# Overview

1 **Installation**  
   1.1 Pip .................................................. 3  
   1.2 Conda .................................................. 3  

2 **Configuration**  
   2.1 Authentication ........................................... 5  
   2.2 Cluster config ........................................... 5  

3 **Amazon Web Services (AWS)**  
   3.1 Overview ................................................ 7  
   3.2 Elastic Compute Cloud (EC2) .............................. 8  
   3.3 Elastic Container Service (ECS) ............................ 12  
   3.4 Fargate .................................................. 16  

4 **DigitalOcean**  
   4.1 Overview ................................................ 21  
   4.2 Droplet .................................................. 21  

5 **Google Cloud Platform**  
   5.1 Overview ................................................ 25  
   5.2 Google Cloud VMs ....................................... 26  

6 **Microsoft Azure**  
   6.1 Overview ................................................ 31  
   6.2 AzureVM .................................................. 33  

7 **Microsoft Azure Machine Learning**  
   7.1 Overview ................................................ 39  
   7.2 AzureML .................................................. 40  

8 **Troubleshooting**  
   8.1 Unable to connect to scheduler ............................. 45  
   8.2 Invalid CPU or Memory .................................... 45  

9 **GPU clusters** .............................................. 47  

10 **Creating custom OS images with Packer**  
    10.1 Installing Packer ....................................... 49
Native Cloud integration for Dask.

This package provides classes for constructing and managing ephemeral Dask clusters on various cloud platforms. To use a cloud provider cluster manager you can import it and instantiate it. Instantiating the class will result in cloud resources being created for you.

```python
from dask_cloudprovider.aws import FargateCluster
cluster = FargateCluster()

# Cluster manager specific config kwargs
```

You can then construct a Dask client with that cluster object to use the cluster.

```python
from dask.distributed import Client
client = Client(cluster)
```

Once you are connected to the cluster you can go ahead and use Dask and all computation will take place on your cloud resource.

Once you are finished be sure to close out your cluster to shut down any cloud resources you have and end any charges.

```python
cluster.close()
```

**Warning:** Cluster managers will attempt to automatically remove hanging cloud resources on garbage collection if the cluster object is destroyed without calling `cluster.close()`, however this is not guaranteed.

To implicitly close your cluster when you are done with it you can optionally contruct the cluster manager via a context manager. However this will result in the creation and destruction of the whole cluster whenever you run this code.

```python
from dask_cloudprovider.aws import FargateCluster
from dask.distributed import Client

with FargateCluster(...) as cluster:
    with Client(cluster) as client:
        # Do some Dask things
```
CHAPTER 1

Installation

1.1 Pip

$ pip install dask-cloudprovider[all]

You can also restrict your install to just a specific cloud provider by giving their name instead of all.

$ pip install dask-cloudprovider[aws]   # or
$ pip install dask-cloudprovider[azure] # or
$ pip install dask-cloudprovider[azureml] # or
$ pip install dask-cloudprovider[digitalocean] # or
$ pip install dask-cloudprovider[gcp]

1.2 Conda

$ conda install -c conda-forge dask-cloudprovider
Each cluster manager in Dask Cloudprovider will require some configuration specific to the cloud services you wish to use. Many config options will have sensible defaults and often you can create a cluster with just your authentication credentials configured.

### 2.1 Authentication

All cluster managers assume you have already configured your credentials for the cloud you are using.

For AWS this would mean storing your access key and secret key in `~/.aws/credentials`. The AWS CLI can create this for you by running the command `aws configure`.

See each cluster manager for specific details.

**Warning:** Most cluster managers also allow passing credentials as keyword arguments, although this would result in credentials being stored in code and is not advised.

### 2.2 Cluster config

Configuration can be passed to a cluster manager via keyword arguments, YAML config or environment variables.

For example the `FargateCluster` manager for AWS ECS takes a `scheduler_mem` configuration option to set how much memory to give the scheduler in megabytes. This can be configured in the following ways.

```python
from dask_cloudprovider.aws import FargateCluster

cluster = FargateCluster(
    scheduler_mem=8192
)
```
# ~/.config/dask/cloudprovider.yaml

```yaml
cloudprovider:
  ecs:
    scheduler_mem: 8192

$ export DASK_CLOUDPROVIDER__ECS__SCHEDULER_MEM=8192
```

See each cluster manager and the Dask configuration docs for more information.
### 3.1 Overview

#### 3.1.1 Authentication

In order to create clusters on AWS you need to set your access key, secret key and region. The simplest way is to use the `aws` command line tool.

```
$ pip install awscli
$ aws configure
```

#### 3.1.2 Credentials

In order for your Dask workers to be able to connect to other AWS resources such as S3 they will need credentials. This can be done by attaching IAM roles to individual resources or by passing credentials as environment variables. See each cluster manager docstring for more information.
3.2 Elastic Compute Cloud (EC2)

```python
class dask_cloudprovider.aws.EC2Cluster (region=None, availability_zone=None, bootstrap=None, auto_shutdown=None, ami=None, instance_type=None, vpc=None, subnet_id=None, security_groups=None, filesystem_size=None, key_name=None, iam_instance_profile=None, docker_image=None, **kwargs)
```

Deploy a Dask cluster using EC2.

This creates a Dask scheduler and workers on EC2 instances.

All instances will run a single configurable Docker container which should contain a valid Python environment with Dask and any other dependencies.

All optional parameters can also be configured in a `cloudprovider.yaml` file in your Dask configuration directory or via environment variables.

For example `ami` can be set via `DASK_CLOUDPROVIDER__EC2__AMI`.

See https://docs.dask.org/en/latest/configuration.html for more info.

Parameters

- **region**: string (optional)
  The region to start you clusters. By default this will be detected from your config.

- **availability_zone**: string or List(string) (optional)
  The availability zone to start you clusters. By default AWS will select the AZ with most free capacity. If you specify more than one then scheduler and worker VMs will be randomly assigned to one of your chosen AZs.

- **bootstrap**: bool (optional)
  It is assumed that the `ami` will not have Docker installed (or the NVIDIA drivers for GPU instances). If `bootstrap` is True these dependencies will be installed on instance start. If you are using a custom AMI which already has these dependencies set this to False.

- **worker_command**: string (optional)
  The command workers should run when starting. By default this will be "dask-worker" unless `instance_type` is a GPU instance in which case `dask-cuda-worker` will be used.

- **ami**: string (optional)
  The base OS AMI to use for scheduler and workers.
  This must be a Debian flavour distribution. By default this will be the latest official Ubuntu 20.04 LTS release from canonical.
  If the AMI does not include Docker it will be installed at runtime. If the `instance_type` is a GPU instance the NVIDIA drivers and Docker GPU runtime will be installed at runtime.

- **instance_type**: string (optional)
  A valid EC2 instance type. This will determine the resources available to your workers.

  See https://aws.amazon.com/ec2/instance-types/.
  By default will use `t2.micro`.

- **vpc**: string (optional)
  The VPC ID in which to launch the instances.
  Will detect and use the default VPC if not specified.
subnet_id: string (optional) The Subnet ID in which to launch the instances.

Will use all subnets for the VPC if not specified.

security_groups: List(string) (optional) The security group ID that will be attached to the workers.

Must allow all traffic between instances in the security group and ports 8786 and 8787 between the scheduler instance and wherever you are calling EC2Cluster from.

By default a Dask security group will be created with ports 8786 and 8787 exposed to the internet.

filesystem_size: int (optional) The instance filesystem size in GB.

Defaults to 40.

key_name: str (optional) The SSH key name to assign to all instances created by the cluster manager. You can list your existing key pair names with aws ec2 describe-key-pairs --query 'KeyPairs[*].KeyName' --output text.

NOTE: You will need to ensure your security group allows access on port 22. If security_groups is not set the default group will not contain this rule and you will need to add it manually.

iam_instance_profile: dict (optional) An IAM profile to assign to VMs. This can be used for allowing access to other AWS resources such as S3. See https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html.

n_workers: int Number of workers to initialise the cluster with. Defaults to 0.

worker_module: str The Python module to run for the worker. Defaults to distributed.cli.dask_worker

worker_options: dict Params to be passed to the worker class. See distributed.worker.Worker for default worker class. If you set worker_module then refer to the docstring for the custom worker class.

scheduler_options: dict Params to be passed to the scheduler class. See distributed.scheduler.Scheduler.

docker_image: string (optional) The Docker image to run on all instances.

