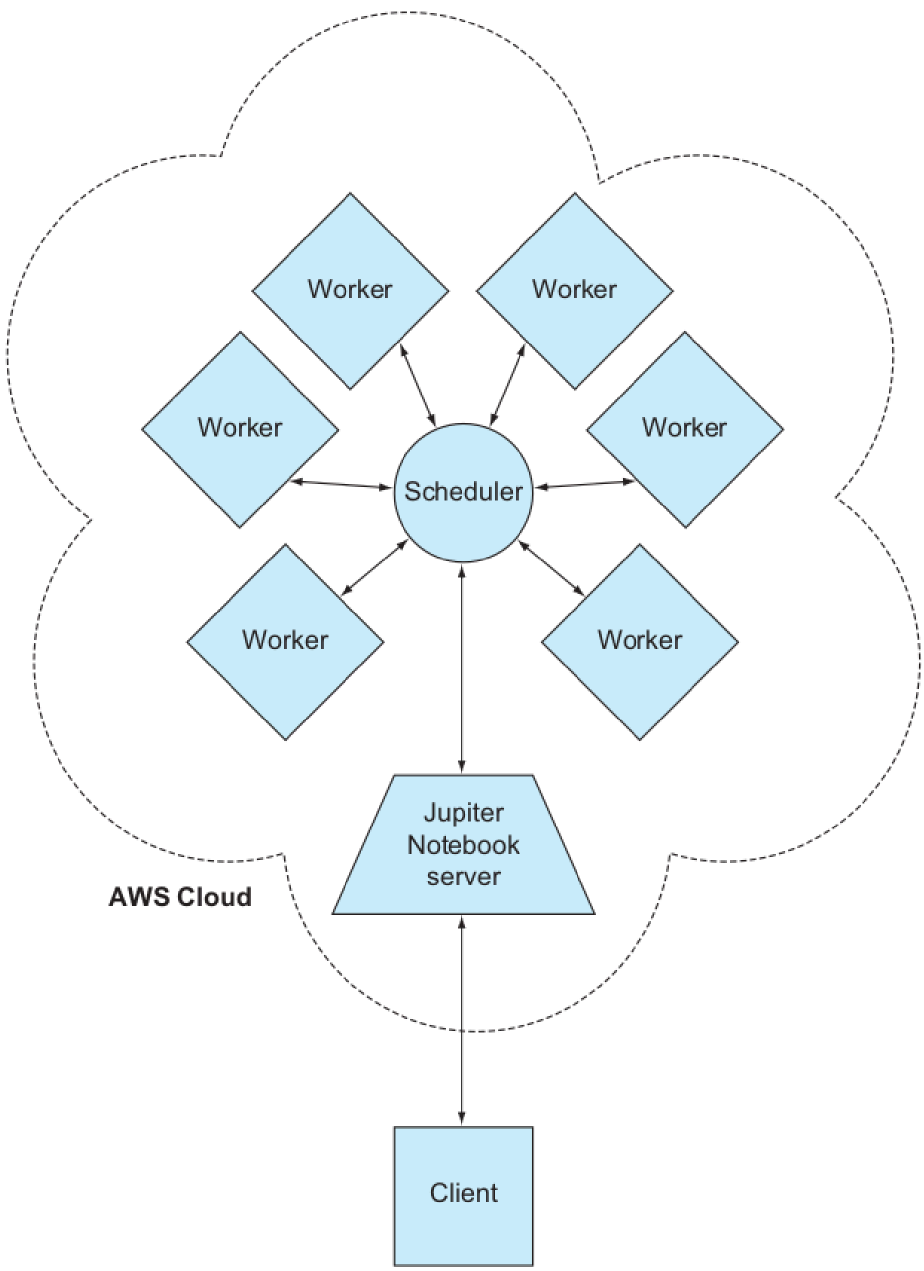
**Architecture of Dask Cluster in AWS**

The system has four distinct aspects: the jupyter server, the scheduler, the client and the workers. Their roles are as follows:

- Jupyter server - A frontend for the user to run code and submit jobs to the cluster
- Scheduler - Receives the jobs from the client via the Jupyter server, divides the work to be done, and coordinates the workers to complete the job
- Worker - Receives the tasks from the scheduler and computes them
- Client - Show the results to the user

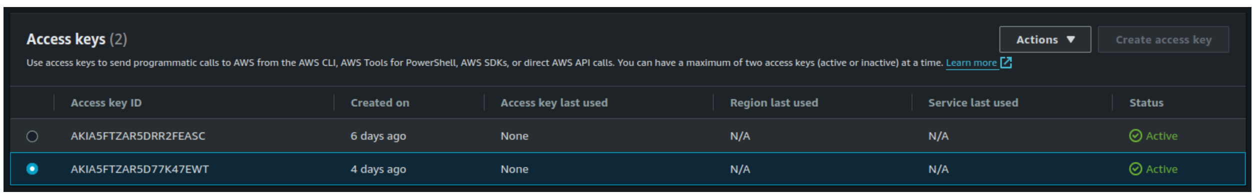


The setup involved a six-step process to get the cluster up and running:

1. Generate a security key.
2. Set up the ECS cluster.
3. Arrange the network settings for the cluster.
4. Establish a shared data drive using Elastic File System (EFS).
5. Reserve storage, build, and deploy the Docker images within Elastic Container Repository (ECR).
6. Connect to the cluster.

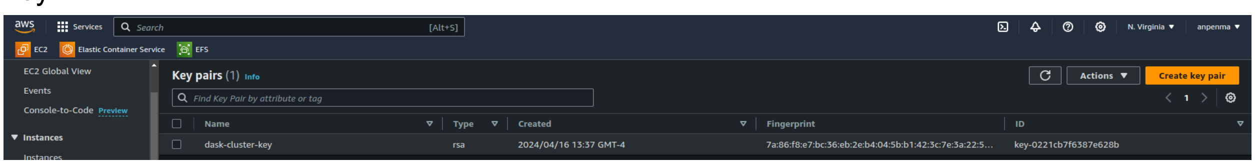
### Generate a Security Key

First a security key was created via the AWS security credentials page. This is needed for access via CLI in the local system to build and upload the docker images into the Elastic Container Repository (ECR).

