook-service.redhat-ods-applications.svc:443/mutate-
ray-io-v1-raycluster?timeout=10s\": no endpoints available for service \"kueue-webhook-
service\"","reason":"InternalError","details":{"causes":[{"message":"failed calling webhook
\"mraycluster.kb.io\": failed to call webhook: Post \"https://kueue-webhook-service.redhat-ods-
applications.svc:443/mutate-ray-io-v1-raycluster?timeout=10s\": no endpoints available for service
\"kueue-webhook-service\""}]},"code":500}
```

Diagnosis

The Kueue pod might not be running.

Resolution

Contact your administrator to request assistance.

6.5. MY RAY CLUSTER DOES NOT START

Problem

After you run the **cluster.up()** command, when you run either the **cluster.details()** command or the **cluster.status()** command, the Ray Cluster remains in the **Starting** status instead of changing to the **Ready** status. No pods are created.

Diagnosis

1. In the OpenShift console, select your project from the **Project** list.
2. Check the workload resource:
 - a. Click **Search**, and from the **Resources** list, select **Workload**.
 - b. Select the workload resource that is created with the Ray cluster resource, and click the **YAML** tab.
 - c. Check the text in the **status.conditions.message** field, which provides the reason for remaining in the **Starting** state.
3. Check the Ray cluster resource:
 - a. Click **Search**, and from the **Resources** list, select **RayCluster**.
 - b. Select the Ray cluster resource, and click the **YAML** tab.
 - c. Check the text in the **status.conditions.message** field.

Resolution

If you cannot resolve the problem, contact your administrator to request assistance.

6.6. I SEE A "DEFAULT LOCAL QUEUE NOT FOUND" ERROR MESSAGE

Problem

After you run the **cluster.up()** command, the following error is shown:

Default Local Queue with kueue.x-k8s.io/default-queue: **true** annotation not found please create a default Local Queue or provide the local_queue name in Cluster Configuration.

Diagnosis

No default local queue is defined, and a local queue is not specified in the cluster configuration.

Resolution

1. In the OpenShift console, select your project from the **Project** list.
2. Click **Search**, and from the **Resources** list, select **LocalQueue**.
3. Resolve the problem in one of the following ways:

- If a local queue exists, add it to your cluster configuration as follows:

```
local_queue="<local_queue_name>"
```

- If no local queue exists, contact your administrator to request assistance.

6.7. I SEE A "LOCAL_QUEUE PROVIDED DOES NOT EXIST" ERROR MESSAGE

Problem

After you run the **cluster.up()** command, the following error is shown:

local_queue provided does not exist or is not in this namespace. Please provide the correct local_queue name in Cluster Configuration.

Diagnosis

An incorrect value is specified for the local queue in the cluster configuration, or an incorrect default local queue is defined. The specified local queue either does not exist, or exists in a different namespace.

Resolution

1. In the OpenShift console, select your project from the **Project** list.
2. Click **Search**, and from the **Resources** list, select **LocalQueue**.
3. Resolve the problem in one of the following ways:
 - If a local queue exists, ensure that you spelled the local queue name correctly in your cluster configuration, and that the **namespace** value in the cluster configuration matches your project name. If you do not specify a **namespace** value in the cluster configuration, the Ray cluster is created in the current project.

- If no local queue exists, contact your administrator to request assistance.

6.8. I CANNOT CREATE A RAY CLUSTER OR SUBMIT JOBS

Problem

After you run the **cluster.up()** command, an error similar to the following error is shown:

```
RuntimeError: Failed to get RayCluster CustomResourceDefinition: (403)
Reason: Forbidden
HTTP response body: {"kind":"Status","apiVersion":"v1","metadata":
{"status":"Failure","message":"rayclusters.ray.io is forbidden: User
\"system:serviceaccount:regularuser-project:regularuser-workbench\" cannot list resource
\"rayclusters\" in API group \"ray.io\" in the namespace \"regularuser-
project\"","reason":"Forbidden","details":{"group\":\"ray.io\",\"kind\":\"rayclusters"},"code":403}
```

Diagnosis

The correct OpenShift login credentials are not specified in the **TokenAuthentication** section of your notebook code.

Resolution

1. Identify the correct OpenShift login credentials as follows:
 - a. In the OpenShift console header, click your username and click **Copy login command**.
 - b. In the new tab that opens, log in as the user whose credentials you want to use.
 - c. Click **Display Token**.
 - d. From the **Log in with this token** section, copy the **token** and **server** values.
2. In your notebook code, specify the copied **token** and **server** values as follows:

```
auth = TokenAuthentication(
    token = "<token>",
    server = "<server>",
    skip_tls=False
)
auth.login()
```

6.9. MY POD PROVISIONED BY KUEUE IS TERMINATED BEFORE MY IMAGE IS PULLED

Problem

Kueue waits for a period of time before marking a workload as ready, to enable all of the workload pods to become provisioned and running. By default, Kueue waits for 5 minutes. If the pod image is very large and is still being pulled after the 5-minute waiting period elapses, Kueue fails the workload and terminates the related pods.

Diagnosis

1. In the OpenShift console, select your project from the **Project** list.
2. Click **Search**, and from the **Resources** list, select **Pod**.
3. Click the Ray head pod name to open the pod details page.
4. Click the **Events** tab, and review the pod events to check whether the image pull completed successfully.

Resolution

If the pod takes more than 5 minutes to pull the image, contact your administrator to request assistance.