Dask Kubernetes Documentation

Release 0.1.dev53+g02eea0fdf

Dask kubernetes Developers

Mar 02, 2026

GETTING STARTED

1 Quickstart **3**

2 What is the operator? **5**

3 What resources does the operator manage? **7**

3.1 Worker Groups 7

3.2 Clusters 7

3.3 Jobs 7

3.4 Autoscalers 8

Index **47**

Welcome to the documentation for the Dask Kubernetes Operator.

> **Note**
>
> If you are looking for high-level documentation on deploying Dask on Kubernetes new users should head to the Dask documentation page on Kubernetes.

The package `dask-kubernetes` provides a Dask operator for Kubernetes. `dask-kubernetes` is one of many options for deploying Dask clusters, see Deploying Dask in the Dask documentation for an overview of additional options.

CHAPTER

ONE

QUICKSTART

KubeCluster deploys Dask clusters on Kubernetes clusters using custom Kubernetes resources. It is designed to dynamically launch ad-hoc deployments.

```
$ # Install operator CRDs and controller, needs to be done once on your Kubernetes␣
↪cluster
$ helm install --repo https://helm.dask.org --create-namespace -n dask-operator --
↪generate-name dask-kubernetes-operator
```

```
$ # Install dask-kubernetes
$ pip install dask-kubernetes
```

```
from dask_kubernetes.operator import KubeCluster
cluster = KubeCluster(name="my-dask-cluster", image='ghcr.io/dask/dask:latest')
cluster.scale(10)
```

CHAPTER

TWO

WHAT IS THE OPERATOR?

The Dask Operator is a set of custom resources and a controller that runs on your Kubernetes cluster and allows you to create and manage your Dask clusters as Kubernetes resources. Creating clusters can either be done via the *Kubernetes API with kubectl* or the *Python API with KubeCluster*.

To *install the operator* you need to apply some custom resource definitions that allow us to describe Dask resources and the operator itself which is a small Python application that watches the Kubernetes API for events related to our custom resources and creates other resources such as `Pods` and `Services` accordingly.

CHAPTER

THREE

WHAT RESOURCES DOES THE OPERATOR MANAGE?

The operator manages a hierarchy of resources, some custom resources to represent Dask primitives like clusters and worker groups, and native Kubernetes resources such as pods and services to run the cluster processes and facilitate communication.

3.1 Worker Groups

A `DaskWorkerGroup` represents a homogenous group of workers that can be scaled. The resource is similar to a native Kubernetes `Deployment` in that it manages a group of workers with some intelligence around the `Pod` lifecycle. A worker group must be attached to a Dask Cluster resource in order to function.

All Kubernetes annotations on the `DaskWorkerGroup` resource will be passed onto worker `Pod` resources. Annotations created by *kopf* or *kubectl* (i.e. starting with "kopf.zalando.org" or "kubectl.kubernetes.io") will not be passed on.

3.2 Clusters

The `DaskCluster` custom resource creates a Dask cluster by creating a scheduler `Pod`, scheduler `Service` and default `DaskWorkerGroup` which in turn creates worker `Pod` resources.

Workers connect to the scheduler via the scheduler `Service` and that service can also be exposed to the user in order to connect clients and perform work.

The operator also has support for creating additional worker groups. These are extra groups of workers with different configuration settings and can be scaled separately. You can then use resource annotations to schedule different tasks to different groups.

All *Kubernetes annotations <https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/>* on the `DaskCluster` resource will be passed onto the scheduler `Pod` and `Service` as well the `DaskWorkerGroup` resources. Annotations created by *kopf* or *kubectl* (i.e. starting with "kopf.zalando.org" or "kubectl.kubernetes.io") will not be passed on.

For example you may wish to have a smaller pool of workers that have more memory for memory intensive tasks, or GPUs for compute intensive tasks.

3.3 Jobs

A `DaskJob` is a batch style resource that creates a `Pod` to perform some specific task from start to finish alongside a `DaskCluster` that can be leveraged to perform the work.

All *Kubernetes annotations <https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/>* on the `DaskJob` resource will be passed on to the job-runner `Pod` resource. If one also wants to set Kubernetes annotations on the cluster-related resources (scheduler and worker `Pods`), these can be set as `spec.cluster.metadata`

in the DaskJob resource. Annotations created by *kopf* or *kubectl* (i.e. starting with "kopf.zalando.org" or "kubectl.kubernetes.io") will not be passed on.

Once the job Pod runs to completion the cluster is removed automatically to save resources. This is great for workflows like training a distributed machine learning model with Dask.

3.4 Autoscalers

A DaskAutoscaler resource will communicate with the scheduler periodically and auto scale the default DaskWorkerGroup to the desired number of workers.

```
from dask_kubernetes.operator import KubeCluster
cluster = KubeCluster(name="my-dask-cluster", image='ghcr.io/dask/dask:latest')
cluster.adapt(minimum=1, maximum=10)
```

3.4.1 Installing

Python package

You can install dask-kubernetes with pip, conda, or by installing from source.

Pip

Pip can be used to install dask-kubernetes and its Python dependencies:

```
pip install dask-kubernetes --upgrade  # Install everything from last released version
```

Conda

To install the latest version of dask-kubernetes from the conda-forge repository using conda:

```
conda install dask-kubernetes -c conda-forge
```

Install from Source

To install dask-kubernetes from source, clone the repository from github:

```
git clone https://github.com/dask/dask-kubernetes.git
cd dask-kubernetes
python setup.py install
```

or use pip locally if you want to install all dependencies as well:

```
pip install -e .
```

You can also install directly from git main branch:

```
pip install git+https://github.com/dask/dask-kubernetes
```

Operator

To use the Dask Operator you must install the custom resource definitions, service account, roles, and the operator controller deployment.

Quickstart

```
$ helm install --repo https://helm.dask.org --create-namespace -n dask-operator --
↪generate-name dask-kubernetes-operator
```

Installing with Helm

The operator has a Helm chart which can be used to manage the installation of the operator. The chart is published in the Dask Helm repo repository, and can be installed via:

```
$ helm repo add dask https://helm.dask.org
"dask" has been added to your repositories

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "dask" chart repository
Update Complete. Happy Helming!

$ helm install --create-namespace -n dask-operator --generate-name dask/dask-kubernetes-
↪operator
NAME: dask-kubernetes-operator-1666875935
NAMESPACE: dask-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Operator has been installed successfully.
```

Then you should be able to list your Dask clusters via `kubectl`.

```
$ kubectl get daskclusters
No resources found in default namespace.
```

We can also check the operator pod is running:

```
$ kubectl get pods -A -l app.kubernetes.io/name=dask-kubernetes-operator
NAMESPACE       NAME                                        READY   STATUS    RESTARTS  ␣
↪AGE
dask-operator   dask-kubernetes-operator-775b8bbbd5-zdrf7   1/1     Running   0         ␣
↪74s
```

> **Warning**
>
> Please note that Helm does not support updating or deleting CRDs. If updates are made to the CRD templates in future releases (to support future k8s releases, for example) you may have to manually update the CRDs or delete/reinstall the Dask Operator.

Single namespace

By default the controller is installed with a `ClusterRole` and watches all namespaces. You can also just install it into a single namespace by setting the following options.

```
$ helm install -n my-namespace --generate-name dask/dask-kubernetes-operator --set rbac.
↪cluster=false --set kopfArgs="{--namespace=my-namespace}"
NAME: dask-kubernetes-operator-1749875935
NAMESPACE: my-namespace
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Operator has been installed successfully.
```

Prometheus

The operator helm chart also contains some optional *ServiceMonitor* and *PodMonitor* resources to enable Prometheus scraping of Dask components. As not all clusters have the Prometheus operator installed these are disabled by default. You can enable them with the following comfig options.

```
metrics:
   scheduler:
      enabled: true
      serviceMonitor:
         enabled: true
   worker:
      enabled: true
      serviceMonitor:
         enabled: true
```

You'll also need to ensure the container images you choose for your Dask components have the `prometheus_client` library installed. If you're using the official Dask images you can install this at runtime.

```
from dask_kubernetes.operator import KubeCluster
cluster = KubeCluster(name="monitored", env={"EXTRA_PIP_PACKAGES": "prometheus_client"})
```

Chart Configuration Reference

Dask-kubernetes-operator

A helm chart for managing the deployment of the dask kubernetes operator and CRDs

Configuration

The following table lists the configurable parameters of the Dask-kubernetes-operator chart and their default values.