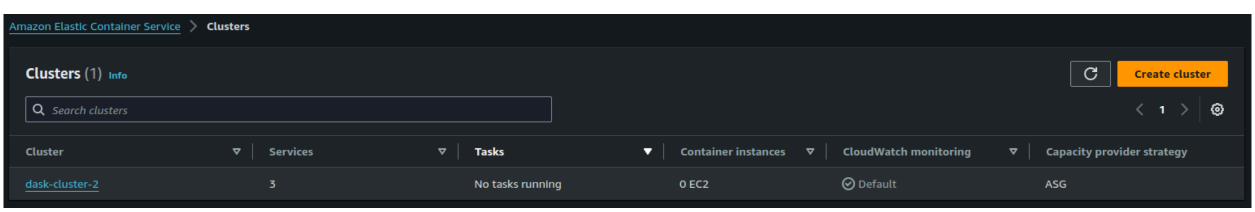


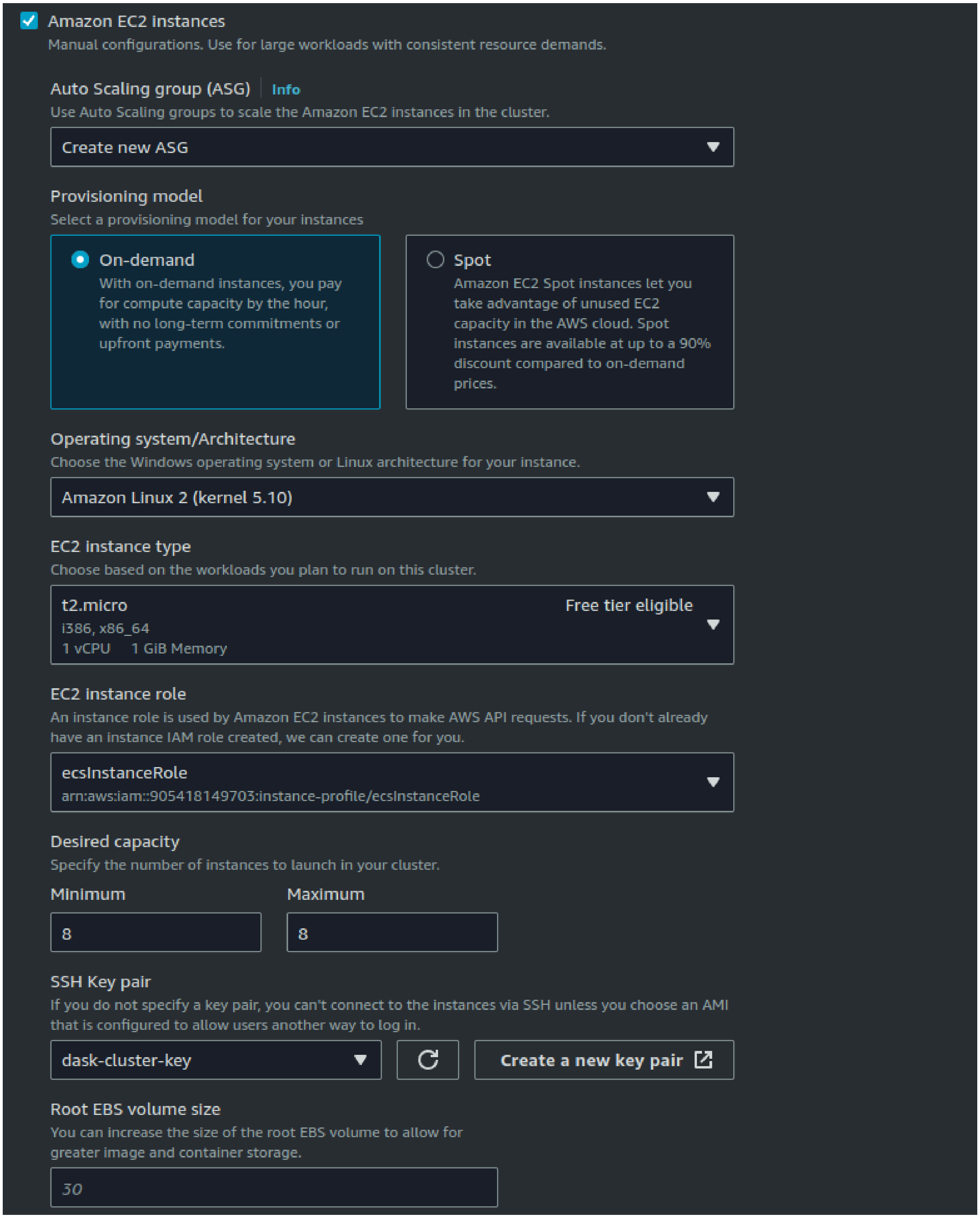

### Setting up the ECS Cluster

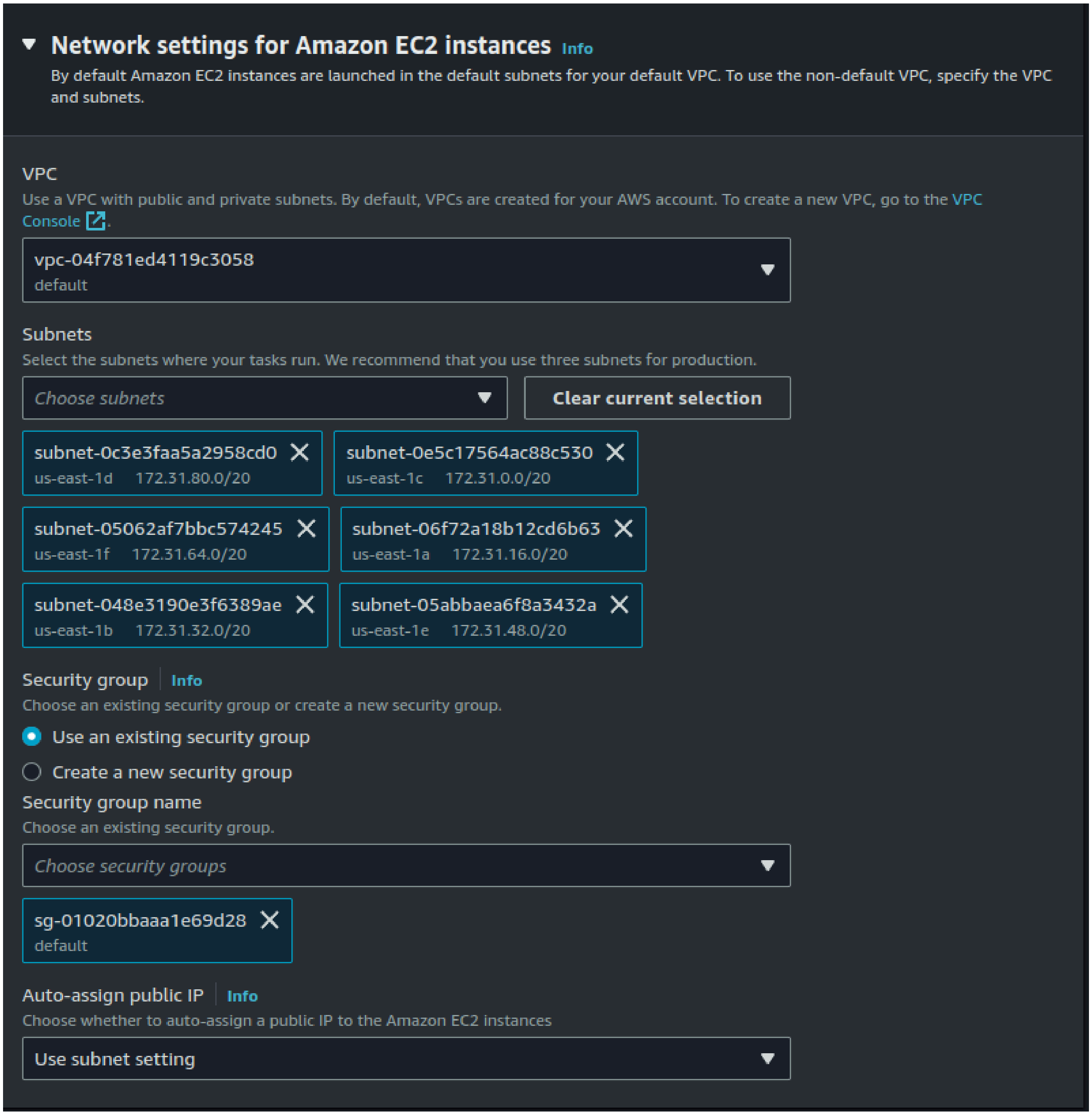
The first step in this process was to obtain an SSH key that we will use to associate with the EC2 instances that were created by the ECS cluster. To do this under the EC2 home page, there is an option for Key Pairs. With this a Key Pair was created with the name of 'Dask Cluster Key'.




With the Key Pair saved, next an ECS Cluster was to be created. This was done using the ECS Create Cluster wizard. The operating system chosen was Amazon Linux 2. Next for the EC2 instance type, we went with the t2.micro which was under the eligible types in the free tier. We went with the free tier first so that we didn't have to exhaust resources rapidly in just the setup. Following this a total of 8 instances were specified as the desired capacity. The Key-pair was specified as the one created earlier. Rest of the options were default. With this we had ECS start 8 instances under the name of dask-cluster.

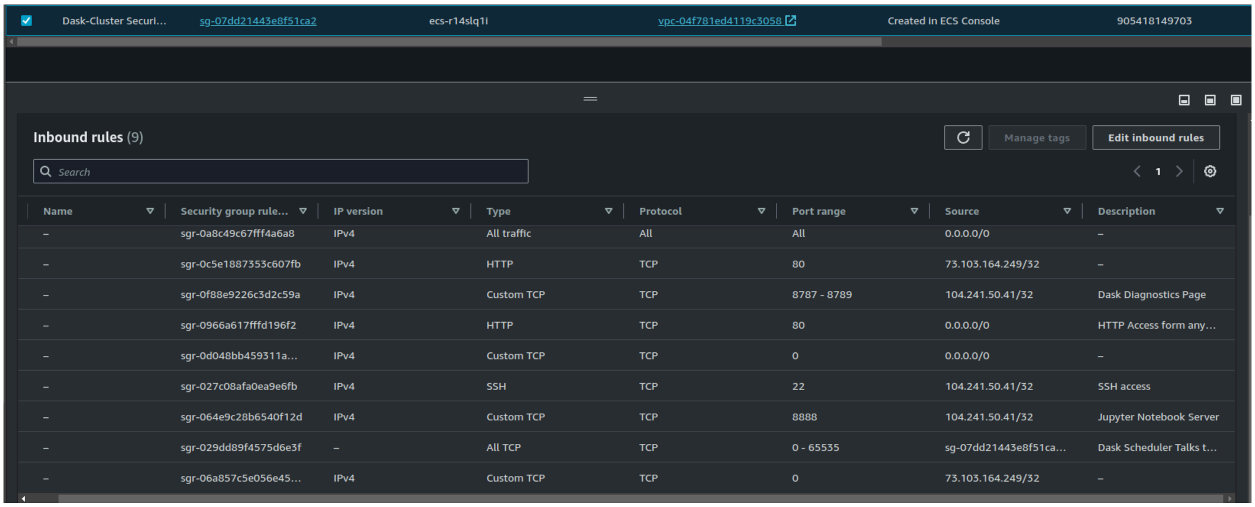



Amazon EC2 instances
Manual configurations. Use for large workloads with consistent resource demands.
Auto Scaling group (ASG)
Info
Use Auto Scaling groups to scale the Amazon EC2 instances in the cluster.
Create new ASG
Provisioning model
Select a provisioning model for your instances
On-demand
With on-demand instances, you pay for compute capacity by the hour, with no long-term commitments or upfront payments.
Spot
Amazon EC2 Spot instances let you take advantage of unused EC2 capacity in the AWS cloud. Spot instances are available at up to a 90% discount compared to on-demand prices.
Operating system/Architecture
Choose the Windows operating system or Linux architecture for your instance.
Amazon Linux 2 (kernel 5.10)
EC2 instance type
Choose based on the workloads you plan to run on this cluster.
t2.micro
Free tier eligible
i386, x86_64
1 vCPU 1 GiB Memory
EC2 instance role
An instance role is used by Amazon EC2 instances to make AWS API requests. If you don't already have an instance IAM role created, we can create one for you.
ecsInstanceRole
arn:aws:iam::905418149703:instance-profile/ecsInstanceRole
Desired capacity
Specify the number of instances to launch in your cluster.
Minimum
8
Maximum
8
SSH Key pair
If you do not specify a key pair, you can't connect to the instances via SSH unless you choose an AMI that is configured to allow users another way to log in.
dask-cluster-key
Create a new key pair
Root EBS volume size
You can increase the size of the root EBS volume to allow for greater image and container storage.
30