This image must have a valid Python environment and have dask installed in order for the dask-scheduler and dask-worker commands to be available. It is recommended the Python environment matches your local environment where EC2Cluster is being created from.

For GPU instance types the Docker image much have NVIDIA drivers and dask-cuda installed.

By default the daskdev/dask:latest image will be used.

env_vars: dict (optional) Environment variables to be passed to the worker.

silence_logs: bool Whether or not we should silence logging when setting up the cluster.

asynchronous: bool If this is intended to be used directly within an event loop with async/await

security [Security or bool, optional] Configures communication security in this cluster.

Can be a security object, or True. If True, temporary self-signed credentials will be created automatically.
Notes

Resources created

<table>
<thead>
<tr>
<th>Resource</th>
<th>Name</th>
<th>Purpose</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Instance</td>
<td>dask-scheduler-{cluster uuid}</td>
<td>Dask Scheduler</td>
<td>EC2 Pricing</td>
</tr>
<tr>
<td>EC2 Instance</td>
<td>dask-worker-{cluster uuid}-{worker uuid}</td>
<td>Dask Workers</td>
<td>EC2 Pricing</td>
</tr>
</tbody>
</table>

Credentials

In order for Dask workers to access AWS resources such as S3 they will need credentials.

The best practice way of doing this is to pass an IAM role to be used by workers. See the `iam_instance_profile` keyword for more information.

Alternatively you could read in your local credentials created with `aws configure` and pass them along as environment variables. Here is a small example to help you do that.

```python
>>> def get_aws_credentials():
...     parser = configparser.RawConfigParser()
...     parser.read(os.path.expanduser('~/.aws/config'))
...     config = parser.items('default')
...     parser.read(os.path.expanduser('~/.aws/credentials'))
...     credentials = parser.items('default')
...     all_credentials = {key.upper(): value for key, value in [*config, ...
...     with contextlib.suppress(KeyError):
...         all_credentials['AWS_REGION'] = all_credentials.pop('REGION')
...     return all_credentials

>>> cluster = EC2Cluster(env_vars=get_aws_credentials())
```

Manual cleanup

If for some reason the cluster manager is terminated without being able to perform cleanup the default behaviour of `EC2Cluster` is for the scheduler and workers to time out. This will result in the host VMs shutting down. This cluster manager also creates instances with the terminate on shutdown setting so all resources should be removed automatically.

If for some reason you chose to override those settings and disable auto cleanup you can destroy resources with the following CLI command.

```bash
export CLUSTER_ID="cluster id printed during creation"
aws ec2 describe-instances \
    --filters "Name=tag:Dask Cluster,Values=${CLUSTER_ID}" \
    --query "Reservations[*].Instances[*].[InstanceId]" \
    --output text | xargs aws ec2 terminate-instances --instance-ids
```

Enable SSH for debugging

```python
>>> from dask_cloudprovider.aws import EC2Cluster
>>> cluster = EC2Cluster(key_name="myawesomemekey", 
    # Security group which allows ports 22, 8786, 8787 and all internal traffic
    security_groups=["sg-aabbcc112233"])

# You can now SSH to an instance with ssh ubuntu@public_ip
```
>>> cluster.close()

Attributes

asynchronous
auto_shutdown
bootstrap
command
dashboard_link
docker_image
gpu_instance
observed
plan
requested
scheduler_address
scheduler_class
worker_class

Methods

adapt(*args[, minimum, maximum]) Turn on adaptivity

```python
cluster.adapt()
```

call_async(f, *args, **kwargs) Run a blocking function in a thread as a coroutine.

```python
cluster.call_async(f, *args, **kwargs)
```

get_logs([cluster, scheduler, workers]) Return logs for the cluster, scheduler and workers

```python
cluster.get_logs()
```

get_tags() Generate tags to be applied to all resources.

```python
cluster.get_tags()
```

ew_worker_spec() Return name and spec for the next worker

```python
cluster.new_worker_spec()
```

scale([n, memory, cores]) Scale cluster to n workers

```python
cluster.scale(n)
```

scale_up([n, memory, cores]) Scale cluster to n workers

```python
cluster.scale_up(n)
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>Close the cluster.</td>
</tr>
<tr>
<td>get_cloud_init</td>
<td>Get cloud initialization configuration.</td>
</tr>
<tr>
<td>logs</td>
<td>Get logs for the cluster, scheduler and workers.</td>
</tr>
<tr>
<td>render_cloud_init</td>
<td>Render cloud initialization configuration.</td>
</tr>
<tr>
<td>scale_down</td>
<td>Scale cluster to n workers.</td>
</tr>
<tr>
<td>sync</td>
<td>Sync the cluster.</td>
</tr>
</tbody>
</table>

3.2. Elastic Compute Cloud (EC2)
3.3 Elastic Container Service (ECS)