Parameter	Description	Default
`image.name`	Docker image for the operator	`"ghcr.io/dask/dask-kubernetes-operator"`
`image.tag`	Release version	`"set-by-chartpress"`
`image.pullPolicy`	Pull policy	`"IfNotPresent"`
`imagePullSecrets`	Image pull secrets for private registries	`[]`
`nameOverride`	Override release name (not including random UUID)	`""`
`fullnameOverride`	Override full release name	`""`
`serviceAccount.create`	Create a service account for the operator to use	`true`
`serviceAccount.annotations`	Annotations to add to the service account	`{}`
`serviceAccount.name`	The name of the service account to use. If not set and create is true, a name is generated using the fullname template.	`""`
`rbac.create`	Create a Role/ClusterRole needed by the operator and bind it to the service account	`true`
`rbac.cluster`	Creates a ClusterRole if true, else create a namespaced Role	`true`
`podAnnotations`	Extra annotations for the operator pod	`{}`
`podSecurityContext`	Security context for the operator pod	`{}`
`securityContext.capabilities.drop`		`["ALL"]`
`securityContext.runAsNonRoot`		`true`
`securityContext.runAsUser`		`1000`
`securityContext.allowPrivilegeEscalation`		`false`
`securityContext.readOnlyRootFilesystem`		`true`
`resources`	Resources for the operator pod	`{}`
`volumes`	Volumes for the operator pod	`[]`
`volumeMounts`	Volume mounts for the operator container	`[]`
`extraEnv`	Extra environment variables for the operator container	`[]`
`nodeSelector`	Node selector	`{}`
`tolerations`	Tolerations	`[]`
`affinity`	Affinity	`{}`
`priorityClassName`	Priority class	`null`
`livenessProbe.timeoutSeconds`		`5`

continues on next page

Table 1 – continued from previous page

Parameter	Description	Default
`kopfArgs`	Command line flags to pass to kopf on start up	`["--all-namespaces"]`
`metrics.scheduler.enabled`	Enable scheduler metrics. Pip package [prometheus-client](https://pypi.org/project/prometheus-client/) should be present on scheduler.	`false`
`metrics.scheduler.serviceMonitor.enabled`	Enable scheduler servicemonitor.	`false`
`metrics.scheduler.serviceMonitor.namespace`	Deploy servicemonitor in different namespace, e.g. monitoring.	`""`
`metrics.scheduler.serviceMonitor.namespaceSelector`	Selector to select which namespaces the Endpoints objects are discovered from.	`{}`
`metrics.scheduler.serviceMonitor.additionalLabels`	Additional labels to add to the ServiceMonitor metadata.	`{}`
`metrics.scheduler.serviceMonitor.interval`	Interval at which metrics should be scraped.	`"15s"`
`metrics.scheduler.serviceMonitor.jobLabel`	The label to use to retrieve the job name from.	`""`
`metrics.scheduler.serviceMonitor.targetLabels`	TargetLabels transfers labels on the Kubernetes Service onto the target.	`["dask.org/cluster-name"]`
`metrics.scheduler.serviceMonitor.metricRelabelings`	MetricRelabelConfigs to apply to samples before ingestion.	`[]`
`metrics.worker.enabled`	Enable workers metrics. Pip package [prometheus-client](https://pypi.org/project/prometheus-client/) should be present on workers.	`false`
`metrics.worker.podMonitor.enabled`	Enable workers podmonitor	`false`
`metrics.worker.podMonitor.namespace`	Deploy podmonitor in different namespace, e.g. monitoring.	`""`
`metrics.worker.podMonitor.namespaceSelector`	Selector to select which namespaces the Endpoints objects are discovered from.	`{}`
`metrics.worker.podMonitor.additionalLabels`	Additional labels to add to the PodMonitor metadata.	`{}`
`metrics.worker.podMonitor.interval`	Interval at which metrics should be scraped.	`"15s"`
`metrics.worker.podMonitor.jobLabel`	The label to use to retrieve the job name from.	`""`
`metrics.worker.podMonitor.podTargetLabels`	PodTargetLabels transfers labels on the Kubernetes Pod onto the target.	`["dask.org/cluster-name", "dask.org/workergroup-name"]`
`metrics.worker.podMonitor.metricRelabelings`	MetricRelabelConfigs to apply to samples before ingestion.	`[]`
`workerAllocation.size`		`null`

continues on next page

Table 1 – continued from previous page

Parameter	Description	Default
workerAllocation.delay		null

Helm Chart reference generated by sphinx-helm <https://sphinx-helm.readthedocs.io>`_.

Installing with Manifests

If you prefer to install the operator from static manifests with `kubectl` and set configuration options with tools like `kustomize` you can generate the default manifests with:

```
$ helm template --include-crds --repo https://helm.dask.org release dask-kubernetes-
↪operator | kubectl apply -f -
```

Working with JupyterHub

In order to access the Dask Operator's dashboard with JupyterHub, some additional configuration is needed.

Configuring JupyterHub Network Policy

The default JupyterHub configuration doesn't allow communication between notebook pods and the Kubernetes API. Consider that *kubeconfig* is already configured for your notebook pods, and in order to enable the creation of `DaskCluster` resources directly from a notebook, you should add the following section to your JupyterHub `values.yaml`:

```
singleuser:
   serviceAccountName: my-serviceaccount
   networkPolicy:
     enabled: True
     ingress: []
     egress:
       - ports:
           - port: 443
             protocol: TCP
         to:
           - namespaceSelector:
              matchLabels:
                name: kube-system
     egressAllowRules:
       cloudMetadataServer: false
       dnsPortsCloudMetadataServer: true
       dnsPortsKubeSystemNamespace: true
       dnsPortsPrivateIPs: true
       nonPrivateIPs: true
       privateIPs: true
```

Then apply your updated configuration:

```
$ helm upgrade --cleanup-on-fail --install <your-jupyterhub-name> jupyterhub/jupyterhub -
↪-namespace <your-namespace> -f values.yaml
```

And restart your notebook pod:

```
$ kubectl delete po -n <your-namespace> jupyter-<your-user-name>
```

Creating Dask clusters is now possible directly from your notebook. See KubeCluster for more details.

Accessing Dask Scheduler Dashboard from JupyterLab

In order for your notebook pod to reach its own Dask Scheduler's dashboard, you should add the following files to your JupyterHub configuration:

```
singleuser:
  extraFiles:
    # Allow proxy requests to the Dask Scheduler's dashboard
    jupyter-server-config:
      mountPath: /etc/jupyter/jupyter_server_config.py
      stringData: |
        # WARNING: This setting is inherently insecure and should be used with care.
        # Ideally, restrict this to only allow your Dask Scheduler's domain instead of␣
↪allowing everything.
        c.ServerProxy.host_allowlist = lambda app, host: True

    # Configure the default path to reach Dask Scheduler's dashboard
    dask-dashboard-config:
      mountPath: /home/jovyan/.config/dask/distributed.yaml
      stringData: |
        distributed:
          dashboard:
            link: "/user/{JUPYTERHUB_USER}/proxy/{host}:{port}/status"
```

Then, apply your updated configuration and restart your notebook pod as previously described.

Warning

Setting `c.ServerProxy.host_allowlist = lambda app, host: True` is inherently insecure and may enable unintended access to other services within your network. Ideally, you should restrict the host to your Dask Scheduler's namespace or cluster name and namespace (for instance, "<dask-cluster-name>.<namespace>"). However, depending on your configuration, matching by suffix may be sufficient. For example:

```
# Authorize network flows to the Kubernetes namespace in which your Dask cluster will␣
↪be deployed.
c.ServerProxy.host_allowlist = lambda app, host: host.endswith('.your-dask-operator-
↪namespace')
```

Kubeflow

In order to use the Dask Operator with Kubeflow you need to perform some extra installation steps.

User permissions

Kubeflow doesn't know anything about our Dask custom resource definitions so we need to update the `kubeflow-kubernetes-edit` cluster role. This role allows users with cluster edit permissions to create pods, jobs and other resources and we need to add the Dask custom resources to that list. Edit the existing `clusterrole` and add a new rule to the `rules` section for `kubernetes.dask.org` that allows all operations on all custom resources in our API namespace.

```
$ kubectl patch clusterrole kubeflow-kubernetes-edit --type="json" --patch '[{"op": "add
↪", "path": "/rules/-", "value": {"apiGroups": ["kubernetes.dask.org"],"resources": ["*
↪"],"verbs": ["*"]}}]'
clusterrole.rbac.authorization.k8s.io/kubeflow-kubernetes-edit patched
```

Dashboard access

If you are using the Jupyter Notebook service in KubeFlow there are a couple of extra steps you need to do to be able to access the Dask dashboard. The dashboard will be running on the scheduler pod and accessible via the scheduler service, so to access that your Jupyter container will need to have the jupyter-server-proxy extension installed. If you are using the Dask Jupter Lab extension this will be installed automatically for you.

By default the proxy will only allow proxying other services running on the same host as the Jupyter server, which means you can't access the scheduler running in another pod. So you need to set some extra config to tell the proxy which hosts to allow. Given that we can already execute arbitrary code in Jupyter (and therefore interact with other services within the Kubernetes cluster) we can allow all hosts in the proxy settings with `c.ServerProxy.host_allowlist = lambda app, host: True`.

The *dask_kubernetes.operator.KubeCluster* and `distributed.Client` objects both have a `dashboard_link` attribute that you can view to find the URL of the dashboard, and this is also used in the widgets shown in Jupyter. The default link will not work on KubeFlow so you need to change this to `"{NB_PREFIX}/proxy/{host}:{port}/status"` to ensure it uses the Jupyter proxy.