Network settings for Amazon EC2 instances Info
By default Amazon EC2 instances are launched in the default subnets for your default VPC. To use the non-default VPC, specify the VPC and subnets.
VPC
Use a VPC with public and private subnets. By default, VPCs are created for your AWS account. To create a new VPC, go to the VPC Console.
vpc-04f781ed4119c3058
default
Subnets
Select the subnets where your tasks run. We recommend that you use three subnets for production.
Choose subnets
Clear current selection
subnet-0c3e3faa5a2958cd0
us-east-1d 172.31.80.0/20
subnet-0e5c17564ac88c530
us-east-1c 172.31.0.0/20
subnet-05062af7bbc574245
us-east-1f 172.31.64.0/20
subnet-06f72a18b12cd6b63
us-east-1a 172.31.16.0/20
subnet-048e3190e3f6389ae
us-east-1b 172.31.32.0/20
subnet-05abbaea6f8a3432a
us-east-1e 172.31.48.0/20
Security group Info
Choose an existing security group or create a new security group.
Use an existing security group
Create a new security group
Security group name
Choose an existing security group.
Choose security groups
sg-01020bbaaa1e69d28
default
Auto-assign public IP Info
Choose whether to auto-assign a public IP to the Amazon EC2 instances
Use subnet setting

### Arrange the Network Settings for the Cluster

With the cluster running and the cluster's firewall rules needed to be configured so we can ssh to it. When ECS creates a cluster, it will automatically create a security group for it in EC2. So here a few inbound rules were added. For the ssh access, TCP protocol was assigned with port 22 and ip address as the user's current address. Second rule was type All TCP with the security group as the ip address for allowing Dask Scheduler to communicate with the workers. Then for the Jupyter server a custom TCP with port 8888 was created and ip address as the user's. Finally, for viewing the Dask diagnostics page, a custom TCP rule with port range of 8787 to 8789 and ip address as user's was selected. Most things with instances in a cluster are temporary like its ip address and compute resources. But if we want persistent storage, we have to leverage the Elastic File System in AWS.



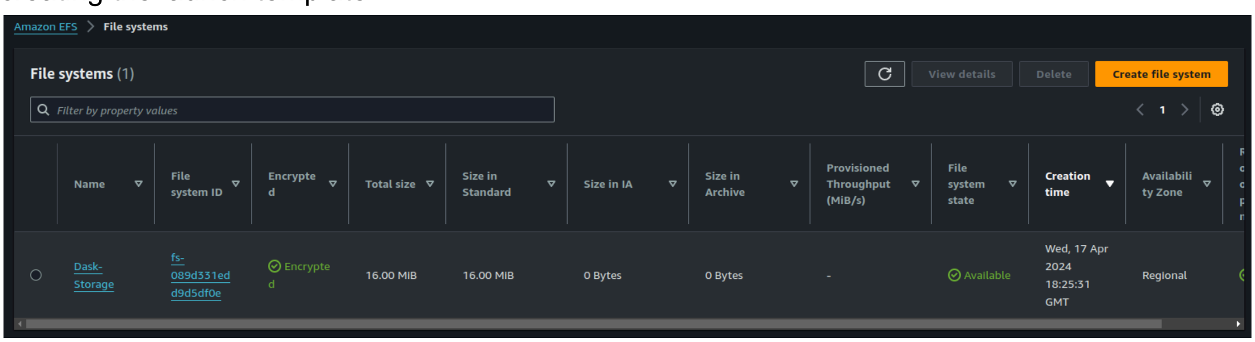
| | | | | | |
|---|---|---|---|---|---|
| Dask-Cluster Securi... | sg-07dd21443e8f51ca2 | ecs-r14slq1l | vpc-04f781ed4119c3058 | Created in ECS Console | 905418149703 |

Inbound rules (9)

| Name | Security group rule... | IP version | Type | Protocol | Port range | Source | Description |
|---|---|---|---|---|---|---|---|
| – | sgr-0a8c49c67fff4a6a8 | IPv4 | All traffic | All | All | 0.0.0.0/0 | – |
| – | sgr-0c5e1887353c607fb | IPv4 | HTTP | TCP | 80 | 73.103.164.249/32 | – |
| – | sgr-0f88e9226c3d2c59a | IPv4 | Custom TCP | TCP | 8787 - 8789 | 104.241.50.41/32 | Dask Diagnostics Page |
| – | sgr-0966a617fffd196f2 | IPv4 | HTTP | TCP | 80 | 0.0.0.0/0 | HTTP Access form any... |
| – | sgr-0d048bb459311a... | IPv4 | Custom TCP | TCP | 0 | 0.0.0.0/0 | – |
| – | sgr-027c08afa0ea9e6fb | IPv4 | SSH | TCP | 22 | 104.241.50.41/32 | SSH access |
| – | sgr-064e9c28b6540f12d | IPv4 | Custom TCP | TCP | 8888 | 104.241.50.41/32 | Jupyter Notebook Server |
| – | sgr-029dd89f4575d6e3f | – | All TCP | TCP | 0 - 65535 | sg-07dd21443e8f51ca... | Dask Scheduler Talks t... |
| – | sgr-06a857c5e056e45... | IPv4 | Custom TCP | TCP | 0 | 73.103.164.249/32 | – |

### Establishing a Shared Data Drive in using Elastic File System (EFS)

In order to create an EFS drive, the instances' VPC ID is needed. This was obtained from the EC2 Instance Manager. Post this, the EFS creation wizard was launched. In this, the VPC ID was provided. Additionally, security groups were removed and rest was left at default. With this the shared data drive was initiated. The DNS name for the filesystem was then saved for creating the launch template.