```python
class dask_cloudprovider.aws.ECSCluster(fargate_scheduler=False, fargate_workers=False, image=None, scheduler_cpu=None, scheduler_mem=None, scheduler_timeout=None, scheduler_extra_args=None, scheduler_task_kwargs=None, worker_cpu=None, worker_mem=None, worker_gpu=None, worker_extra_args=None, worker_task_kwargs=None, n_workers=None, cluster_arn=None, cluster_name_template=None, execution_role_arn=None, task_role_arn=None, task_role_policies=None, cloudwatch_logs_group=None, cloudwatch_logs_stream_prefix=None, cloudwatch_logs_default_retention=None, vpc=None, subnets=None, security_groups=None, environment=None, tags=None, find_address_timeout=None, skip_cleanup=None, aws_access_key_id=None, aws_secret_access_key=None, region_name=None, platform_version=None, fargate_use_private_ip=False, mount_points=None, volumes=None, mount_volumes_on_scheduler=False, **kwargs)
```

Deploy a Dask cluster using ECS

This creates a dask scheduler and workers on an existing ECS cluster.

All the other required resources such as roles, task definitions, tasks, etc will be created automatically like in `FargateCluster`.

**Parameters**

- **fargate_scheduler**: bool (optional) Select whether or not to use fargate for the scheduler.
  
  Defaults to False. You must provide an existing cluster.

- **fargate_workers**: bool (optional) Select whether or not to use fargate for the workers.
  
  Defaults to False. You must provide an existing cluster.

- **image**: str (optional) The docker image to use for the scheduler and worker tasks.
  
  Defaults to `daskdev/dask:latest` or `rapidsai/rapidsai:latest` if `worker_gpu` is set.

- **scheduler_cpu**: int (optional) The amount of CPU to request for the scheduler in milli-cpu (1/1024).
  
  Defaults to 1024 (one vCPU). See the troubleshooting guide for information on the valid values for this argument.

- **scheduler_mem**: int (optional) The amount of memory to request for the scheduler in MB.
  
  Defaults to 4096 (4GB). See the troubleshooting guide for information on the valid values for this argument.

- **scheduler_timeout**: str (optional) The scheduler task will exit after this amount of time if there are no clients connected.
Defaults to 5 minutes.

**scheduler_extra_args: List[str] (optional)** Any extra command line arguments to pass to dask-scheduler, e.g. `['--tls-cert', '/path/to/cert.pem']`

Defaults to `None`, no extra command line arguments.

**scheduler_task_kwargs: dict (optional)** Additional keyword arguments for the scheduler ECS task.

**worker_cpu: int (optional)** The amount of CPU to request for worker tasks in milli-cpu (1/1024).

Defaults to 4096 (four vCPUs). See the troubleshooting guide for information on the valid values for this argument.

**worker_mem: int (optional)** The amount of memory to request for worker tasks in MB.

Defaults to 16384 (16GB). See the troubleshooting guide for information on the valid values for this argument.

**worker_gpu: int (optional)** The number of GPUs to expose to the worker.

To provide GPUs to workers you need to use a GPU ready docker image that has dask-cuda installed and GPU nodes available in your ECS cluster. Fargate is not supported at this time.

Defaults to `None`, no GPUs.

**worker_extra_args: List[str] (optional)** Any extra command line arguments to pass to dask-worker, e.g. `['--tls-cert', '/path/to/cert.pem']`

Defaults to `None`, no extra command line arguments.

**worker_task_kwargs: dict (optional)** Additional keyword arguments for the workers ECS task.

**n_workers: int (optional)** Number of workers to start on cluster creation.

Defaults to `None`.

**cluster_arn: str (optional if fargate is true)** The ARN of an existing ECS cluster to use for launching tasks.

Defaults to `None` which results in a new cluster being created for you.

**cluster_name_template: str (optional)** A template to use for the cluster name if `cluster_arn` is set to `None`.

Defaults to `'dask-\{uuid\}'`

**execution_role_arn: str (optional)** The ARN of an existing IAM role to use for ECS execution.

This ARN must have `sts:AssumeRole` allowed for `ecs-tasks.amazonaws.com` and allow the following permissions:

- `ecr:GetAuthorizationToken`
- `ecr:BatchCheckLayerAvailability`
- `ecr:GetDownloadUrlForLayer`
- `ecr:GetRepositoryPolicy`
- `ecr:DescribeRepositories`
- `ecr:ListImages`
- `ecr:DescribeImages`
- `ecr:BatchGetImage`
- `logs:*`
- `ec2:AuthorizeSecurityGroupIngress`
- `ec2:Describe*`
- `elasticloadbalancing:DeregisterInstancesFromLoadBalancer`
- `elasticloadbalancing:DeregisterTargets`
- `elasticloadbalancing:Describe*`
- `elasticloadbalancing:RegisterInstancesWithLoadBalancer`
- `elasticloadbalancing:RegisterTargets`

Defaults to `None` (one will be created for you).

**task_role_arn:** `str (optional)` The ARN for an existing IAM role for tasks to assume. This defines which AWS resources the dask workers can access directly. Useful if you need to read from S3 or a database without passing credentials around.

Defaults to `None` (one will be created with S3 read permission only).

**task_role_policies:** `List[str] (optional)` If you do not specify a `task_role_arn` you may want to list some IAM Policy ARNs to be attached to the role that will be created for you.

E.g if you need your workers to read from S3 you could add 
arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess.

Default `None` (no policies will be attached to the role)

**cloudwatch_logs_group:** `str (optional)` The name of an existing cloudwatch log group to place logs into.

Default `None` (one will be created called dask-ecs)

**cloudwatch_logs_stream_prefix:** `str (optional)` Prefix for log streams.

Defaults to the cluster name.

**cloudwatch_logs_default_retention:** `int (optional)` Retention for logs in days. For use when log group is auto created.

Defaults to `30`.

**vpc:** `str (optional)` The ID of the VPC you wish to launch your cluster in.

Defaults to `None` (your default VPC will be used).

**subnets:** `List[str] (optional)` A list of subnets to use when running your task.

Defaults to `None` (all subnets available in your VPC will be used).

**security_groups:** `List[str] (optional)` A list of security group IDs to use when launching tasks.

Defaults to `None` (one will be created which allows all traffic between tasks and access to ports `8786` and `8787` from anywhere).

**environment:** `dict (optional)` Extra environment variables to pass to the scheduler and worker tasks.
Useful for setting `EXTRA_APT_PACKAGES`, `EXTRA_CONDA_PACKAGES` and `
` `EXTRA_PIP_PACKAGES` if you’re using the default image.

Defaults to None.

**tags: dict (optional)** Tags to apply to all resources created automatically.

Defaults to None. Tags will always include `{"createdBy": "dask-cloudprovider"}`

**find_address_timeout: int** Configurable timeout in seconds for finding the task IP from the
cloudwatch logs.

Defaults to 60 seconds.

**skip_cleanup: bool (optional)** Skip cleaning up of stale resources. Useful if you have lots of
resources and this operation takes a while.

Default False.

**platform_version: str (optional)** Version of the AWS Fargate platform to use, e.g. “1.4.0” or
“LATEST”. This setting has no effect for the EC2 launch type.

Defaults to None.

**fargate_use_private_ip: bool (optional)** Whether to use a private IP (if True) or public IP (if
False) with Fargate.

Default False.

**mount_points: list (optional)** List of mount points as documented here: https://docs.aws.
amazon.com/AmazonECS/latest/developerguide/efs-volumes.html

Default None.

**volumes: list (optional)** List of volumes as documented here: https://docs.aws.amazon.com/
AmazonECS/latest/developerguide/efs-volumes.html

Default None.

**mount_volumes_on_scheduler: bool (optional)** Whether to also mount volumes in the sched-
uler task. Any volumes and mount points specified will always be mounted in worker tasks.
This setting controls whether volumes are also mounted in the scheduler task.

Default False.

**kwargs: dict** Additional keyword arguments to pass to SpecCluster.

**Examples**

```python
>>> from dask_cloudprovider.aws import ECSCluster
>>> cluster = ECSCluster(cluster_arn="arn:aws:ecs:<region>:<acctid>:cluster/
˓<clustername>")
```

There is also support in ECSCluster for GPU aware Dask clusters. To do this you need to create an ECS
cluster with GPU capable instances (from the g3, p3 or p3dn families) and specify the number of GPUs each
worker task should have.

```python
>>> from dask_cloudprovider.aws import ECSCluster
>>> cluster = ECSCluster(
...
...
worker_gpu=1)
```
By setting the `worker_gpu` option to something other than `None` will cause the cluster to run `dask-cuda-worker` as the worker startup command. Setting this option will also change the default Docker image to `rapidsai/rapidsai:latest`, if you’re using a custom image you must ensure the NVIDIA CUDA toolkit is installed with a version that matches the host machine along with `dask-cuda`.

**Attributes**

- `asynchronous`
- `dashboard_link`
- `observed`
- `plan`
- `requested`
- `scheduler_address`
- `tags`

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adapt</code>(*args[, minimum, maximum])`</td>
<td>Turn on adaptivity</td>
</tr>
<tr>
<td><code>get_logs</code>([cluster, scheduler, workers])`</td>
<td>Return logs for the cluster, scheduler and workers</td>
</tr>
<tr>
<td><code>new_worker_spec</code>()</td>
<td>Return name and spec for the next worker</td>
</tr>
<tr>
<td><code>scale</code>(n, memory, cores)</td>
<td>Scale cluster to n workers</td>
</tr>
<tr>
<td><code>scale_up</code>(n, memory, cores)</td>
<td>Scale cluster to n workers</td>
</tr>
</tbody>
</table>

**3.4 Fargate**

**class** `dask_cloudprovider.aws.FargateCluster(**kwargs)`

Deploy a Dask cluster using Fargate on ECS

This creates a dask scheduler and workers on a Fargate powered ECS cluster. If you do not configure a cluster one will be created for you with sensible defaults.

**Parameters**

- `**kwargs`: dict Keyword arguments to be passed to `ECSCluster`.

**Notes**

**IAM Permissions**

To create a `FargateCluster` the cluster manager will need to various AWS resources ranging from IAM roles to VPCs to ECS tasks. Depending on your use case you may want the cluster to create all of these for you, or you may wish to specify them yourself ahead of time.

Here is the full minimal IAM policy that you need to create the whole cluster:
If you specify all of the resources yourself you will need a minimal policy of:

```json
{
  "Statement": [
    {
      "Action": [
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:CreateSecurityGroup",
        "ec2:CreateTags",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:DeleteSecurityGroup",
        "ecs:CreateCluster",
        "ecs:DescribeTasks",
        "ecs:ListAccountSettings",
        "ecs:RegisterTaskDefinition",
        "ecs:RunTask",
        "ecs:StopTask",
        "ecs:ListClusters",
        "ecs:DescribeClusters",
        "ecs:DeleteCluster",
        "ecs:ListTaskDefinitions",
        "ecs:DescribeTaskDefinition",
        "ecs:DeregisterTaskDefinition",
        "iam:AttachRolePolicy",
        "iam:CreateRole",
        "iam:TagRole",
        "iam:PassRole",
        "iam:DeleteRole",
        "iam:ListRoleTags",
        "iam:ListAttachedRolePolicies",
        "iam:DetachRolePolicy",
        "logs:DescribeLogGroups"
      ],
      "Effect": "Allow",
      "Resource": [
        "*
      ]
    },
    {
      "Action": [
        "ec2:CreateTags",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ecs:DescribeTasks",
        "ecs:ListAccountSettings",
        "ecs:RegisterTaskDefinition",
        "ecs:RunTask",
        "ecs:StopTask",
      ]
    }
  ],
  "Version": "2012-10-17"
}
```
Examples

The `FargateCluster` will create a new Fargate ECS cluster by default along with all the IAM roles, security groups, and so on that it needs to function.