To apply these configuration options to the Jupyter pod you can create a `PodDefault` configuration object that can be selected when launching the notebook. Create a new file with the following contents.

```yaml
# configure-dask-dashboard.yaml
apiVersion: "kubeflow.org/v1alpha1"
kind: PodDefault
metadata:
name: configure-dask-dashboard
spec:
selector:
   matchLabels:
      configure-dask-dashboard: "true"
desc: "configure dask dashboard"
env:
   - name: DASK_DISTRIBUTED__DASHBOARD__LINK
      value: "{NB_PREFIX}/proxy/{host}:{port}/status"
volumeMounts:
   - name: jupyter-server-proxy-config
   mountPath: /root/.jupyter/jupyter_server_config.py
   subPath: jupyter_server_config.py
volumes:
   - name: jupyter-server-proxy-config
   configMap:
      name: jupyter-server-proxy-config
---
apiVersion: v1
kind: ConfigMap
metadata:
name: jupyter-server-proxy-config
data:
jupyter_server_config.py: |
```

(continues on next page)

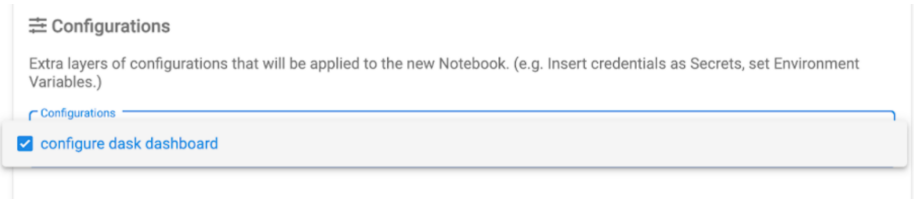
(continued from previous page)

```
c.ServerProxy.host_allowlist = lambda app, host: True
```

Then apply this to your KubeFlow user's namespace with `kubectl`. For example with the default `user@example.com` user it would be.

```
$ kubectl apply -n kubeflow-user-example-com -f configure-dask-dashboard.yaml
```

Then when you launch your Jupyter Notebook server be sure to check the `configure dask dashboard` configuration option.



Supported Versions

Python

All Dask projects generally follow the NEP 29 deprecation policy for Python where each Python minor version is support ed for 42 months. Due to Python's 12 month release cycle this ensures at least the current version and two previous versions are supported.

The Dask Kubernetes CI tests all PRs against all supported Python versions.

Kubernetes

For Kubernetes we follow the yearly support KEP. Due to the 4-6 month release cycle this also ensures that at least the current and two previous versions are supported.

The Dask Kubernetes CI tests all PRs against all supported Kubernetes versions.

> **Note**
>
> To keep the CI matrix smaller we test all Kubernetes versions against the latest Python, and all Python versions against the latest Kubernetes. We do not test older versions of Python and Kubernetes together. See dask/dask-kubernetes#559 for more information.

3.4.2 KubeCluster

Cluster manager

The operator has a cluster manager called *`dask_kubernetes.operator.KubeCluster`* that you can use to conveniently create and manage a Dask cluster in Python. Then connect a Dask `distributed.Client` object to it directly and perform your work.

The goal of the cluster manager is to abstract away the complexity of the Kubernetes resources and provide a clean and simple Python API to manager clusters while still getting all the benefits of the operator.

Under the hood the Python cluster manager will interact with ther Kubernetes API to create *custom resources* for us.

To create a cluster in the default namespace, run the following

```
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster(name='foo')
```

You can change the default configuration of the cluster by passing additional args to the python class (`namespace`, `n_workers`, etc.) of your cluster. See the API reference *API*

You can scale the cluster

```
# Scale up the cluster
cluster.scale(5)

# Scale down the cluster
cluster.scale(1)
```

You can autoscale the cluster

```
# Allow cluster to autoscale between 1 and 10 workers
cluster.adapt(minimum=1, maximum=10)

# Disable autoscaling by explicitly scaling to your desired number of workers
cluster.scale(1)
```

You can connect to the client

```
from dask.distributed import Client

# Connect Dask to the cluster
client = Client(cluster)
```

Finally delete the cluster by running

```
cluster.close()
```

Additional worker groups

Additional worker groups can also be created via the cluster manager in Python.

```
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster(name='foo')

cluster.add_worker_group(name="highmem", n_workers=2, resources={"requests": {"memory":
↪"2Gi"}, "limits": {"memory": "64Gi"}})
```

We can also scale the worker groups by name from the cluster object.

```
cluster.scale(5, worker_group="highmem")
```

Additional worker groups can also be deleted in Python.

```
cluster.delete_worker_group(name="highmem")
```

Any additional worker groups you create will be deleted when the cluster is deleted.

Customising your cluster

The `KubeCluster` class can take a selection of keyword arguments to make it quick and easy to get started, however the underlying *DaskCluster* resource can be much more complex and configured in many ways. Rather than exposing every possibility via keyword arguments instead you can pass a valid `DaskCluster` resource spec which will be used when creating the cluster. You can also generate a spec with *make_cluster_spec()* which `KubeCluster` uses internally and then modify it with your custom options.

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

config = {
   "name": "foo",
   "n_workers": 2,
   "resources":{"requests": {"memory": "2Gi"}, "limits": {"memory": "64Gi"}}
}

cluster = KubeCluster(**config)
# is equivalent to
cluster = KubeCluster(custom_cluster_spec=make_cluster_spec(**config))
```

You can also modify the spec before passing it to `KubeCluster`, for example if you want to set `nodeSelector` on your worker pods you could do it like this:

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

spec = make_cluster_spec(name="selector-example", n_workers=2)
spec["spec"]["worker"]["spec"]["nodeSelector"] = {"disktype": "ssd"}

cluster = KubeCluster(custom_cluster_spec=spec)
```

You could also have the scheduler run a Jupyter server. With this configuration you can access a Jupyter server via the Dask dashboard.

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

spec = make_cluster_spec(name="jupyter-example", n_workers=2, env={"EXTRA_PIP_PACKAGES":
↪"jupyterlab"})
spec["spec"]["scheduler"]["spec"]["containers"][0]["args"].append("--jupyter")

cluster = KubeCluster(custom_cluster_spec=spec)
```

The `cluster.add_worker_group()` method also supports passing a `custom_spec` keyword argument which can be generated with *make_worker_spec()*.

```
from dask_kubernetes.operator import KubeCluster, make_worker_spec

cluster = KubeCluster(name="example")

worker_spec = make_worker_spec(cluster_name=cluster.name, n_workers=2, resources={"limits
↪": {"nvidia.com/gpu": 1}})
worker_spec["spec"]["nodeSelector"] = {"cloud.google.com/gke-nodepool": "gpu-node-pool"}

cluster.add_worker_group(custom_spec=worker_spec)
```

Private container registry

One common use case where `make_cluster_spec` comes in handy is when pulling container images from a private registry. The Kubernetes documentation suggests creating a `Secret` with your registry credentials and then set the `imagePullSecrets` option in the `Pod` spec. The `KubeCluster` class doesn't expose any way to set `imagePullSecrets` so we will need to generate a spec and update it before creating the cluster. Thankfully `make_pod_spec` makes this quick and painless.

```
$ kubectl create secret docker-registry regcred \
      --docker-server=<your-registry-server> \
      --docker-username=<your-name> \
      --docker-password=<your-pword> \
      --docker-email=<your-email>
```

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

# Generate the spec
spec = make_cluster_spec(name="custom", image="foo.com/jacobtomlinson/dask:latest")

# Set the imagePullSecrets for the scheduler and worker pods
spec["spec"]["worker"]["spec"]["imagePullSecrets"] = [{"name": "regcred"}]
spec["spec"]["scheduler"]["spec"]["imagePullSecrets"] = [{"name": "regcred"}]

# Create the cluster
cluster = KubeCluster(custom_cluster_spec=spec)
```

Role-Based Access Control (RBAC)

In order to spawn a Dask cluster from a pod that runs on the cluster, the service account creating that pod will require a set of RBAC permissions. Create a service account you will use for Dask, and then attach the following ClusterRole to that ServiceAccount via a ClusterRoleBinding:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dask-cluster-role
rules:
  # Application: watching & handling for the custom resource we declare.
  - apiGroups: [kubernetes.dask.org]
    resources: [daskclusters, daskworkergroups, daskworkergroups/scale, daskjobs,␣
↪daskautoscalers]
    verbs: [get, list, watch, patch, create, delete]

  # Application: other resources it needs to watch and get information from.
  - apiGroups:
    - ""  # indicates the core API group
    resources: [pods, pods/status]
    verbs:
    - "get"
    - "list"
    - "watch"

  - apiGroups:
```

(continues on next page)

(continued from previous page)