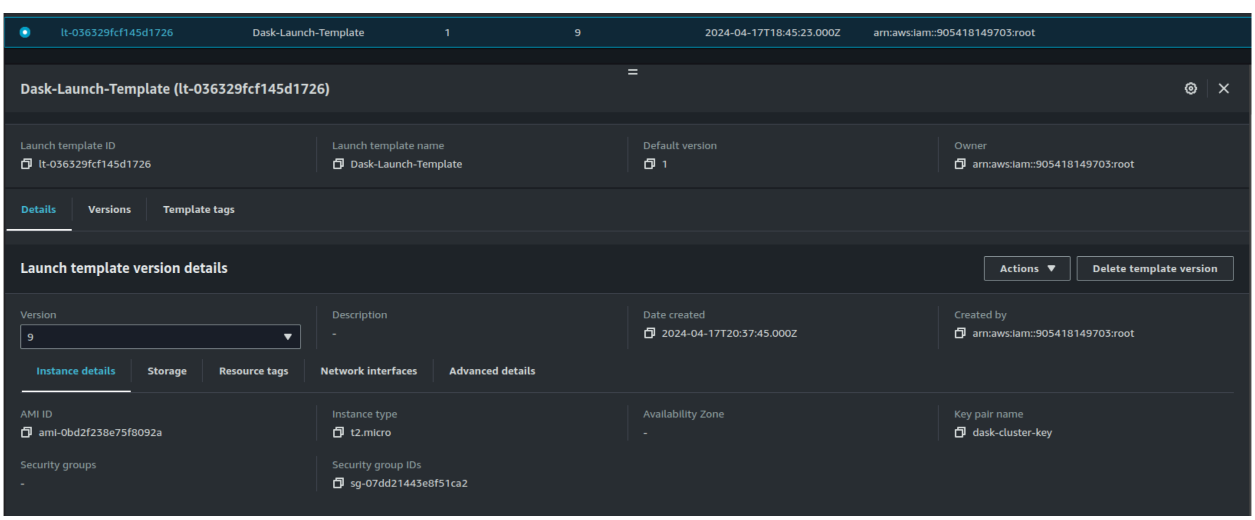
Amazon EFS > File systems

File systems (1)

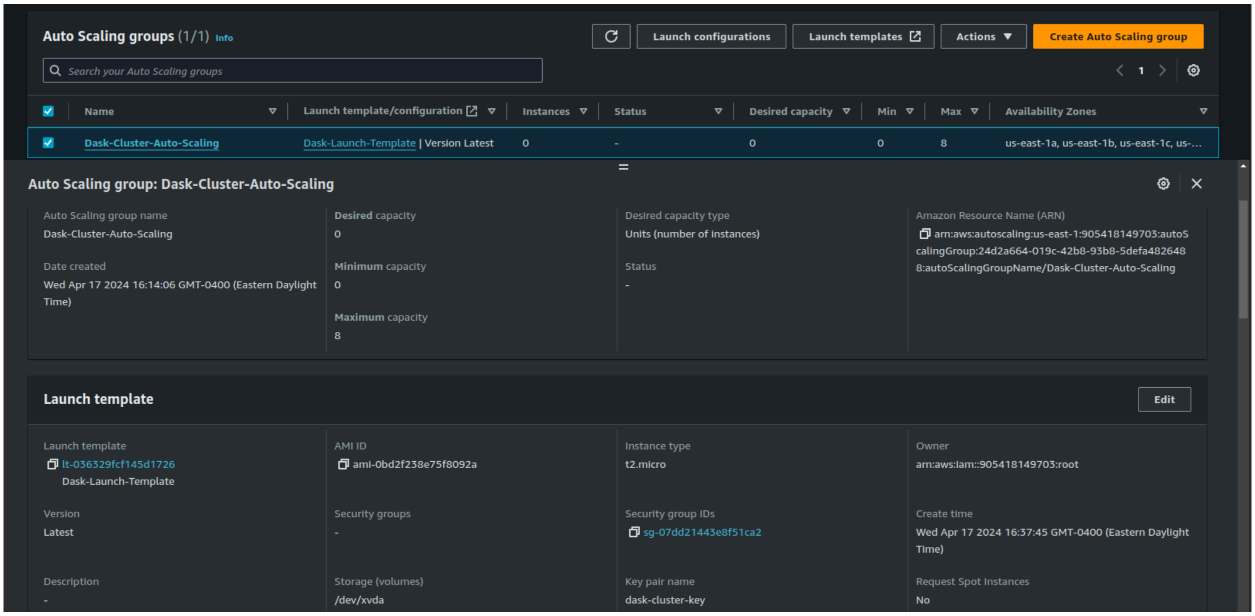
| Name | File system ID | Encrypted | Total size | Size in Standard | Size in IA | Size in Archive | Provisioned Throughput (MiB/s) | File system state | Creation time | Availability Zone |
|---|---|---|---|---|---|---|---|---|---|---|
| Dask-Storage | fs-089d331edd9d5df0e | Encrypted | 16.00 MiB | 16.00 MiB | 0 Bytes | 0 Bytes | - | Available | Wed, 17 Apr 2024 18:25:31 GMT | Regional |

Next, we had to create a Launch template that is to be provided to the auto-scaling group for automatically launching the instances. For this we copied an existing ECS template and modified the security group and provided the below code to be run on startup for each instance which includes the DNS for the file system and cluster they need to be associated to:

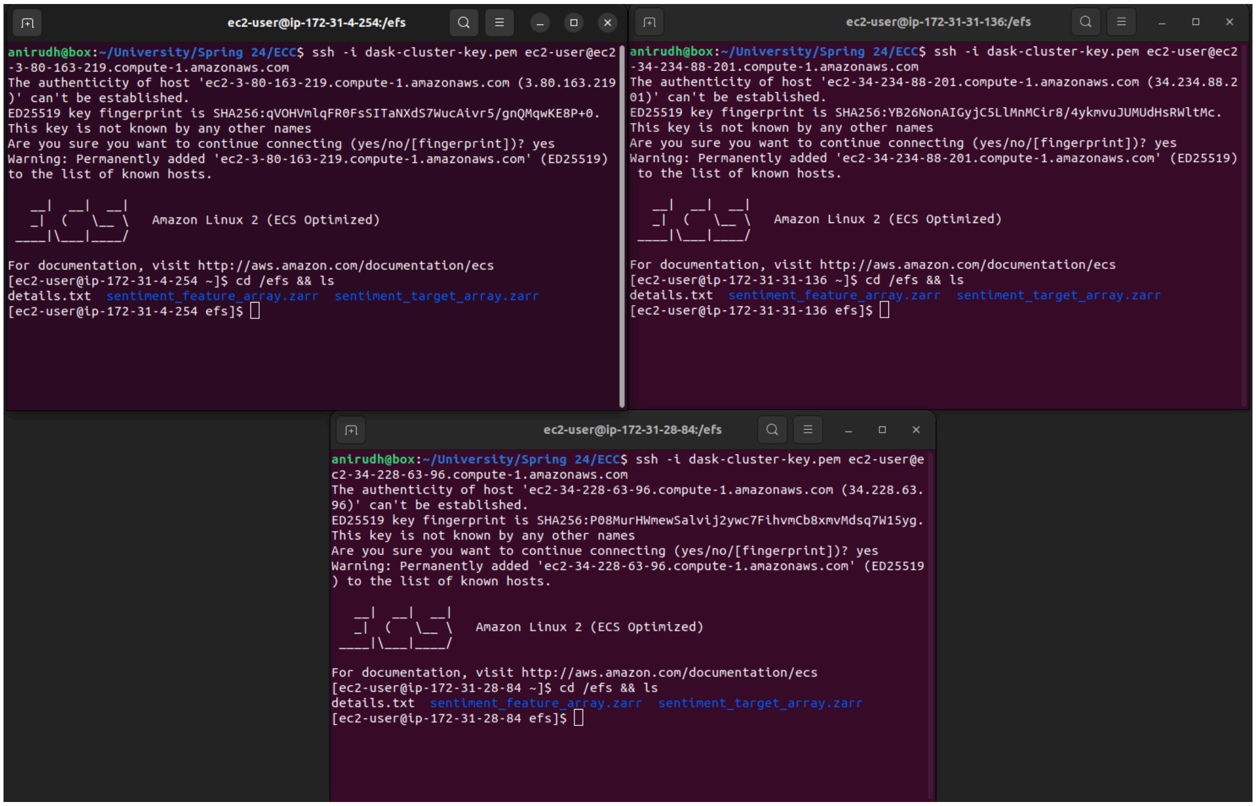
```
Content-Type: multipart/mixed; boundary="==BOUNDARY=="
MIME-Version: 1.0

--==BOUNDARY==
Content-Type: text/cloud-boothook; charset="us-ascii"

# Install nfs-utils
cloud-init-per once yum_update yum update -y
cloud-init-per once install_nfs_utils yum install -y nfs-utils

# Create /efs folder
cloud-init-per once mkdir_efs mkdir /efs

# Mount /efs
cloud-init-per once mount_efs echo -e
'fs-089d331edd9d5df0e.efs.us-east-1.amazonaws.com:/ /efs nfs4
nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2
0 0' >>
/etc/fstab
mount -a

--==BOUNDARY==
Content-Type: text/x-shellscript; charset="us-ascii"

#!/bin/bash
echo ECS_CLUSTER=dask-cluster-2 >> /etc/ecs/ecs.config;
echo ECS_BACKEND_HOST= >> /etc/ecs/ecs.config
--==BOUNDARY==--
```