```python
>>> from dask_cloudprovider.aws import FargateCluster
>>> cluster = FargateCluster()
```

Note that in many cases you will want to specify a custom Docker image to `FargateCluster` so that Dask has the packages it needs to execute your workflow.

```python
>>> from dask_cloudprovider.aws import FargateCluster
>>> cluster = FargateCluster(image="<hub-user>/<repo-name>[:<tag>]"
```

One strategy to ensure that package versions match between your custom environment and the Docker container is to create your environment from an `environment.yml` file, export the exact package list for that environment using `conda list --export > package-list.txt`, and then use the pinned package versions contained in `package-list.txt` in your Dockerfile. You could use the default Dask Dockerfile as a template and simply add your pinned additional packages.

Attributes

- asynchronous
- dashboard_link
- observed
- plan
- requested
- scheduler_address
- tags

Methods
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adapt(*args[, minimum, maximum])</td>
<td>Turn on adaptivity</td>
</tr>
<tr>
<td>get_logs([cluster, scheduler, workers])</td>
<td>Return logs for the cluster, scheduler and workers</td>
</tr>
<tr>
<td>new_worker_spec()</td>
<td>Return name and spec for the next worker</td>
</tr>
<tr>
<td>scale([n, memory, cores])</td>
<td>Scale cluster to n workers</td>
</tr>
<tr>
<td>scale_up([n, memory, cores])</td>
<td>Scale cluster to n workers</td>
</tr>
</tbody>
</table>

### 3.4. Fargate

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
</tr>
<tr>
<td>logs</td>
</tr>
<tr>
<td>scale_down</td>
</tr>
<tr>
<td>sync</td>
</tr>
</tbody>
</table>
4.1 Overview

4.1.1 Authentication

To authenticate with DigitalOcean you must first generate a personal access token.

Then you must put this in your Dask configuration at `cloudprovider.digitalocean.token`. This can be done by adding the token to your YAML configuration or exporting an environment variable.

```yaml
# ~/.config/dask/cloudprovider.yaml
cloudprovider:
digitalocean:
  token: "yourtoken"
```

```bash
$ export DASK_CLOUDPROVIDER__DIGITALOCEAN__TOKEN="yourtoken"
```

4.2 Droplet

```python
class dask_cloudprovider.digitalocean.DropletCluster(region: str = None, size: str = None, image: str = None, **kwargs)
```

Cluster running on Digital Ocean droplets.

VMs in DigitalOcean (DO) are referred to as droplets. This cluster manager constructs a Dask cluster running on VMs.
When configuring your cluster you may find it useful to install the doctl tool for querying the DO API for available options.

https://www.digitalocean.com/docs/apis-clis/doctl/how-to/install/

**Parameters**

- **region**: str The DO region to launch you cluster in. A full list can be obtained with doctl compute region list.

- **size**: str The VM size slug. You can get a full list with doctl compute size list. The default is s-1vcpu-1gb which is 1GB RAM and 1 vCPU

- **image**: str The image ID to use for the host OS. This should be a Ubuntu variant. You can list available images with doctl compute image list --public | grep ubuntu.*x64.

- **worker_module**: str The Dask worker module to start on worker VMs.

- **n_workers**: int Number of workers to initialise the cluster with. Defaults to 0.

- **worker_module**: str The Python module to run for the worker. Defaults to distributed.cli.dask_worker

- **worker_options**: dict Params to be passed to the worker class. See distributed.worker.Worker for default worker class. If you set worker_module then refer to the docstring for the custom worker class.

- **scheduler_options**: dict Params to be passed to the scheduler class. See distributed.scheduler.Scheduler.

- **docker_image**: string (optional) The Docker image to run on all instances.

  This image must have a valid Python environment and have dask installed in order for the dask-scheduler and dask-worker commands to be available. It is recommended the Python environment matches your local environment where EC2Cluster is being created from.

  For GPU instance types the Docker image much have NVIDIA drivers and dask-cuda installed.

  By default the daskdev/dask:latest image will be used.

- **env_vars**: dict (optional) Environment variables to be passed to the worker.

- **silence_logs**: bool Whether or not we should silence logging when setting up the cluster.

- **asynchronous**: bool If this is intended to be used directly within an event loop with async/await

- **security**: [Security or bool, optional] Configures communication security in this cluster. Can be a security object, or True. If True, temporary self-signed credentials will be created automatically.

**Examples**

Create the cluster.

```python
>>> from dask_cloudprovider.digitalocean import DropletCluster
>>> cluster = DropletCluster(n_workers=1)
Creating scheduler instance
Created droplet dask-38b817c1-scheduler
Waiting for scheduler to run
```
Scheduler is running
Creating worker instance
Created droplet dask-38b817c1-worker-dc95260d

Connect a client.

```python
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

Do some work.

```python
>>> import dask.array as da

>>> arr = da.random.random((1000, 1000), chunks=(100, 100))
>>> arr.mean().compute()
0.5001550986751964
```

Close the cluster

```python
>>> client.close()
>>> cluster.close()
```

You can also do this all in one go with context managers to ensure the cluster is created and cleaned up.

```python
>>> with DropletCluster(n_workers=1) as cluster:
...     with Client(cluster) as client:
...         print(da.random.random((1000, 1000), chunks=(100, 100)).mean().compute())
```
scheduler_address
scheduler_class
worker_class

Methods

- **adapt(*args[, minimum, maximum])**  
  Turn on adaptivity
- **call_async(f, *args, **kwargs)**  
  Run a blocking function in a thread as a coroutine.
- **get_logs([cluster, scheduler, workers])**  
  Return logs for the cluster, scheduler and workers
- **get_tags()**  
  Generate tags to be applied to all resources.
- **new_worker_spec()**  
  Return name and spec for the next worker
- **scale([n, memory, cores])**  
  Scale cluster to n workers
- **scale_up([n, memory, cores])**  
  Scale cluster to n workers

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>close</code></td>
</tr>
<tr>
<td><code>get_cloud_init</code></td>
</tr>
<tr>
<td><code>logs</code></td>
</tr>
<tr>
<td><code>render_cloud_init</code></td>
</tr>
<tr>
<td><code>scale_down</code></td>
</tr>
<tr>
<td><code>sync</code></td>
</tr>
</tbody>
</table>
5.1 Overview

5.1.1 Authentication

In order to create clusters on GCP you need to set your authentication credentials. You can do this via the gcloud command line tool.

```bash
$ gcloud auth login
```

Alternatively you can use a service account which provides credentials in a JSON file. You must set the GOOGLE_APPLICATION_CREDENTIALS environment variable to the path to the JSON file.

```bash
$ export GOOGLE_APPLICATION_CREDENTIALS=/path/to/credentials.json
```

5.1.2 Project ID

To use Dask Cloudprovider with GCP you must also configure your Project ID. Generally when creating a GCP account you will create a default project. This can be found at the top of the GCP dashboard.

Your Project ID must be added to your Dask config file.

```yaml
# ~/.config/dask/cloudprovider.yaml
cloudprovider:
  gcp:
    projectid: "YOUR PROJECT ID"
```

Or via an environment variable.

```
$ export DASK_GCP_PROJECT_ID="YOUR PROJECT ID"
```
5.2 Google Cloud VMs

Cluster running on GCP VM Instances.

This cluster manager constructs a Dask cluster running on Google Cloud Platform VMs.

When configuring your cluster you may find it useful to install the `gcloud` tool for querying the GCP API for available options.

https://cloud.google.com/sdk/gcloud

**Parameters**

**projectid:** `str` Your GCP project ID. This must be set either here or in your Dask config.

https://cloud.google.com/resource-manager/docs/creating-managing-projects

See the GCP docs page for more info.

https://cloudprovider.dask.org/en/latest/gcp.html#project-id

**zone:** `str` The GCP zone to launch your cluster in. A full list can be obtained with `gcloud compute zones list`.

**network:** `str` The GCP VPC network/subnetwork to use. The default is `default`. If using firewall rules, please ensure the following accesses are configured:

- egress 0.0.0.0/0 on all ports for downloading docker images and general data access
- ingress 10.0.0.0/8 on all ports for internal communication of workers
- ingress 0.0.0.0/0 on 8786-8787 for external accessibility of the dashboard/scheduler
- (optional) ingress 0.0.0.0/0 on 22 for ssh access

**machine_type:** `str` The VM machine_type. You can get a full list with `gcloud compute machine-types list`. The default is `n1-standard-1` which is 3.75GB RAM and 1 vCPU

**source_image:** `str` The OS image to use for the VM. Dask Cloudprovider will bootstrap Ubuntu based images automatically. Other images require Docker and for GPUs the NVIDIA Drivers and NVIDIA Docker.