```
    - ""  # indicates the core API group
    resources: [services]
    verbs:
    - "get"
    - "list"
    - "watch"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dask-cluster-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: dask-cluster-role
subjects:
  - kind: ServiceAccount
    name: dask-sa  # adjust name based on the service account you created
```

API

KubeCluster(*[, name, namespace, image, ...])	Launch a Dask Cluster on Kubernetes using the Operator
KubeCluster.scale(n[, worker_group])	Scale cluster to n workers
KubeCluster.adapt([minimum, maximum])	Turn on adaptivity
KubeCluster.get_logs()	Get logs for Dask scheduler and workers.
KubeCluster.add_worker_group(name[, ...])	Create a dask worker group by name
KubeCluster.delete_worker_group(name)	Delete a dask worker group by name
KubeCluster.close([timeout])	Delete the dask cluster

class dask_kubernetes.operator.**KubeCluster**(** (Keyword-only parameters separator (PEP 3102))*, *name: str | None = None*, *namespace: str | None = None*, *image: str | None = None*, *n_workers: int | None = None*, *resources: Dict[str, str] | None = None*, *env: List[dict] | Dict[str, str] | None = None*, *worker_command: List[str] | None = None*, *port_forward_cluster_ip: bool | None = None*, *create_mode: CreateMode | None = None*, *shutdown_on_close: bool | None = None*, *idle_timeout: int | None = None*, *resource_timeout: int | None = None*, *scheduler_service_type: str | None = None*, *custom_cluster_spec: str | dict | None = None*, *scheduler_forward_port: int | None = None*, *jupyter: bool = False*, *loop: IOLoop | None = None*, *asynchronous: bool = False*, *quiet: bool = False*, ***kwargs*)

Launch a Dask Cluster on Kubernetes using the Operator

This cluster manager creates a Dask cluster by deploying the necessary kubernetes resources the Dask Operator needs to create pods. It can also connect to an existing cluster by providing the name of the cluster.

Parameters

name: str
Name given the Dask cluster. Required except when custom_cluster_spec is passed, in which

case it's ignored in favor of custom_cluster_spec["metadata"]["name"].

namespace: str (optional)
Namespace in which to launch the workers. Defaults to current namespace if available or "default"

image: str (optional)
Image to run in Scheduler and Worker Pods.

n_workers: int
Number of workers on initial launch. Use `scale` to change this number in the future

resources: Dict[str, str]
env: List[dict] | Dict[str, str]
List of environment variables to set on scheduler and workers. Can be a list of dicts using the same structure as k8s envs or a single dictionary of key/value pairs

worker_command: List[str] | str
The command to use when starting the worker. If command consists of multiple words it should be passed as a list of strings. Defaults to `"dask-worker"`.

port_forward_cluster_ip: bool (optional)
If the chart uses ClusterIP type services, forward the ports locally. If you are running it locally it should be the port you are forwarding to `<port>`.

create_mode: CreateMode (optional)
How to handle cluster creation if the cluster resource already exists. Default behavior is to create a new cluster if one with that name doesn't exist, or connect to an existing one if it does. You can also set `CreateMode.CREATE_ONLY` to raise an exception if a cluster with that name already exists. Or `CreateMode.CONNECT_ONLY` to raise an exception if a cluster with that name doesn't exist.

shutdown_on_close: bool (optional)
Whether or not to delete the cluster resource when this object is closed. Defaults to `True` when creating a cluster and `False` when connecting to an existing one.

idle_timeout: int (optional)
If set Kubernetes will delete the cluster automatically if the scheduler is idle for longer than this timeout in seconds.

resource_timeout: int (optional)
Time in seconds to wait for Kubernetes resources to enter their expected state. Example: If the `DaskCluster` resource that gets created isn't moved into a known `status.phase` by the controller then it is likely the controller isn't running or is malfunctioning and we time out and clean up with a useful error. Example 2: If the scheduler Pod enters a `CrashBackoffLoop` state for longer than this timeout we give up with a useful error. Defaults to `60` seconds.

scheduler_service_type: str (optional)
Kubernetes service type to use for the scheduler. Defaults to `ClusterIP`.

jupyter: bool (optional)
Start Jupyter on the scheduler node.

custom_cluster_spec: str | dict (optional)
Path to a YAML manifest or a dictionary representation of a `DaskCluster` resource object which will be used to create the cluster instead of generating one from the other keyword arguments.

scheduler_forward_port: int (optional)
The port to use when forwarding the scheduler dashboard. Will utilize a random port by default

quiet: bool
If enabled, suppress all printed output. Defaults to `False`.

****kwargs: dict**
Additional keyword arguments to pass to LocalCluster

Attributes

asynchronous
Are we running in the event loop?

called_from_running_loop
dashboard_link
jupyter_link
loop
name
observed
plan
requested
scheduler_address

Methods

adapt([minimum, maximum])	Turn on adaptivity
add_worker_group(name[, n_workers, image, ...])	Create a dask worker group by name
close([timeout])	Delete the dask cluster
delete_worker_group(name)	Delete a dask worker group by name
from_name(name, **kwargs)	Create an instance of this class to represent an existing cluster by name.
get_client()	Return client for the cluster
get_logs()	Get logs for Dask scheduler and workers.
scale(n[, worker_group])	Scale cluster to n workers
sync(func, *args[, asynchronous, ...])	Call *func* with *args* synchronously or asynchronously depending on the calling context
wait_for_workers(n_workers[, timeout])	Blocking call to wait for n workers before continuing

generate_rich_output
logs

See also

KubeCluster.from_name

Examples

```
>>> from dask_kubernetes.operator import KubeCluster
>>> cluster = KubeCluster(name="foo")
```

You can add another group of workers (default is 3 workers) >>> cluster.add_worker_group('additional', n=4)

You can then resize the cluster with the scale method >>> cluster.scale(10)

And optionally scale a specific worker group >>> cluster.scale(10, worker_group='additional')

You can also resize the cluster adaptively and give it a range of workers >>> cluster.adapt(20, 50)

You can pass this cluster directly to a Dask client >>> from dask.distributed import Client >>> client = Client(cluster)

You can also access cluster logs >>> cluster.get_logs()

You can also connect to an existing cluster >>> existing_cluster = KubeCluster.from_name(name="ialreadyexist")

adapt(*minimum=None*, *maximum=None*)

Turn on adaptivity

Parameters

minimum
[int] Minimum number of workers

minimum
[int] Maximum number of workers

Examples

```
>>> cluster.adapt()  # Allow scheduler to add/remove workers within k8s cluster␣
↪resource limits
>>> cluster.adapt(minimum=1, maximum=10) # Allow scheduler to add/remove␣
↪workers within 1-10 range
```

add_worker_group(*name*, *n_workers=3*, *image=None*, *resources=None*, *worker_command=None*, *env=None*, *custom_spec=None*)

Create a dask worker group by name

Parameters

name: str
Name of the worker group

n_workers: int
Number of workers on initial launch. Use `.scale(n_workers, worker_group=name)` to change this number in the future.

image: str (optional)
Image to run in Scheduler and Worker Pods. If ommitted will use the cluster default.

resources: Dict[str, str]
Resources to be passed to the underlying pods. If ommitted will use the cluster default.

env: List[dict]
List of environment variables to pass to worker pod. If ommitted will use the cluster default.

custom_spec: dict (optional)
A dictionary representation of a worker spec which will be used to create the `DaskWorkerGroup` instead of generating one from the other keyword arguments.

Examples

```
>>> cluster.add_worker_group("high-mem-workers", n_workers=5)
```

close(*timeout=3600*)

Delete the dask cluster

delete_worker_group(*name*)

Delete a dask worker group by name

Parameters

name: str
Name of the worker group

Examples

```
>>> cluster.delete_worker_group("high-mem-workers")
```

classmethod from_name(*name*, ***kwargs*)

Create an instance of this class to represent an existing cluster by name.

Will fail if a cluster with that name doesn't already exist.

Parameters

name: str
Name of the cluster to connect to

Examples

```
>>> cluster = KubeCluster.from_name(name="simple-cluster")
```

get_logs()

Get logs for Dask scheduler and workers.

Examples

```
>>> cluster.get_logs()
{'foo': ...,
'foo-default-worker-0269dbfa0cfd4a22bcd9d92ae032f4d2': ...,
'foo-default-worker-7c1ccb04cd0e498fb21babaedd00e5d4': ...,
'foo-default-worker-d65bee23bdae423b8d40c5da7a1065b6': ...}
Each log will be a string of all logs for that container. To view
it is recommeded that you print each log.
>>> print(cluster.get_logs()["testdask-scheduler-5c8ffb6b7b-sjgrg"])
...
distributed.scheduler - INFO - -----------------------------------------------
distributed.scheduler - INFO - Clear task state
distributed.scheduler - INFO -   Scheduler at:   tcp://10.244.0.222:8786
distributed.scheduler - INFO -   dashboard at:                     :8787
...
```

scale(*n*, *worker_group='default'*)

Scale cluster to n workers

Parameters

n
[int] Target number of workers

worker_group
[str] Worker group to scale

Examples