With the launch template created, the next step involves supplying it to the auto-scaling group. For the auto-scaling group creation, a few inputs were given such as VPC, launch template, subnets, and group size. With this we could finally start the auto-scaling feature working after terminating all the old instances.

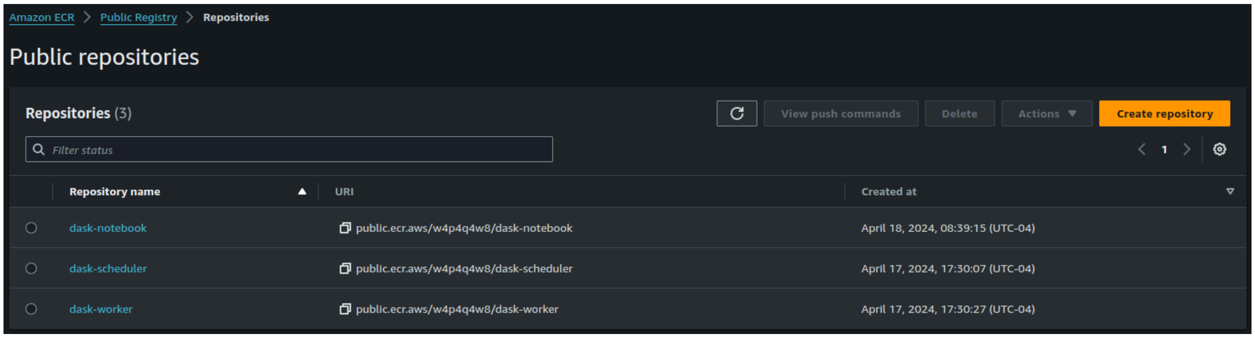



Once all the instances were running, a test had to be performed to check whether the EFS drive had been successfully mounted onto all the instances. For this using a file was sent to an instance using the scp from a local system. Once it was sent, the expectation was that the file would be in all of the remaining instances. After checking all the instances we were able to confirm that indeed the file was present everywhere.