A list of available images can be found with `gcloud compute images list`

**Valid values are:**

- The short image name provided it is in `projectid`.
- The full image name `projects/<projectid>/global/images/<source_image>`.
- The full image URI such as those listed in `gcloud compute images list --uri`.

```python
class dask_cloudprovider.gcp.GCPCluster(projectid=None, zone=None, network=None, machine_type=None, source_image=None, docker_image=None, ngpus=None, gpu_type=None,filesystem_size=None, auto_shutdown=None, bootstrap=True, **kwargs)
```
The default is `projects/ubuntu-os-cloud/global/images/ubuntu-minimal-1804-bionic-v20201014`.

docker_image: string (optional) The Docker image to run on all instances.

This image must have a valid Python environment and have *dask* installed in order for the *dask-scheduler* and *dask-worker* commands to be available. It is recommended the Python environment matches your local environment where *EC2Cluster* is being created from.

For GPU instance types the Docker image much have NVIDIA drivers and *dask-cuda* installed.

By default the *daskdev/dask:latest* image will be used.

ngpus: int (optional) The number of GPUs to atatch to the instance. Default is 0.

gpu_type: str (optional) The name of the GPU to use. This must be set if ngpus>0. You can see a list of GPUs available in each zone with `gcloud compute accelerator-types list`.

filesystem_size: int (optional) The VM filesystem size in GB. Defaults to 50.

n_workers: int (optional) Number of workers to initialise the cluster with. Defaults to 0.

bootstrap: bool (optional) Install Docker and NVIDIA drivers if ngpus>0. Set to False if you are using a custom `source_image` which already has these requirements. Defaults to True.

worker_class: str The Python class to run for the worker. Defaults to *dask.distributed.Nanny*.

worker_options: dict (optional) Params to be passed to the worker class. See `distributed.worker.Worker` for default worker class. If you set `worker_class` then refer to the docstring for the custom worker class.

env_vars: dict (optional) Environment variables to be passed to the worker.

scheduler_options: dict (optional) Params to be passed to the scheduler class. See `distributed.scheduler.Scheduler`.

silence_logs: bool (optional) Whether or not we should silence logging when setting up the cluster.

asynchronous: bool (optional) If this is intended to be used directly within an event loop with async/await

security [Security or bool (optional)] Configures communication security in this cluster. Can be a security object, or True. If True, temporary self-signed credentials will be created automatically.

**Examples**

Create the cluster.

```python
>>> from dask_cloudprovider.gcp import GCPCluster
>>> cluster = GCPCluster(n_workers=1)
Launching cluster with the following configuration:
Source Image: projects/ubuntu-os-cloud/global/images/ubuntu-minimal-1804-bionic-v20201014
Docker Image: daskdev/dask:latest
```
Connect a client.

```python
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

Do some work.

```python
>>> import dask.array as da
>>> arr = da.random.random((1000, 1000), chunks=(100, 100))
>>> arr.mean().compute()
0.5001550986751964
```

Close the cluster

```python
>>> cluster.close()
Closing Instance: dask-acc897b9-worker-bfbc94bc
Closing Instance: dask-acc897b9-scheduler
```

You can also do this all in one go with context managers to ensure the cluster is created and cleaned up.

```python
>>> with GCPCluster(n_workers=1) as cluster:
...     with Client(cluster) as client:
...         print(da.random.random((1000, 1000), chunks=(100, 100)).mean().compute())
```

(continues on next page)
Attributes

- asynchronous
- auto_shutdown
- bootstrap
- command
- dashboard_link
- docker_image
- gpu_instance
- observed
- plan
- requested
- scheduler_address
- scheduler_class
- worker_class

Methods

- adapt(*args[, minimum, maximum])
- call_async(f, *args, **kwargs)
- get_logs([cluster, scheduler, workers])
- get_tags()
- new_worker_spec()
- scale([n, memory, cores])
- scale_up([n, memory, cores])
- close
- get_cloud_init
- logs
- render_cloud_init
- scale_down
- sync

5.2. Google Cloud VMs
Microsoft Azure

AzureVMCluster(location, resource_group, ...) Cluster running on Azure Virtual machines.

6.1 Overview

6.1.1 Authentication

In order to create clusters on Azure you need to set your authentication credentials. You can do this via the az command line tool.

```
$ az login
```

Note: Setting the default output to **table** with `az configure` will make the `az` tool much easier to use.

6.1.2 Resource Groups

To create resources on Azure they must be placed in a resource group. Dask Cloudprovider will need a group to create Dask components in.

You can list existing groups via the cli.

```
$ az group list
```

You can also create a new resource group if you do not have an existing one.

```
$ az group create --location <location> --name <resource group name> --subscription <subscription>
```

You can get a full list of locations with `az account list-locations` and subscriptions with `az account list`. 

31
Take note of your resource group name for later.

### 6.1.3 Virtual Networks

Compute resources on Azure must be placed in virtual networks (vnet). Dask Cloudprovider will require an existing vnet to connect compute resources to.

You can list existing vnets via the cli.

```bash
$ az network vnet list
```

You can also create a new vnet via the cli.

```bash
$ az network vnet create -g <resource group name> -n <vnet name> --address-prefix 10.0.0.0/16
    --subnet-name <subnet name> --subnet-prefix 10.0.0.0/24
```

This command will create a new vnet in your resource group with one subnet with the 10.0.0.0/24 prefix. For more than 255 compute resources you will need additional subnets.

Take note of your vnet name for later.

### 6.1.4 Security Groups

To allow network traffic to reach your Dask cluster you will need to create a security group which allows traffic on ports 8786-8787 from wherever you are.

You can list existing security groups via the cli.

```bash
$ az network nsg list
```

Or you can create a new security group.

```bash
$ az network nsg create -g <resource group name> --name <security group name>
$ az network nsg rule create -g <resource group name> --nsg-name <security group name> --priority 500 --source-address-prefixes '*' --destination-port-ranges 8786 8787 --destination-address-prefixes '*' --access Allow --protocol Tcp --description "Allow Internet to Dask on ports 8786,8787."
```

This example allows all traffic to 8786-8787 from the internet. It is recommended you make your rules more restrictive than this by limiting it to your corporate network or specific IP.

Again take note of this security group name for later.
Cluster running on Azure Virtual machines.

This cluster manager constructs a Dask cluster running on Azure Virtual Machines.

When configuring your cluster you may find it useful to install the `az` tool for querying the Azure API for available options.

https://docs.microsoft.com/en-us/cli/azure/install-azure-cli

**Parameters**

- **location**: str The Azure location to launch you cluster in. List available locations with `az account list-locations`.
- **resource_group**: str The resource group to create components in. List your resource groups with `az group list`.
- **vnet**: str The vnet to attach VM network interfaces to. List your vnets with `az network vnet list`.
- **security_group**: str The security group to apply to your VMs. This must allow ports 8786-8787 from wherever you are running this from. List your security groups with `az network nsg list`.
- **public_ingress**: bool Assign a public IP address to the scheduler. Default True.
- **vm_size**: str Azure VM size to use for scheduler and workers. Default `Standard_DS1_v2`. List available VM sizes with `az vm list-sizes --location <location>`.
- **scheduler_vm_size**: str Azure VM size to use for scheduler. If not set will use the `vm_size`.
- **vm_image**: dict By default all VMs will use the latest Ubuntu LTS release with the following configuration