```
>>> cluster.scale(10)  # scale cluster to ten workers
>>> cluster.scale(7, worker_group="high-mem-workers") # scale worker group high-
↪mem-workers to seven workers
```

dask_kubernetes.operator.**make_cluster_spec**(*name*, *image='ghcr.io/dask/dask:latest'*, *n_workers=None*, *resources=None*, *env=None*, *worker_command='dask-worker'*, *scheduler_service_type='ClusterIP'*, *idle_timeout=0*, *jupyter=False*)

Generate a `DaskCluster` kubernetes resource.

Populate a template with some common options to generate a `DaskCluster` kubernetes resource.

Parameters

name: str
Name of the cluster

image: str (optional)
Container image to use for the scheduler and workers

n_workers: int (optional)
Number of workers in the default worker group

resources: dict (optional)
Resource limits to set on scheduler and workers

env: dict (optional)
Environment variables to set on scheduler and workers

worker_command: str (optional)
Worker command to use when starting the workers

idle_timeout: int (optional)
Timeout to cleanup idle cluster

jupyter: bool (optional)
Start Jupyter on the Dask scheduler

dask_kubernetes.operator.**make_worker_spec**(*image='ghcr.io/dask/dask:latest'*, *n_workers=3*, *resources=None*, *env=None*, *worker_command='dask-worker'*)

3.4.3 Custom Resources

The Dask Operator has a few custom resources that can be used to create various Dask components.

- *DaskCluster* creates a full Dask cluster with a scheduler and workers.
- *DaskWorkerGroup* creates homogenous groups of workers, `DaskCluster` creates one by default but you can add more if you want multiple worker types.
- *DaskJob* creates a `Pod` that will run a script to completion along with a `DaskCluster` that the script can leverage.

DaskCluster

The `DaskCluster` custom resource creates a Dask cluster by creating a scheduler `Pod`, scheduler `Service` and default *DaskWorkerGroup* which in turn creates worker `Pod` resources.

Let's create an example called `cluster.yaml` with the following configuration:

```yaml
# cluster.yaml
apiVersion: kubernetes.dask.org/v1
kind: DaskCluster
metadata:
  name: simple
spec:
  worker:
    replicas: 2
    spec:
      containers:
      - name: worker
        image: "ghcr.io/dask/dask:latest"
        imagePullPolicy: "IfNotPresent"
        args:
          - dask-worker
          - --name
          - $(DASK_WORKER_NAME)
          - --dashboard
          - --dashboard-address
          - "8788"
        ports:
          - name: http-dashboard
            containerPort: 8788
            protocol: TCP
  scheduler:
    spec:
      containers:
      - name: scheduler
        image: "ghcr.io/dask/dask:latest"
        imagePullPolicy: "IfNotPresent"
        args:
          - dask-scheduler
        ports:
          - name: tcp-comm
            containerPort: 8786
            protocol: TCP
          - name: http-dashboard
            containerPort: 8787
            protocol: TCP
        readinessProbe:
          httpGet:
            port: http-dashboard
            path: /health
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          httpGet:
```

(continues on next page)

(continued from previous page)

```
          port: http-dashboard
          path: /health
        initialDelaySeconds: 15
        periodSeconds: 20
  service:
    type: NodePort
    selector:
      dask.org/cluster-name: simple
      dask.org/component: scheduler
    ports:
    - name: tcp-comm
      protocol: TCP
      port: 8786
      targetPort: "tcp-comm"
    - name: http-dashboard
      protocol: TCP
      port: 8787
      targetPort: "http-dashboard"
```

Editing this file will change the default configuration of you Dask cluster. See the Configuration Reference *DaskAutoscaler*. Now apply `cluster.yaml`

```
$ kubectl apply -f cluster.yaml
daskcluster.kubernetes.dask.org/simple created
```

We can list our clusters:

```
$ kubectl get daskclusters
NAME             AGE
simple   47s
```

To connect to this Dask cluster we can use the service that was created for us.

```
$ kubectl get svc -l dask.org/cluster-name=simple
NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
simple                  ClusterIP   10.96.85.120   <none>        8786/TCP,8787/TCP   86s
```

We can see here that port `8786` has been exposed for the Dask communication along with `8787` for the Dashboard.

How you access these service endpoints will vary depending on your Kubernetes cluster configuration. For this quick example we could use `kubectl` to port forward the service to your local machine.

```
$ kubectl port-forward svc/simple 8786:8786
Forwarding from 127.0.0.1:8786 -> 8786
Forwarding from [::1]:8786 -> 8786
```

Then we can connect to it from a Python session.

```
>>> from dask.distributed import Client
>>> client = Client("localhost:8786")
>>> print(client)
<Client: 'tcp://10.244.0.12:8786' processes=3 threads=12, memory=23.33 GiB>
```

We can also list all of the pods created by the operator to run our cluster.

```
$ kubectl get po -l dask.org/cluster-name=simple
NAME                                                              READY  ␣
↪STATUS    RESTARTS   AGE
simple-default-worker-13f4f0d13bbc40a58cfb81eb374f26c3            1/1    ␣
↪Running   0          104s
simple-default-worker-aa79dfae83264321a79f1f0ffe91f700            1/1    ␣
↪Running   0          104s
simple-default-worker-f13c4f2103e14c2d86c1b272cd138fe6            1/1    ␣
↪Running   0          104s
simple-scheduler                                                  1/1    ␣
↪Running   0          104s
```

The workers we see here are created by our clusters default `workergroup` resource that was also created by the operator.

You can scale the `workergroup` like you would a `Deployment` or `ReplicaSet`:

```
$ kubectl scale --replicas=5 daskworkergroup simple-default
daskworkergroup.kubernetes.dask.org/simple-default
```

We can verify that new pods have been created.

```
$ kubectl get po -l dask.org/cluster-name=simple
NAME                                                              READY  ␣
↪STATUS    RESTARTS   AGE
simple-default-worker-13f4f0d13bbc40a58cfb81eb374f26c3            1/1    ␣
↪Running   0          5m26s
simple-default-worker-a52bf313590f432d9dc7395875583b52            1/1    ␣
↪Running   0          27s
simple-default-worker-aa79dfae83264321a79f1f0ffe91f700            1/1    ␣
↪Running   0          5m26s
simple-default-worker-f13c4f2103e14c2d86c1b272cd138fe6            1/1    ␣
↪Running   0          5m26s
simple-default-worker-f4223a45b49d49288195c540c32f0fc0            1/1    ␣
↪Running   0          27s
simple-scheduler                                                  1/1    ␣
↪Running   0          5m26s
```

Finally we can delete the cluster either by deleting the manifest we applied before, or directly by name:

```
$ kubectl delete -f cluster.yaml
daskcluster.kubernetes.dask.org "simple" deleted

$ kubectl delete daskcluster simple
daskcluster.kubernetes.dask.org "simple" deleted
```

DaskWorkerGroup

When we create a `DaskCluster` resource a default worker group is created for us. But we can add more by creating more manifests. This allows us to create workers of different shapes and sizes that Dask can leverage for different tasks.

Let's create an example called `highmemworkers.yaml` with the following configuration:

```
# highmemworkers.yaml
apiVersion: kubernetes.dask.org/v1
```

(continues on next page)

(continued from previous page)

```
kind: DaskWorkerGroup
metadata:
  name: simple-highmem
spec:
  cluster: simple
  worker:
    replicas: 2
    spec:
      containers:
      - name: worker
        image: "ghcr.io/dask/dask:latest"
        imagePullPolicy: "IfNotPresent"
        resources:
          requests:
            memory: "32Gi"
          limits:
            memory: "32Gi"
        args:
          - dask-worker
          - --name
          - $(DASK_WORKER_NAME)
          - --resources
          - MEMORY=32e9
          - --dashboard
          - --dashboard-address
          - "8788"
        ports:
          - name: http-dashboard
            containerPort: 8788
            protocol: TCP
```

The main thing we need to ensure is that the `cluster` option matches the name of the cluster we created earlier. This will cause the workers to join that cluster.

See the *DaskAutoscaler*. Now apply `highmemworkers.yaml`

```
$ kubectl apply -f highmemworkers.yaml
daskworkergroup.kubernetes.dask.org/simple-highmem created
```

We can list our clusters:

```
$ kubectl get daskworkergroups
NAME                AGE
simple-default      2 hours
simple-highmem      47s
```

We don't need to worry about deleting this worker group seperately, because it has joined the existing cluster Kubernetes will delete it when the `DaskCluster` resource is deleted.

Scaling works the same was as the default worker group and can be done with the `kubectl scale` command.

DaskJob

The `DaskJob` custom resource behaves similarly to other Kubernetes batch resources. It creates a `Pod` that executes a command to completion. The difference is that the `DaskJob` also creates a `DaskCluster` alongside it and injects the appropriate configuration into the job `Pod` for it to automatically connect to and leverage the Dask cluster.

Let's create an example called `job.yaml` with the following configuration:

```yaml
# job.yaml
apiVersion: kubernetes.dask.org/v1
kind: DaskJob
metadata:
  name: simple-job
  namespace: default
spec:
  job:
    spec:
      containers:
        - name: job
          image: "ghcr.io/dask/dask:latest"
          imagePullPolicy: "IfNotPresent"
          args:
            - python
            - -c
            - "from dask.distributed import Client; client = Client(); # Do some work..."

  cluster:
    spec:
      worker:
        replicas: 2
        spec:
          containers:
            - name: worker
              image: "ghcr.io/dask/dask:latest"
              imagePullPolicy: "IfNotPresent"
              args:
                - dask-worker
                - --name
                - $(DASK_WORKER_NAME)
                - --dashboard
                - --dashboard-address
                - "8788"
              ports:
                - name: http-dashboard
                  containerPort: 8788
                  protocol: TCP
              env:
                - name: WORKER_ENV
                  value: hello-world # We dont test the value, just the name
      scheduler:
        spec:
          containers:
            - name: scheduler
              image: "ghcr.io/dask/dask:latest"
```

(continues on next page)

(continued from previous page)

```
            imagePullPolicy: "IfNotPresent"
            args:
              - dask-scheduler
            ports:
              - name: tcp-comm
                containerPort: 8786
                protocol: TCP
              - name: http-dashboard
                containerPort: 8787
                protocol: TCP
            readinessProbe:
              httpGet:
                port: http-dashboard
                path: /health
              initialDelaySeconds: 5
              periodSeconds: 10
            livenessProbe:
              httpGet:
                port: http-dashboard
                path: /health
              initialDelaySeconds: 15
              periodSeconds: 20
            env:
              - name: SCHEDULER_ENV
                value: hello-world
        service:
          type: ClusterIP
          selector:
            dask.org/cluster-name: simple-job
            dask.org/component: scheduler
          ports:
            - name: tcp-comm
              protocol: TCP
              port: 8786
              targetPort: "tcp-comm"
            - name: http-dashboard
              protocol: TCP
              port: 8787
              targetPort: "http-dashboard"
```

Editing this file will change the default configuration of you Dask job. See the *DaskAutoscaler*. Now apply `job.yaml`

```
$ kubectl apply -f job.yaml
  daskjob.kubernetes.dask.org/simple-job created
```

Now if we check our cluster resources we should see our job and cluster pods being created.