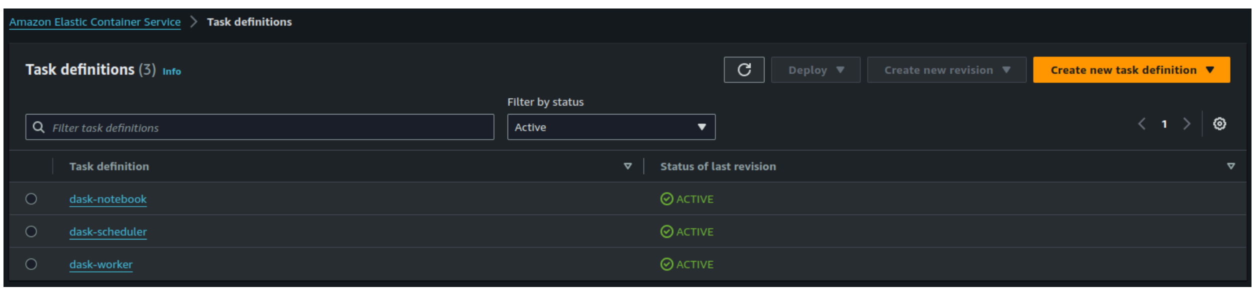


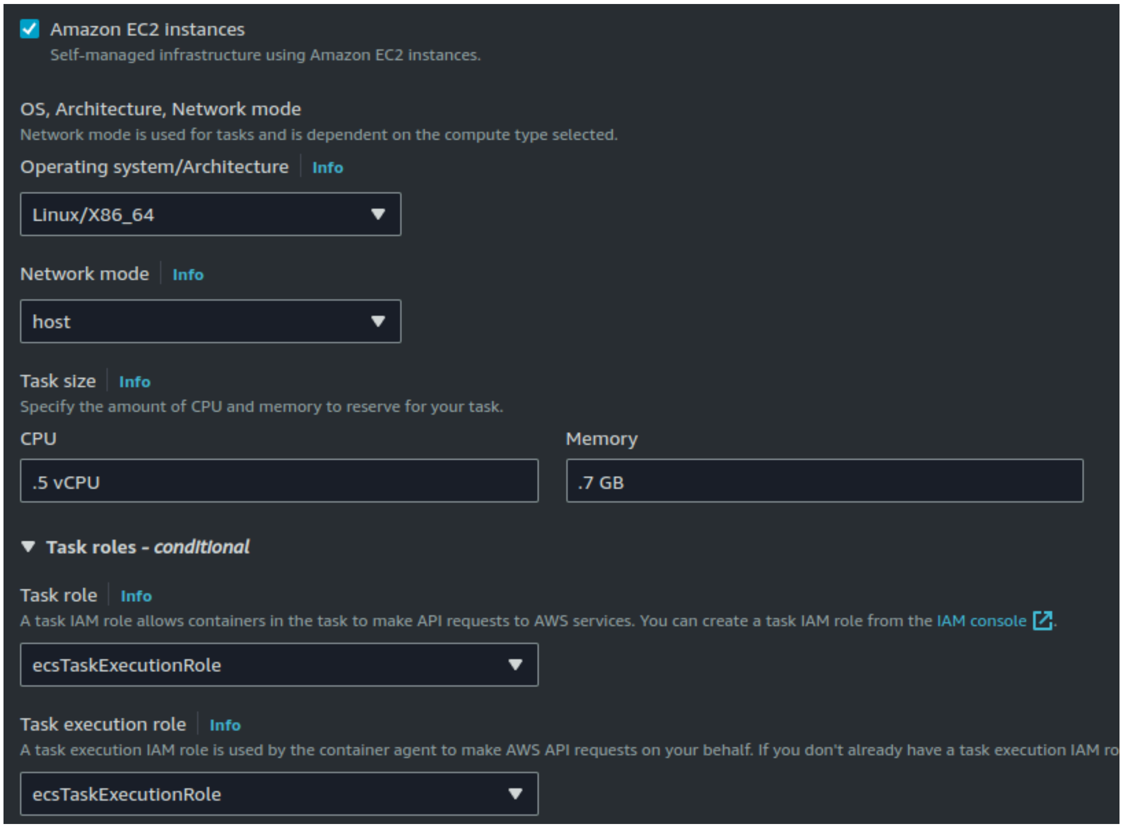

**Reserve storage, build, and deploy the Docker images within Elastic Container Repository (ECR)**

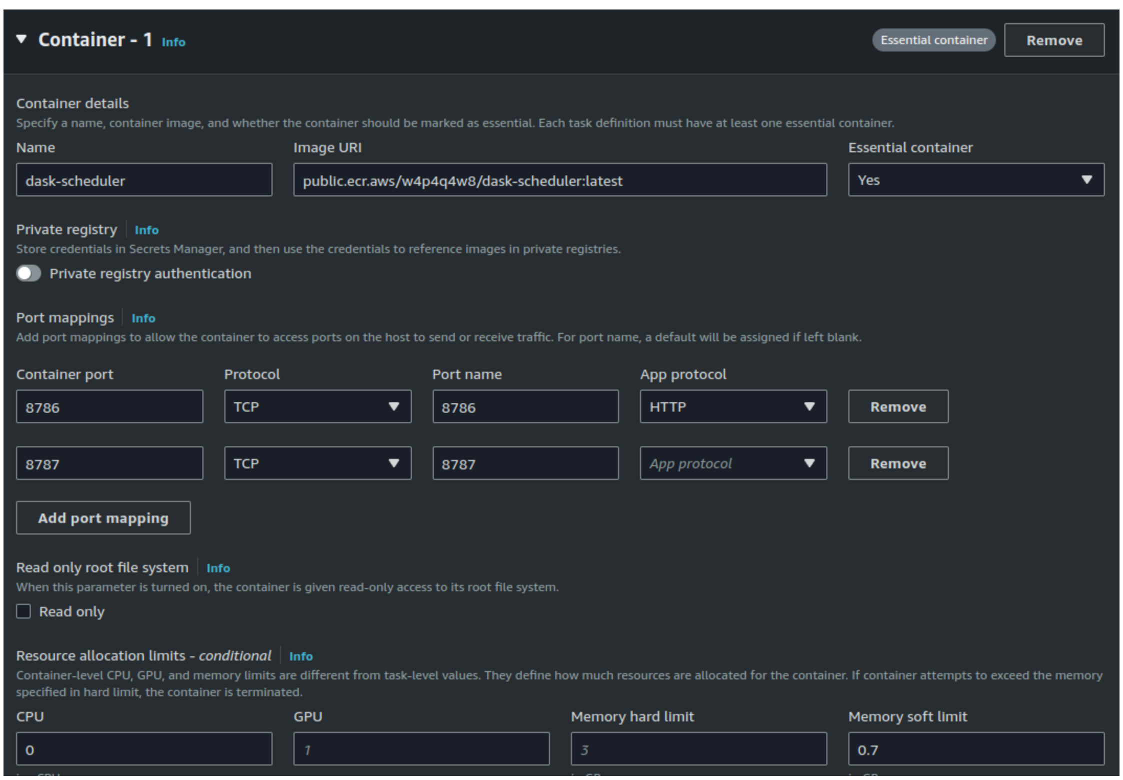
The purpose of this step is to build our Docker images and upload them into the ECR. Thankfully, AWS has streamlined and simplified the process a lot. But in order for the images to be uploaded via CLI in the local system, first AWS CLI had to be configured in the local machine. This involved creating an IAM user with permission to upload and modify files. With the user created and logged into the AWS CLI, the docker commands to be used were already provided in the ECR under the option push commands and were run in the terminal for each repository separately. There were 3 repositories created for this project: dask-notebook, dask-scheduler, dask-worker. But it will always initialize dask-scheduler first and only then notebook and workers will start as scheduler’s ip address is required for the nodes to establish communication.