  ```json
  {"publisher": "Canonical", "offer": "UbuntuServer","sku": "18.04-LTS", "version": "latest"}
  ```

  You can override any of these options by passing a dict with matching keys here. For example if you wish to try Ubuntu 19.04 you can pass `{"sku": "19.04"}` and the publisher, offer and version will be used from the default.
- **bootstrap**: bool (optional) It is assumed that the VHD will not have Docker installed (or the NVIDIA drivers for GPU instances). If `bootstrap` is True these dependencies will be installed on instance start. If you are using a custom VHD which already has these dependencies set this to False.
- **auto_shutdown**: bool (optional) Shutdown the VM if the Dask process exits. Default True.
- **worker_module**: str The Dask worker module to start on worker VMs.
- **n_workers**: int Number of workers to initialise the cluster with. Defaults to 0.
- **worker_module**: str The Python module to run for the worker. Defaults to `distributed`. cli.dask_worker
worker_options: dict Params to be passed to the worker class. See distributed.
worker.Worker for default worker class. If you set worker_module then refer to
the docstring for the custom worker class.

scheduler_options: dict Params to be passed to the scheduler class. See distributed.
scheduler.Scheduler.

docker_image: string (optional) The Docker image to run on all instances.
This image must have a valid Python environment and have dask installed in order for the
dask-scheduler and dask-worker commands to be available. It is recommended the
Python environment matches your local environment where AzureVMCluster is being
created from.

For GPU instance types the Docker image much have NVIDIA drivers and dask-cuda
installed.

By default the daskdev/dask:latest image will be used.
silence_logs: bool Whether or not we should silence logging when setting up the cluster.

asynchronous: bool If this is intended to be used directly within an event loop with async/await

security [Security or bool, optional] Configures communication security in this cluster. Can
be a security object, or True. If True, temporary self-signed credentials will be created
automatically.

Examples

Minimal example
Create the cluster

```python
>>> from dask_cloudprovider.azure import AzureVMCluster
>>> cluster = AzureVMCluster(resource_group="<resource group>",
... vnet="<vnet>",
... security_group="<security group>",
... n_workers=1)
Creating scheduler instance
Assigned public IP
Network interface ready
Creating VM
Created VM dask-564cc8b-scheduler
Waiting for scheduler to run
Scheduler is running
Creating worker instance
Network interface ready
Creating VM
Created VM dask-564cc8b-worker-e1ebfc0e
```

Connect a client.

```python
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

Do some work.

```python
>>> import dask.array as da
>>> arr = da.random.random((1000, 1000), chunks=(100, 100))
```
Close the cluster.

```python
>>> client.close()
>>> cluster.close()
```

Terminated VM `dask-5648cc8b-worker-e1ebfc0e`

Removed disks for VM `dask-5648cc8b-worker-e1ebfc0e`

Deleted network interface

Terminated VM `dask-5648cc8b-scheduler`

Removed disks for VM `dask-5648cc8b-scheduler`

Deleted network interface

Unassigned public IP

You can also do this all in one go with context managers to ensure the cluster is created and cleaned up.

```python
>>> with AzureVMCluster(resource_group="<resource group>",
...                       vnet="<vnet>",
...                       security_group="<security group>",
...                       n_workers=1) as cluster:
...     with Client(cluster) as client:
...         print(da.random.random((1000, 1000), chunks=(100, 100)).mean().compute())
```

Creating scheduler instance

Assigned public IP

Network interface ready

Creating VM

Created VM `dask-1e6dac4e-scheduler`

Waiting for scheduler to run

Scheduler is running

Creating worker instance

Network interface ready

Creating VM

Created VM `dask-1e6dac4e-worker-c7c4ca23`

0.4996427609642539

Terminated VM `dask-1e6dac4e-worker-c7c4ca23`

Removed disks for VM `dask-1e6dac4e-worker-c7c4ca23`

Deleted network interface

Terminated VM `dask-1e6dac4e-scheduler`

Removed disks for VM `dask-1e6dac4e-scheduler`

Deleted network interface

Unassigned public IP

**RAPIDS example**

You can also use `AzureVMCluster` to run a GPU enabled cluster and leverage the RAPIDS accelerated libraries.

```python
>>> cluster = AzureVMCluster(resource_group="<resource group>",
...                            vnet="<vnet>",
...                            security_group="<security group>",
...                            n_workers=1,
...                            vm_size="Standard_NC12s_v3", # Or any NVIDIA GPU
...                            docker_image="rapidsai/rapidsai:cuda11.0-runtime-
...                            ubuntu18.04-py3.8",
```
Run some GPU code.

```python
>>> from dask.distributed import Client
>>> client = Client(cluster)
>>>
>>> def get_gpu_model():
...     import pynvml
...     pynvml.nvmlInit()
...     return pynvml.nvmlDeviceGetName(pynvml.nvmlDeviceGetHandleByIndex(0))

>>> client.submit(get_gpu_model).result()
'b'Tesla V100-PCIE-16GB'
```

Close the cluster.

```python
>>> client.close()
>>> cluster.close()
```

**Attributes**

- `asynchronous`
- `auto_shutdown`
- `bootstrap`
- `command`
- `dashboard_link`
- `docker_image`
- `gpu_instance`
- `observed`
- `plan`
- `requested`
- `scheduler_address`
- `scheduler_class`
- `worker_class`

**Methods**

- `adapt(*args[, minimum, maximum])`
  
  Turn on adaptivity
- `call_async(f, *args, **kwargs)`
  
  Run a blocking function in a thread as a coroutine.
- `get_logs([cluster, scheduler, workers])`
  
  Return logs for the cluster, scheduler and workers
- `get_tags()`
  
  Generate tags to be applied to all resources.
- `new_worker_spec()`
  
  Return name and spec for the next worker
- `scale(n, memory, cores)`
  
  Scale cluster to n workers
- `scale_up(n, memory, cores)`
  
  Scale cluster to n workers
<table>
<thead>
<tr>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_cloud_init</td>
</tr>
<tr>
<td>logs</td>
</tr>
<tr>
<td>render_cloud_init</td>
</tr>
<tr>
<td>scale_down</td>
</tr>
<tr>
<td>sync</td>
</tr>
</tbody>
</table>
Warning: The Azure ML integration has been deprecated and will be removed in a future release. Please use the dask_cloudprovider.azure.AzureVMCluster cluster manager instead.

```python
 AzureMLCluster(workspace[, compute_target, ...])
```

Deploy a Dask cluster using Azure ML

### 7.1 Overview

To start using dask_cloudprovider.AzureMLCluster you need, at a minimum, an Azure subscription and an AzureML Workspace.
7.2 AzureML

class dask_cloudprovider.azureml.AzureMLCluster(workspace, compute_target=None, environment_definition=None, experiment_name=None, initial_node_count=None, jupyter=None, jupyter_port=None, dashboard_port=None, scheduler_port=None, scheduler_idle_timeout=None, worker_development_idle_timeout=None, additional_ports=None, admin_username=None, admin_ssh_key=None, datastores=None, code_store=None, vnet_resource_group=None, vnet=None, subnet=None, show_output=False, telemetry_opt_out=None, asynchronous=False, **kwargs)

Deploy a Dask cluster using Azure ML.

This creates a dask scheduler and workers on an Azure ML Compute Target.

Parameters


vm_size: str (optional) Azure VM size to be used in the Compute Target - see https://aka.ms/azureml/vmsizes.


Defaults to []. To mount all datastores in the workspace, set to ws.datastores.values().


Defaults to the “AzureML-Dask-CPU” or “AzureML-Dask-GPU” curated environment.

scheduler_idle_timeout: int (optional) Number of idle seconds leading to scheduler shut down.

Defaults to 1200 (20 minutes).

experiment_name: str (optional) The name of the Azure ML Experiment used to control the cluster.

Defaults to dask-cloudprovider.

initial_node_count: int (optional) The initial number of nodes for the Dask Cluster.

Defaults to 1.

jupyter: bool (optional) Flag to start JupyterLab session on the headnode of the cluster.

Defaults to False.
**jupyter_port**: int (optional)  Port on headnode to use for hosting JupyterLab session.

Defaults to 9000.

**dashboard_port**: int (optional)  Port on headnode to use for hosting Dask dashboard.

Defaults to 9001.

**scheduler_port**: int (optional)  Port to map the scheduler port to via SSH-tunnel if machine not on the same VNET.

Defaults to 9002.

**worker_death_timeout**: int (optional)  Number of seconds to wait for a worker to respond before removing it.

Defaults to 30.