```
$ kubectl get pods
NAME                                                READY   STATUS            ␣
↪RESTARTS   AGE
simple-job-scheduler                                1/1     Running           ␣
↪0          8s
```

(continues on next page)

(continued from previous page)

```
simple-job-runner                                    1/1     Running             ␣
↪0          8s
simple-job-default-worker-1f6c670fba                 1/1     Running             ␣
↪0          8s
simple-job-default-worker-791f93d9ec                 1/1     Running             ␣
↪0          8s
```

Our runner pod will be doing whatever we configured it to do. In our example you can see we just create a simple `dask.distributed.Client` object like this:

```
from dask.distributed import Client


client = Client()

# Do some work...
```

We can do this because the job pod gets some additional environment variables set at runtime which tell the `Client` how to connect to the cluster, so the user doesn't need to worry about it.

The job pod has a default restart policy of `OnFalure` so if it exits with anything other than a `0` return code it will be restarted automatically until it completes successfully. When it does return a `0` it will go into a `Completed` state and the Dask cluster will be cleaned up automatically freeing up Kubernetes cluster resources.

```
$ kubectl get pods
NAME                                                 READY   STATUS              ␣
↪RESTARTS   AGE
simple-job-runner                                    0/1     Completed           ␣
↪0          14s
simple-job-scheduler                                 1/1     Terminating         ␣
↪0          14s
simple-job-default-worker-1f6c670fba                 1/1     Terminating         ␣
↪0          14s
simple-job-default-worker-791f93d9ec                 1/1     Terminating         ␣
↪0          14s
```

When you delete the `DaskJob` resource everything is delete automatically, whether that's just the `Completed` runner pod left over after a successful run or a full Dask cluster and runner that is still running.

```
$ kubectl delete -f job.yaml
daskjob.kubernetes.dask.org "simple-job" deleted
```

DaskAutoscaler

The `DaskAutoscaler` resource allows the scheduler to scale up and down the number of workers using dask's adaptive mode.

By creating the resource the operator controller will periodically poll the scheduler and request the desired number of workers. The scheduler calculates this number by profiling the tasks it is processing and then extrapolating how many workers it would need to complete the current graph within the target duration. By default the target duration is 5 seconds but it can be adjusted via the "distributed.adaptive.target-duration" in the scheduler config.

The controller will constrain this number between the `minimum` and `maximum` values configured in the `DaskAutoscaler` resource and then update the number of replicas in the default `DaskWorkerGroup`.

```
# autoscaler.yaml
apiVersion: kubernetes.dask.org/v1
kind: DaskAutoscaler
metadata:
  name: simple
spec:
  cluster: "simple"
  minimum: 1  # we recommend always having a minimum of 1 worker so that an idle cluster␣
↪can start working on tasks immediately
  maximum: 10 # you can place a hard limit on the number of workers regardless of what␣
↪the scheduler requests
```

```
$ kubectl apply -f autoscaler.yaml
daskautoscaler.kubernetes.dask.org/simple created
```

You can end the autoscaling at any time by deleting the resource. The number of workers will remain at whatever the autoscaler last set it to.

```
$ kubectl delete -f autoscaler.yaml
daskautoscaler.kubernetes.dask.org/simple deleted
```

Note

The autoscaler will only scale the default `WorkerGroup`. If you have additional worker groups configured they will not be taken into account.

Labels and Annotations

Labels and annotations are propagated to child resources, so labels applied to a `DaskCluster` will also be present on the `Pod` and `Service` resources it creates.

- Labels/annotations on `DaskCluster` are propagated to the `DaskWorkerGroup`, scheduler `Pod` and scheduler `Service`.
- Labels/annotations on `DaskWorkerGroup` are propagated to the worker `Pod`.
- Labels/annotations on `DaskJob` are propagated to the job `Pod` and `DaskCluster`.

Some resources also have subresource metadata options for setting labels and annotations on the resources it creates.

- `DaskCluster` has `spec.worker.metadata` which is merged into the labels/annotations for the `DaskWorkerGroup`.
- `DaskCluster` has `spec.scheduler.metadata` which is merged into the labels/annotations for the scheduler `Pod` and scheduler `Service`.
- `DaskJob` has `spec.job.metadata` which is merged into the labels/annotations for the job `Pod`.

The order of label/annotation application is `top_level <= subresource <= base`. So if the `DaskCluster` has a label of `foo=bar` but the `spec.worker.metadata.labels` had a label of `foo=baz` then the worker `Pod` would have `foo=baz`.

Equally, if the reserved base label `dask.org/component` is set at either the top-level or subresource-level this will be overridden by the controller. So `setting dask.org/component=superworker` in `DaskCluster.spec.worker.metadata.labels` will have no effect and the worker `Pod` will still have the expected label of `dask.org/component=worker`.

Example

The following `DaskCluster` has top-level annotations as well as worker and scheduler subresource annotations.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskCluster
metadata:
  name: example
  annotations:
    hello: world
spec:
  worker:
    replicas: 2
    metadata:
      annotations:
        foo: bar
    spec:
      ...
  scheduler:
    metadata:
      annotations:
        fizz: buzz
    spec:
      ...
```

The resulting scheduler `Pod` metadata annotations would be.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-scheduler
  annotations:
    fizz: buzz
    hello: world
...
```

Full Configuration Reference

Full `DaskCluster` spec reference.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskCluster
metadata:
  name: example
spec:
  worker:
    replicas: 2 # number of replica workers to spawn
    spec: ... # PodSpec, standard k8s pod - https://kubernetes.io/docs/reference/
↪generated/kubernetes-api/v1.21/#podspec-v1-core
  scheduler:
    spec: ... # PodSpec, standard k8s pod - https://kubernetes.io/docs/reference/
↪generated/kubernetes-api/v1.21/#podspec-v1-core
    service: ... # ServiceSpec, standard k8s service - https://kubernetes.io/docs/
↪reference/generated/kubernetes-api/v1.21/#servicespec-v1-core
```

(continues on next page)

(continued from previous page)

```
  idleTimeout: 5  # Number of seconds to time out scheduler liveness probe if no activity
```

Full `DaskWorkerGroup` spec reference.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskWorkerGroup
metadata:
  name: example
spec:
  cluster: "name of DaskCluster to associate worker group with"
  worker:
    replicas: 2 # number of replica workers to spawn
    spec: ... # PodSpec, standard k8s pod - https://kubernetes.io/docs/reference/
↪generated/kubernetes-api/v1.21/#podspec-v1-core
```

Full `DaskJob` spec reference.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskJob
metadata:
  name: example
spec:
  job:
    spec: ... # PodSpec, standard k8s pod - https://kubernetes.io/docs/reference/
↪generated/kubernetes-api/v1.21/#podspec-v1-core
  cluster:
    spec: ... # ClusterSpec, DaskCluster resource spec
```

Full `DaskAutoscaler` spec reference.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskAutoscaler
metadata:
  name: example
spec:
  cluster: "name of DaskCluster to autoscale"
  minimum: 0  # minimum number of workers to create
  maximum: 10 # maximum number of workers to create
```

3.4.4 Extending (advanced)

You can extend the functionality of the Dask Operator controller by writing plugins. You may wish to do this if you want the operator to create other resources like Istio `VirtualSerivce`, `Gateway` and `Certificate` resources. Extra resources like this may end up being a common requirement, but given the endless possibilities of k8s cluster setups it's hard to make this configurable.

To help cluster administrators ensure the Dask Operator does exactly what they need we support extending the controller via plugins.

Controller Design Overview

The Dask Operator's controller is built using kopf which allows you to write custom handler functions in Python for any Kubernetes event. The Dask Operator has a selection of *Custom Resources* and the controller handles create/update/delete events for these resources. For example whenever a `DaskCluster` resource is created the controller sets the `status.phase` attribute to `Created`.