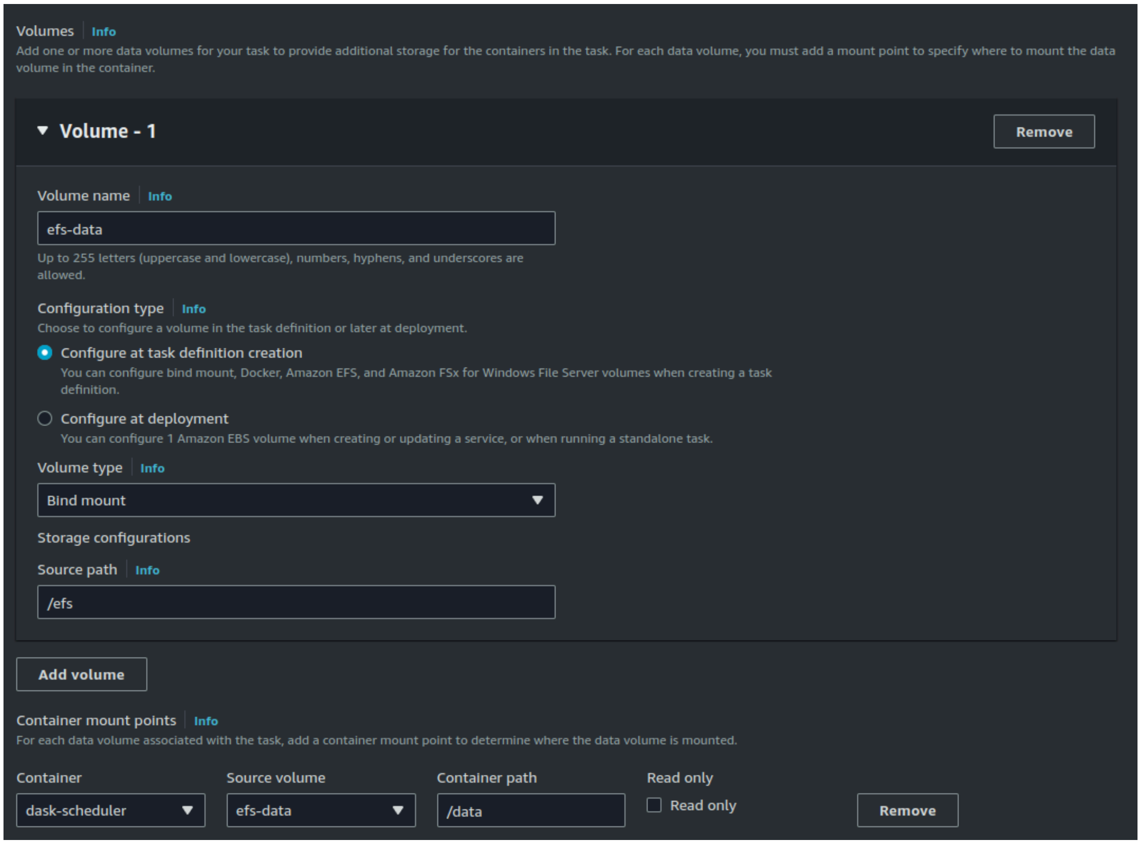



With the docker images in-place, a task definition had to be supplied for the images to be initialized as containers in the instances. Under the task definition creation wizard, the cpu and memory were allocated according to t2.micro's capacity, for the container the image URI for the respective repositories were provided. Additionally, the container ports were mapped to specific host ports so as to avoid any conflicts. Logs were enabled so as to obtain the Jupyter server's token to login and to also identify problems when something goes wrong. Additionally, the EFS drive location was mentioned as the mounting point for a shared data space.




Amazon EC2 instances
Self-managed infrastructure using Amazon EC2 instances.
OS, Architecture, Network mode
Network mode is used for tasks and is dependent on the compute type selected.
Operating system/Architecture | Info
Linux/X86_64
Network mode | Info
host
Task size | Info
Specify the amount of CPU and memory to reserve for your task.
CPU
.5 vCPU
Memory
.7 GB
Task roles - *conditional*
Task role | Info
A task IAM role allows containers in the task to make API requests to AWS services. You can create a task IAM role from the IAM console.
ecsTaskExecutionRole
Task execution role | Info
A task execution IAM role is used by the container agent to make AWS API requests on your behalf. If you don't already have a task execution IAM ro
ecsTaskExecutionRole


Container - 1 Info
Essential container
Remove
Container details
Specify a name, container image, and whether the container should be marked as essential. Each task definition must have at least one essential container.
Name
dask-scheduler
Image URI
public.ecr.aws/w4p4q4w8/dask-scheduler:latest
Essential container
Yes
Private registry | Info
Store credentials in Secrets Manager, and then use the credentials to reference images in private registries.
Private registry authentication
Port mappings | Info
Add port mappings to allow the container to access ports on the host to send or receive traffic. For port name, a default will be assigned if left blank.
Container port
Protocol
Port name
App protocol
8786
TCP
8786
HTTP
Remove
8787
TCP
8787
App protocol
Remove
Add port mapping
Read only root file system | Info
When this parameter is turned on, the container is given read-only access to its root file system.
Read only
Resource allocation limits - *conditional* | Info
Container-level CPU, GPU, and memory limits are different from task-level values. They define how much resources are allocated for the container. If container attempts to exceed the memory specified in hard limit, the container is terminated.
CPU
0
GPU
1
Memory hard limit
3
Memory soft limit
0.7




### Connecting to the Cluster

With all these in place, all we had to do to connect to the cluster was to use the public DNS name of the notebook and scheduler to view them on the local browser. The notebook was present on port 8888 and dask's diagnostics page was located on port 8787. With all of these steps successfully done, we could now connect to the cluster and begin running some jobs. To test this, some sample Dask code was used that was already provided as part of the Jupyter server.

```
[3]: # Listing 11.2
from dask.distributed import Client, progress
client = Client('ip-172-31-91-162.ec2.internal:8786')
client
```

```
/opt/conda/lib/python3.10/site-packages/distributed/client.py:1393: VersionMismatchWarning: Mismatched versions found

+---------+----------------+----------------+----------------+
| Package | Client         | Scheduler      | Workers        |
+---------+----------------+----------------+----------------+
| python  | 3.10.14.final.0 | 3.10.12.final.0 | 3.10.12.final.0 |
| toolz   | 0.12.1         | 0.12.0         | 0.12.0         |
+---------+----------------+----------------+----------------+
  warnings.warn(version_module.VersionMismatchWarning(msg[0]["warning"]))
```

[3]:

**Client**

Client-1e85316c-fd9b-11ee-82ee-0242ac110002

**Connection method:** Direct

**Dashboard:** http://ip-172-31-91-162.ec2.internal:8787/status

Launch dashboard in JupyterLab

▾ Scheduler Info

**Scheduler**

Scheduler-53dcc6a7-8264-4210-ae0e-756f4073875f

| | |
|---|---|
| **Comm:** tcp://172.31.91.162:8786 | **Workers:** 6 |
| **Dashboard:** http://172.31.91.162:8787/status | **Total threads:** 6 |
| **Started:** 28 minutes ago | **Total memory:** 4.20 GiB |

▸ Workers

Here we are able to see that Dask is able to successfully connect with the scheduler using its IP address that was provided manually. Ideally just defining an environment variable that stores the scheduler's ip address should tell Dask where the scheduler is. But for some mysterious reason, it just wasn't working. So the scheduler ip address had to be provided in the code.

Checking to see if the dask jobs were being successfully submitted and distributed among the 6 worker nodes.