**additional_ports**: list[tuple[int, int]] (optional)  Additional ports to forward. This requires a list of tuples where the first element is the port to open on the headnode while the second element is the port to map to or forward via the SSH-tunnel.

Defaults to [].

**compute_target**: azureml.core.ComputeTarget (optional)  Azure ML Compute Target - see https://aka.ms/azureml/computetarget.

**admin_username**: str (optional)  Username of the admin account for the AzureML Compute. Required for runs that are not on the same VNET. Defaults to empty string. Throws Exception if machine not on the same VNET.

Defaults to "".

**admin_ssh_key**: str (optional)  Location of the SSH secret key used when creating the AzureML Compute. The key should be passwordless if run from a Jupyter notebook. The id_rsa file needs to have 0700 permissions set. Required for runs that are not on the same VNET. Defaults to empty string. Throws Exception if machine not on the same VNET.

Defaults to "".

**vnet**: str (optional)  Name of the virtual network.

**subnet**: str (optional)  Name of the subnet inside the virtual network vnet.

**vnet_resource_group**: str (optional)  Name of the resource group where the virtual network vnet is located. If not passed, but names for vnet and subnet are passed, vnet_resource_group is assigned with the name of resource group associated with workspace.

**telemetry_opt_out**: bool (optional)  A boolean parameter. Defaults to logging a version of AzureMLCluster with Microsoft. Set this flag to False if you do not want to share this information with Microsoft. Microsoft is not tracking anything else you do in your Dask cluster nor any other information related to your workload.

**asynchronous**: bool (optional)  Flag to run jobs asynchronously.

**kwargs**: dict  Additional keyword arguments.

---

**Examples**

First, import all necessary modules.
Next, create the `Workspace` object given your AzureML Workspace parameters. Check more in the AzureML documentation for `Workspace`.

You can use `ws = Workspace.from_config()` after downloading the config file from the Azure Portal or ML Studio.

```python
>>> subscription_id = "<your-subscription-id-here>"
>>> resource_group = "<your-resource-group>"
>>> workspace_name = "<your-workspace-name>"
```

Then create the cluster.

```python
>>> amlcluster = AzureMLCluster(
...     # required
...     ws,
...     # optional
...     vm_size="STANDARD_DS13_V2",  # Azure VM
...     datastores=ws.datastores.values(),  # Azure ML
...     environment_definition=ws.environments['AzureML-Dask-CPU'],  # Azure ML
...     jupyter=true,  # Start
...     initial_node_count=2,  # number of
...     scheduler_idle_timeout=7200  # scheduler
... )
```

Once the cluster has started, the Dask Cluster widget will print out two links:

1. Jupyter link to a Jupyter Lab instance running on the headnode.
2. Dask Dashboard link.

Note that `AzureMLCluster` uses IPython Widgets to present this information, so if you are working in Jupyter Lab and see text that starts with `VBox(children=`, make sure you have enabled the IPython Widget extension.

To connect to the Jupyter Lab session running on the cluster from your own computer, click the link provided in the widget printed above, or if you need the link directly it is stored in `amlcluster.jupyter_link`.

Once connected, you’ll be in an AzureML Run session. To connect Dask from within the session, just run the following code to connect dask to the cluster:

```python
from azureml.core import Run
from dask.distributed import Client
```
You can stop the cluster with `amlcluster.close()`. The cluster will automatically spin down if unused for 20 minutes by default. Alternatively, you can delete the Azure ML Compute Target or cancel the Run from the Python SDK or UI to stop the cluster.

**Attributes**

- `asynchronous`
- `dashboard_link` Link to Dask dashboard.
- `jupyter_link` Link to JupyterLab on running on the headnode of the cluster.
- `observed`
- `plan`
- `requested`
- `scheduler_address`

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adapt([Adaptive])</code></td>
<td>Turn on adaptivity</td>
</tr>
<tr>
<td><code>close()</code></td>
<td>Close the cluster.</td>
</tr>
<tr>
<td><code>get_logs([cluster, scheduler, workers])</code></td>
<td>Return logs for the cluster, scheduler and workers</td>
</tr>
<tr>
<td><code>scale([workers])</code></td>
<td>Scale the cluster.</td>
</tr>
<tr>
<td><code>scale_down([workers])</code></td>
<td>Scale down the number of workers.</td>
</tr>
<tr>
<td><code>scale_up([workers])</code></td>
<td>Scale up the number of workers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>close()</code></td>
<td>Close the cluster. All Azure ML Runs corresponding to the scheduler and worker processes will be completed. The Azure ML Compute Target will return to its minimum number of nodes after its idle time before scaledown.</td>
</tr>
<tr>
<td><code>dashboard_link</code></td>
<td>Link to Dask dashboard.</td>
</tr>
<tr>
<td><code>jupyter_link</code></td>
<td>Link to JupyterLab on running on the headnode of the cluster. Set <code>jupyter=True</code> when creating the <code>AzureMLCluster</code>.</td>
</tr>
<tr>
<td><code>scale(workers=1)</code></td>
<td>Scale the cluster. Scales to a maximum of the workers available in the cluster.</td>
</tr>
<tr>
<td><code>scale_down(workers=1)</code></td>
<td>Scale down the number of workers. Scales to minimum of 1.</td>
</tr>
<tr>
<td><code>scale_up(workers=1)</code></td>
<td>Scale up the number of workers.</td>
</tr>
</tbody>
</table>
This document contains frequently asked troubleshooting problems.

### 8.1 Unable to connect to scheduler

The most common issue is not being able to connect to the cluster once it has been constructed.

Each cluster manager will construct a Dask scheduler and by default expose it via a public IP address. You must be able to connect to that address on ports 8786 and 8787 from wherever your Python session is.

If you are unable to connect to this address it is likely that there is something wrong with your network configuration, for example you may have corporate policies implementing additional firewall rules on your account.

To reduce the chances of this happening it is often simplest to run Dask Cloudprovider from within the cloud you are trying to use and configure private networking only. See your specific cluster manager docs for info.

### 8.2 Invalid CPU or Memory

When working with `FargateCluster` or `ECSCluster`, CPU and memory arguments can only take values from a fixed set of combinations.

So, for example, code like this will result in an error

```python
from dask_cloudprovider import FargateCluster
cluster = FargateCluster(
    image="daskdev/dask:latest",
    worker_cpu=256,
    worker_mem=30720,
    n_workers=2,
    fargate_use_private_ip=False,
    scheduler_timeout="15 minutes"
)
```

(continues on next page)
client = Client(cluster)
cluster

# botocore.errorfactory.ClientException:
# An error occurred (ClientException) when calling the RegisterTaskDefinition_
# operation:
# No Fargate configuration exists for given values.

This is because ECS and Fargate task definitions with CPU=256 cannot have as much memory as that code is requesting.

The AWS-accepted set of combinations is documented at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-cpu-memory-error.html.
Many cloud providers have GPU offerings and so it is possible to launch GPU enabled Dask clusters with Dask Cloudprovider.

Each cluster manager handles this differently but generally you will need to configure the following settings:

- Configure the hardware to include GPUs. This may be by changing the hardware type or adding accelerators.
- Ensure the OS/Docker image has the NVIDIA drivers. For Docker images it is recommended to use the [RAPIDS images](https://hub.docker.com/r/rapidsai/rapidsai/).
- Set the `worker_module` config option to `dask_cuda.cli.dask_cuda_worker` or `worker_command` option to `dask-cuda-worker`.

In the following AWS `dask_cloudprovider.aws.EC2Cluster` example we set the `ami` to be a Deep Learning AMI with NVIDIA drivers, the `docker_image` to RAPIDS, the `instance_type` to `p3.2xlarge` which has one NVIDIA Tesla V100 and the `worker_module` to `dask_cuda.cli.dask_cuda_worker`.

```python
>>> cluster = EC2Cluster(ami="ami-0c7c7d78f752f8f17", # Example Deep Learning AMI
                       # (Ubuntu 18.04)
                       docker_image="rapidsai/rapidsai:cuda10.1-runtime-ubuntu18.04",
                       instance_type="p3.2xlarge",
                       worker_module="dask_cuda.cli.dask_cuda_worker",
                       bootstrap=False,
                       filesystem_size=120)
```

See each cluster manager’s example sections for info on starting a GPU cluster.
CHAPTER 10

Creating custom OS images with Packer

Many cloud providers in Dask Cloudprovider involve creating VMs and installing dependencies on those VMs at boot time.