```
@kopf.on.create("daskcluster.kubernetes.dask.org")
async def daskcluster_create(name, namespace, logger, patch, **kwargs):
    """When DaskCluster resource is created set the status.phase.

    This allows us to track that the operator is running.
    """
    logger.info(f"DaskCluster {name} created in {namespace}.")
    patch.status["phase"] = "Created"
```

Then there is another handler that watches for `DaskCluster` resources that have been put into this `Created` phase. This handler creates the `Pod`, `Service` and `DaskWorkerGroup` subresources of the cluster and then puts it into a `Running` phase.

```
@kopf.on.field("daskcluster.kubernetes.dask.org", field="status.phase", new="Created")
async def daskcluster_create_components(spec, name, namespace, logger, patch, **kwargs):
    """When the DaskCluster status.phase goes into Pending create the cluster components."
↪"""
    async with kubernetes.client.api_client.ApiClient() as api_client:
        api = kubernetes.client.CoreV1Api(api_client)
        custom_api = kubernetes.client.CustomObjectsApi(api_client)

        # Create scheduler Pod
        data = build_scheduler_pod_spec(...)
        kopf.adopt(data)
        await api.create_namespaced_pod(namespace=namespace, body=data)

        # Create scheduler Service
        data = build_scheduler_service_spec(...)
        kopf.adopt(data)
        await api.create_namespaced_service(namespace=namespace, body=data)

        # Create DaskWorkerGroup
        data = build_worker_group_spec(...)
        kopf.adopt(data)
        await custom_api.create_namespaced_custom_object(group="kubernetes.dask.org",␣
↪version="v1", plural="daskworkergroups", namespace=namespace, body=data)

    # Set DaskCluster to Running phase
    patch.status["phase"] = "Running"
```

Then when the `DaskWorkerGroup` resource is created that triggers the worker creation event handler which creates more `Pod` resources. In turn the creation of `Pod` and `Service` resources will be triggering internal event handlers in Kubernetes which will create containers, set iptable rules, etc.

This model of writing small handlers that are triggered by events in Kubernetes allows you to create powerful tools with simple building blocks.

Writing your own handlers

To avoid users having to write their own controllers the Dask Operator controller supports loading additional handlers from other packages via `entry_points`.

Custom handlers must be packaged as a Python module and be importable.

For example let's say you have a minimal Python package with the following structure:

```
my_controller_plugin/
├── pyproject.toml
└── my_controller_plugin/
    ├── __init__.py
    └── plugin.py
```

If you wanted to write a custom handler that would be triggered when the scheduler `Service` is created then `plugin.py` would look like this:

```
import kopf

@kopf.on.create("service", labels={"dask.org/component": "scheduler"})
async def handle_scheduler_service_create(meta, new, namespace, logger, **kwargs):
    # Do something here
    # See https://kopf.readthedocs.io/en/stable/handlers for documentation on what is␣
↪possible here
```

Then you need to ensure that your `pyproject.toml` registers the plugin as a `dask_operator_plugin`.

```
[option.entry-points.dask_operator_plugin]
my_controller_plugin = "my_controller_plugin.plugin"
```

Then you can package this up and push it to your preferred Python package repository.

Installing your plugin

When the Dask Operator controller starts up it checks for any plugins registered via the `dask_operator_plugin` entry point and loads those too. This means that installing your plugin is as simple as ensuring your plugin package is installed in the controller container image.

The controller uses the `ghcr.io/dask/dask-kubernetes-operator:latest` container image by default so your custom container `Dockerfile` would look something like this:

```
FROM ghcr.io/dask/dask-kubernetes-operator:latest

RUN pip install my-controller-plugin
```

Then when you *install the controller deployment* either via the manifest or with helm you would specify your custom container image instead.

```
helm install --set image.name=my_controller_image myrelease dask/dask-kubernetes-operator
```

3.4.5 Troubleshooting

This page contains common problems and resolutions.

Why am I losing data during scale down?

When scaling down a cluster the controller will attempt to coordinate with the Dask scheduler and decide which workers to remove. If the controller cannot communicate with the scheduler it will fall back to last-in-first-out scaling and will remove the worker with the lowest uptime, even if that worker is actively processing data. This can result in loss of data and recalculation of a graph.

This commonly happens if the version of Dask on the scheduler is very different to the verison on the controller.

To mitigate this Dask has an optional HTTP API which is more decoupled than the RPC and allows for better support between versions.

See https://github.com/dask/dask-kubernetes/issues/807

3.4.6 Migrating from classic

The classic `KubeCluster` class has been replaced with a new version that is built using the Kubernetes Operator pattern.

Installing the operator

To use the new implementation of `KubeCluster` you need to *install the Dask operator custom resources and controller*.

The custom resources allow us to describe our Dask cluster components as native Kubernetes resources rather than directly creating `Pod` and `Service` resources like the classic implementation does.

Unfortunately this requires a small amount of first time setup on you Kubernetes cluster before you can start using `dask-kubernetes`. This is a key reason why the new implementation has breaking changes. The quickest way to install things is with `helm`.

```
$ helm repo add dask https://helm.dask.org
"dask" has been added to your repositories

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "dask" chart repository
Update Complete. Happy Helming!

$ helm install --create-namespace -n dask-operator --generate-name dask/dask-kubernetes-
↪operator
NAME: dask-kubernetes-operator-1666875935
NAMESPACE: dask-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Operator has been installed successfully.
```

Now that you have the controller and CRDs installed on your cluster you can start using the new *dask_kubernetes.operator.KubeCluster*.

Using the new KubeCluster

The way you create clusters with `KubeCluster` has changed so let's look at some comparisons and explore how to migrate from the classic to the new.

Simplified Python API

One of the first big changes we've made is making simple use cases simpler. The only thing you need to create a minimal cluster is to give it a name.

```
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster(name="mycluster")
```

The first step we see folks take in customising their clusters is to modify things like the container image, environment variables, resources, etc. We've made all of the most common options available as keyword arguments to make small changes easier.

```
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster(name="mycluster",
                      image='ghcr.io/dask/dask:latest',
                      n_workers=3
                      env={"FOO": "bar"},
                      resources={"requests": {"memory": "2Gi"}, "limits": {"memory":
↪"64Gi"}})
```

Advanced YAML API

We've taken care to simplify the API for new users, but we have also worked hard to ensure the new implementation provides even more flexibility for advanced users.

Users of the classic implementation of `KubeCluster` have a lot of control over what the worker pods look like because you are required to provide a full YAML `Pod` spec. Instead of creating a loose collection of `Pod` resources directly the new implementation groups everything together into a `DaskCluster` custom resource. This resource contains some cluster configuration options and nested specs for the worker pods and scheduler pod/service. This way things are infinitely configurable, just be careful not to shooot yourself in the foot.

The classic getting started page had the following pod spec example:

```
# worker-spec.yml
kind: Pod
metadata:
  labels:
    foo: bar
spec:
  restartPolicy: Never
  containers:
  - image: ghcr.io/dask/dask:latest
    imagePullPolicy: IfNotPresent
    args: [dask-worker, --nthreads, '2', --no-dashboard, --memory-limit, 6GB, --death-
↪timeout, '60']
    name: dask-worker
    env:
      - name: EXTRA_PIP_PACKAGES
        value: git+https://github.com/dask/distributed
    resources:
      limits:
        cpu: "2"
        memory: 6G
```

(continues on next page)

(continued from previous page)

```
      requests:
        cpu: "2"
        memory: 6G
```

In the new implementation a cluster spec with the same options would look like this:

```
# cluster-spec.yml
apiVersion: kubernetes.dask.org/v1
kind: DaskCluster
metadata:
  name: example
  labels:
    foo: bar
spec:
  worker:
    replicas: 2
    spec:
      restartPolicy: Never
      containers:
      - name: worker
        image: "ghcr.io/dask/dask:latest"
        imagePullPolicy: "IfNotPresent"
        args: [dask-worker, --nthreads, '2', --no-dashboard, --memory-limit, 6GB, --
↪death-timeout, '60', '--name', $(DASK_WORKER_NAME)]
        env:
          - name: EXTRA_PIP_PACKAGES
            value: git+https://github.com/dask/distributed
        resources:
          limits:
            cpu: "2"
            memory: 6G
          requests:
            cpu: "2"
            memory: 6G
  scheduler:
    spec:
      containers:
      - name: scheduler
        image: "ghcr.io/dask/dask:latest"
        imagePullPolicy: "IfNotPresent"
        args:
          - dask-scheduler
        ports:
          - name: tcp-comm
            containerPort: 8786
            protocol: TCP
          - name: http-dashboard
            containerPort: 8787
            protocol: TCP
        readinessProbe:
          httpGet:
            port: http-dashboard
```

(continues on next page)

(continued from previous page)

```
            path: /health
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          httpGet:
            port: http-dashboard
            path: /health
          initialDelaySeconds: 15
          periodSeconds: 20
    service:
      type: ClusterIP
      selector:
        dask.org/cluster-name: example
        dask.org/component: scheduler
      ports:
      - name: tcp-comm
        protocol: TCP
        port: 8786
        targetPort: "tcp-comm"
      - name: http-dashboard
        protocol: TCP
        port: 8787
        targetPort: "http-dashboard"
```

Note that the `spec.worker.spec` section of the new cluster spec matches the `spec` of the old pod spec. But as you can see there is a lot more configuration available in this example including first-class control over the scheduler pod and service.

One powerful difference of using our own custom resources is that *everything* about our cluster is contained in the `DaskCluster` spec and all of the cluster lifecycle logic is handled by our custom controller in Kubernetes. This means we can equally create our cluster with Python or via the `kubectl` CLI. You don't even need to have `dask-kubernetes` installed to manage your clusters if you have other Kubernetes tooling that you would like to integrate with natively.