This can slow down the creation and scaling of clusters, so this page discusses building custom images using Packer to speed up cluster creation.

Packer is a utility which boots up a VM on your desired cloud, runs any installation steps and then takes a snapshot of the VM for use as a template for creating new VMs later. This allows us to run through the installation steps once, and then reuse them when starting Dask components.

10.1 Installing Packer

See the official install docs.

10.2 Packer Overview

To create an image with packer we need to create a JSON config file.

A Packer config file is broken into a couple of sections, builders and provisioners.

A builder configures what type of image you are building (AWS AMI, GCP VMI, etc). It describes the base image you are building on top of and connection information for Packer to connect to the build instance.

When you run packer build /path/to/config.json a VM (or multiple VMs if you configure more than one) will be created automatically based on your builders config section.

Once your build VM is up and running the provisioners will be run. These are steps to configure and provision your machine. In the examples below we are mostly using the shell provisioner which will run commands on the VM to set things up.

Once your provisioning scripts have completed the VM will automatically stop, a snapshot will be taken and you will be provided with an ID which you can then use as a template in future runs of dask-cloudprovider.
10.3 Image Requirements

Each cluster manager that uses VMs will have specific requirements for the VM image.

The AWS ECSCluster for example requires ECS optimised AMIs.

The VM cluster managers such as EC2cluster and DropletCluster just require Docker to be installed (or NVIDIA Docker for GPU VM types).

10.4 Examples

10.4.1 EC2Cluster with cloud-init

When any of the VMCluster based cluster managers, such as EC2Cluster, launch a new default VM it uses the Ubuntu base image and installs all dependencies with cloud-init.

Instead of doing this every time we could use Packer to do this once, and then reuse that image every time.

Each VMCluster cluster manager has a class method called get_cloud_init which takes the same keyword arguments as creating the object itself, but instead returns the cloud-init file that would be generated.

```python
from dask_cloudprovider.aws import EC2Cluster
cloud_init_config = EC2Cluster.get_cloud_init(
    # Pass any kwargs here you would normally pass to `EC2Cluster`
)
print(cloud_init_config)
```

We should see some output like this.

```
#cloud-config
packages:
- apt-transport-https
- ca-certificates
- curl
- gnupg-agent
- software-properties-common

# Enable ipv4 forwarding, required on CIS hardened machines
write_files:
- path: /etc/sysctl.d/enabled_ipv4_forwarding.conf
  content: |
    net.ipv4.conf.all.forwarding=1

# create the docker group
groups:
- docker

# Add default auto created user to docker group
system_info:
  default_user:
    groups: [docker]

runcmd:
```

(continues on next page)
# Install Docker
- curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
- add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
- apt-get update -y
- apt-get install -y docker-ce docker-ce-cli containerd.io
- systemctl start docker
- systemctl enable docker

# Run container
- docker run --net=host daskdev/dask:latest dask-scheduler --version

We should save this output somewhere for reference later. Let’s refer to it as /path/to/cloud-init-config.yaml.

Next we need a Packer config file to build our image, let’s refer to it as /path/to/config.json. We will use the official Ubuntu 20.04 image and specify our cloud-init config file in the user_data_file option.

Packer will not necessarily wait for our cloud-init config to finish executing before taking a snapshot, so we need to add a provisioner that will block until the cloud-init completes.

```json
{
    "builders": [
        {
            "type": "amazon-ebs",
            "region": "eu-west-2",
            "source_ami_filter": {
                "filters": {
                    "virtualization-type": "hvm",
                    "name": "ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*.tar.gz",
                    "root-device-type": "ebs"
                },
                "owners": ["099720109477"],
                "most_recent": true
            },
            "instance_type": "t2.micro",
            "ssh_username": "ubuntu",
            "ami_name": "dask-cloudprovider {{timestamp}}",
            "user_data_file": "/path/to/cloud-init-config.yaml"
        }
    ],
    "provisioners": [
        {
            "type": "shell",
            "inline": [
                "echo 'Waiting for cloud-init'; while [ ! -f /var/lib/cloud/instance/boot-finished ]; do sleep 1; done; echo 'Done'
            ]
        }
    ]
}
```

Then we can build our image with packer build /path/to/config.json.

Then to use our new image we can create an EC2Cluster specifying the AMI and disabling the automatic bootstrapping.

10.4. Examples
from dask.distributed import Client
from dask_cloudprovider.aws import EC2Cluster

cluster = EC2Cluster(
    ami="ami-064f8db7634d19647", # AMI ID provided by Packer
    bootstrap=False
)
cluster.scale(2)

client = Client(cluster)
# Your cluster is ready to use

10.4.2 EC2Cluster with RAPIDS

To launch RAPIDS on AWS EC2 we can select a GPU instance type, choose the official Deep Learning AMIs that Amazon provides and run the official RAPIDS Docker image.

from dask_cloudprovider.aws import EC2Cluster

cluster = EC2Cluster(
    ami="ami-0c7c7d78f752f8f17", # Deep Learning AMI (this ID varies by region so find yours in the AWS Console)
    docker_image="rapidsai/rapidsai:cuda10.1-runtime-ubuntu18.04-py3.8",
    instance_type="p3.2xlarge",
    bootstrap=False, # Docker is already installed on the Deep Learning AMI
    filesystem_size=120,
)
cluster.scale(2)

However every time a VM is created by EC2Cluster the RAPIDS Docker image will need to be pulled from Docker Hub. The result is that the above snippet can take ~20 minutes to run, so let’s create our own AMI which already has the RAPIDS image pulled.

In our builders section we will specify we want to build on top of the latest Deep Learning AMI by specifying "Deep Learning AMI (Ubuntu 18.04) Version *" to list all versions and "most_recent": true to use the most recent.

We also restrict the owners to 898082745236 which is the ID for the official image channel.

The official image already has the NVIDIA drivers and NVIDIA Docker runtime installed so the only step we need to do is to pull the RAPIDS Docker image. That way when a scheduler or worker VM is created the image will already be available on the machine.

```json
{
    "builders": [
        {
            "type": "amazon-ebs",
            "region": "eu-west-2",
            "source_ami_filter": {
                "filters": {
                    "virtualization-type": "hvm",
                    "name": "Deep Learning AMI (Ubuntu 18.04) Version *",
                    "root-device-type": "ebs"
                },
                "owners": [898082745236]
            }
        }
    ]
}
```
Then we can build our image with `packer build /path/to/config.json`.

It took over 20 minutes to build this image, but now that we've done it once we can reuse the image in our RAPIDS powered Dask clusters.

We can then run our code snippet again but this time it will take less than 5 minutes to get a running cluster.

```python
from dask.distributed import Client
from dask_cloudprovider.aws import EC2Cluster

cluster = EC2Cluster(
    ami="ami-04e5539cb82859e69",  # AMI ID provided by Packer
docker_image="rapidsai/rapidsai:cuda10.1-runtime-ubuntu18.04-py3.8",
instance_type="p3.2xlarge",
bootstrap=False,
filesystem_size=120,
)
cluster.scale(2)

client = Client(cluster)
# Your cluster is ready to use
```
Releasing

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

```
git commit --allow-empty -m "Release x.x.x"
git tag -a x.x.x -m 'Version x.x.x'
git push upstream --tags
```
A
AzureMLCluster (class in dask_cloudprovider.azureml), 40
AzureVMCluster (class in dask_cloudprovider.azure), 33

C
close() (dask_cloudprovider.azureml.AzureMLCluster method), 43

D
dashboard_link (dask_cloudprovider.azureml.AzureMLCluster attribute), 43
DropletCluster (class in dask_cloudprovider.digitalocean), 21

E
EC2Cluster (class in dask_cloudprovider.aws), 8
ECSCluster (class in dask_cloudprovider.aws), 12

F
FargateCluster (class in dask_cloudprovider.aws), 16

G
GCPCluster (class in dask_cloudprovider.gcp), 26

J
jupyter_link (dask_cloudprovider.azureml.AzureMLCluster attribute), 43

S
scale() (dask_cloudprovider.azureml.AzureMLCluster method), 43
scale_down() (dask_cloudprovider.azureml.AzureMLCluster method), 43
scale_up() (dask_cloudprovider.azureml.AzureMLCluster method), 43