```
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster(custom_cluster_spec="cluster-spec.yml")
```

Is the same as:

```
$ kubectl apply -f cluster-spec.yml
```

You can still connect to the cluster created via `kubectl` back in Python by name and have all of the convenience of using a cluster manager object.

```
from dask.distributed import Client
from dask_kubernetes.operator import KubeCluster

cluster = KubeCluster.from_name("example")
cluster.scale(5)
client = Client(cluster)
```

Middle ground

There is also a middle ground for users who would prefer to stay in Python and have much of the spec generated for them, but still want to be able to make complex customisations.

When creating a new `KubeCluster` with keyword arguments those arguments are passed to a call to `dask_kubernetes.operator.make_cluster_spec` which is similar to `dask_kubernetes.make_pod_spec` that you may have used in the past. This function generates a dictionary representation of your `DaskCluster` spec which you can modify and pass to `KubeCluster` yourself.

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

cluster = KubeCluster(name="foo", n_workers= 2, env={"FOO": "bar"})

# is equivalent to

spec = make_cluster_spec(name="foo", n_workers= 2, env={"FOO": "bar"})
cluster = KubeCluster(custom_cluster_spec=spec)
```

This is useful if you want the convenience of keyword arguments for common options but still have the ability to make advanced tweaks like setting `nodeSelector` options on the worker pods.

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec

spec = make_cluster_spec(name="selector-example", n_workers=2)
spec["spec"]["worker"]["spec"]["nodeSelector"] = {"disktype": "ssd"}

cluster = KubeCluster(custom_cluster_spec=spec)
```

This can also enable you to migrate smoothly over from the existing tooling if you are using `make_pod_spec` as the classic pod spec is a subset of the new cluster spec.

```
from dask_kubernetes.operator import KubeCluster, make_cluster_spec
from dask_kubernetes.classic import make_pod_spec

# generate your existing classic pod spec
pod_spec = make_pod_spec(**your_custom_options)
pod_spec[...] = ... # Your existing tweaks to the pod spec

# generate a new cluster spec and merge in the existing pod spec
cluster_spec = make_cluster_spec(name="merge-example")
cluster_spec["spec"]["worker"]["spec"] = pod_spec["spec"]

cluster = KubeCluster(custom_cluster_spec=cluster_spec)
```

Troubleshooting

Moving from the classic implementation to the new operator based implementation will require some effort on your part. Sorry about that.

Hopefully this guide has given you enough information that you are motivated and able to make the change. However if you get stuck or you would like input from a Dask maintainer please don't hesitate to reach out to us via the Dask Forum.

3.4.7 Testing

Running the test suite for `dask-kubernetes` doesn't require an existing Kubernetes cluster but does require Docker, kubectl and helm.

Start by installing dask-kubernetes in editable mode - this will ensure that pytest can import dask-kubernetes:

```
$ pip install -e .
```

You will also need to install the test dependencies:

```
$ pip install -r requirements-test.txt
```

Tests are run using pytest:

```
$ pytest
============================================= test session starts␣
↪=============================================
platform darwin -- Python 3.8.8, pytest-6.2.2, py-1.10.0, pluggy-0.13.1 --
cachedir: .pytest_cache
rootdir: /Users/jtomlinson/Projects/dask/dask-kubernetes, configfile: setup.cfg
plugins: anyio-2.2.0, asyncio-0.14.0, kind-21.1.3
collected 64 items

...
================== 56 passed, 1 skipped, 6 xfailed, 1 xpassed, 53 warnings in 404.19s␣
↪(0:06:44) ==================
```

Note

Running `pytest` compiles the Custom Resource Definitions from source using `k8s-crd-resolver`, tests against them and then uninstalls them. You may have to install them again manually.

Kind

To test `dask-kubernetes` against a real Kubernetes cluster we use the pytest-kind plugin.

Kind stands for Kubernetes in Docker and will create a full Kubernetes cluster within a single Docker container on your system. Kubernetes will then make use of the lower level containerd runtime to start additional containers, so your Kubernetes pods will not appear in your `docker ps` output.

By default we set the `--keep-cluster` flag in `setup.cfg` which means the Kubernetes container will persist between `pytest` runs to avoid creation/teardown time. Therefore you may want to manually remove the container when you are done working on `dask-kubernetes`:

```
$ docker stop pytest-kind-control-plane
$ docker rm pytest-kind-control-plane
```

When you run the tests for the first time a config file will be created called `.pytest-kind/pytest-kind/kubeconfig` which is used for authenticating with the Kubernetes cluster running in the container. If you wish to inspect the cluster yourself for debugging purposes you can set the environment variable `KUBECONFIG` to point to that file, then use `kubectl` or `helm` as normal:

```
$ export KUBECONFIG=.pytest-kind/pytest-kind/kubeconfig
$ kubectl get nodes
```

(continues on next page)

(continued from previous page)

```
NAME                          STATUS   ROLES                  AGE   VERSION
pytest-kind-control-plane   Ready    control-plane,master   10m   v1.20.2
$ helm list
NAME    NAMESPACE       REVISION        UPDATED STATUS  CHART   APP VERSION
```

Docker image

Within the test suite there is a fixture which creates a Docker image called `dask-kubernetes:dev` from this Dockerfile. This image will be imported into the kind cluster and then be used in all Dask clusters created. This is the official Dask Docker image but with the very latest trunks of `dask` and `distrubuted` installed. It is recommended that you also have the latest development install of those projects in your local development environment too.

This image may go stale over time so you might want to periodically delete it to ensure it gets recreated with the latest code changes:

```
$ docker rmi dask-kubernetes:dev
```

Linting and formatting

To accept Pull Requests to `dask-kubernetes` we require that they pass `black` formatting and `flake8` linting.

To save developer time we support using pre-commit which runs `black` and `flake8` every time you attempt to locally commit code:

```
$ pip install pre-commit
$ pre-commit install
```

Testing Operator Controller PRs

Sometimes you may want to try out a PR of changes made to the operator controller before it has been merged.

To do this you'll need to build a custom Docker image and push it to a registry that your k8s cluster can pull from.

The custom image needs to take the latest stable release of the controller and install the development branch into it. You can do this directly from GitHub repos with *pip* or you can copy your local files in and install that.

```
FROM ghcr.io/dask/dask-kubernetes-operator:<latest stable release>

RUN pip install git+https://github.com/dask/dask-kubernetes.git@refs/pull/<PR>/head
```

```
$ docker build -t <image>:<tag> .
$ docker push -t <image>:<tag> .
```

Then you can use `helm` to install the controller with your custom image.

```
$ helm install --repo https://helm.dask.org \
    --create-namespace \
    -n dask-operator \
    --generate-name \
    dask-kubernetes-operator \
    --set image.name=<image> \
    --set image.tag=<tag>
```

3.4.8 Releasing

Releases are published automatically when a tag is pushed to GitHub.

```
# Set next version number
export RELEASE=x.x.x

# Create tags
git commit --allow-empty -m "Release $RELEASE"
git tag -a $RELEASE -m "Version $RELEASE"

# Push
git push upstream --tags
```

3.4.9 History

This repository was originally inspired by a Dask+Kubernetes solution within the Jade (Jupyter and Dask Environemt) project out of the UK Met office Informatics Lab. This Dask + Kubernetes solution was primarily developed by Jacob Tomlinson of the Informatics Lab and Matt Pryor of the STFC and funded by NERC.

It was then adapted by Yuvi Panda at the UC Berkeley Institute for Data Science (BIDS) and DSEP programs while using it with JupyterHub on the Pangeo project. It was then brought under the Dask github organization where it lives today.

This repository was originally named *daskernetes* to avoid conflict with an older, Google Cloud Platform specific solution named dask-kubernetes. Eventually this package superceded that one and took on the name dask-kubernetes.





INDEX

A

adapt() (*dask_kubernetes.operator.KubeCluster method*), 23
add_worker_group() (*dask_kubernetes.operator.KubeCluster method*), 23

C

close() (*dask_kubernetes.operator.KubeCluster method*), 24

D

delete_worker_group() (*dask_kubernetes.operator.KubeCluster method*), 24

F

from_name() (*dask_kubernetes.operator.KubeCluster class method*), 24

G

get_logs() (*dask_kubernetes.operator.KubeCluster method*), 24

K

KubeCluster (*class in dask_kubernetes.operator*), 20

M

make_cluster_spec() (*in module dask_kubernetes.operator*), 25
make_worker_spec() (*in module dask_kubernetes.operator*), 25

S

scale() (*dask_kubernetes.operator.KubeCluster method*), 24