# Proposed Publishing Date

| **Role/Reviewed** | **Name** | **Date completed** |
| --- | --- | --- |
| Authors | **Ashish Aggarwal** | **02/23/2021** |
| Product and technical review | **Katherine Dube** | **02/23/2021** |
| Technical Review | **Appointed by Ashish Aggarwal** | **02/23/2021** |
| Legal OK |  |  |
| Final cleanup |  |  |
| FINAL blog posted | **Alex Handy (openshift.com)** |  |

# 

Running an OpenShift Worker Node on AWS Wavelength for Edge Applications

This article provides the steps to run OpenShift on edge computing solutions being deployed by AWS. This provides a unique value proposition for applications that want to remain agnostic toward underlying infrastructure and run on any public cloud provider based solution.

# AWS Wavelength

AWS Wavelength is an AWS Infrastructure offering optimized for mobile edge computing applications. Wavelength Zones are AWS infrastructure deployments that embed AWS compute and storage services within communications service providers’ (CSP) data centers at the edge of the 5G network. This allows application traffic from 5G devices to reach application servers running in Wavelength Zones without leaving the telecommunications network. This avoids the latency that would result from application traffic having to traverse multiple hops across the internet to reach their destination, enabling customers to take full advantage of the latency and bandwidth benefits offered by modern 5G networks. AWS Wavelength Zones are available in 10 U.S. with Verizon, in Tokyo, Japan, with KDDI, and in Daejeon, South Korea, with SKT.

## Wavelength Concepts

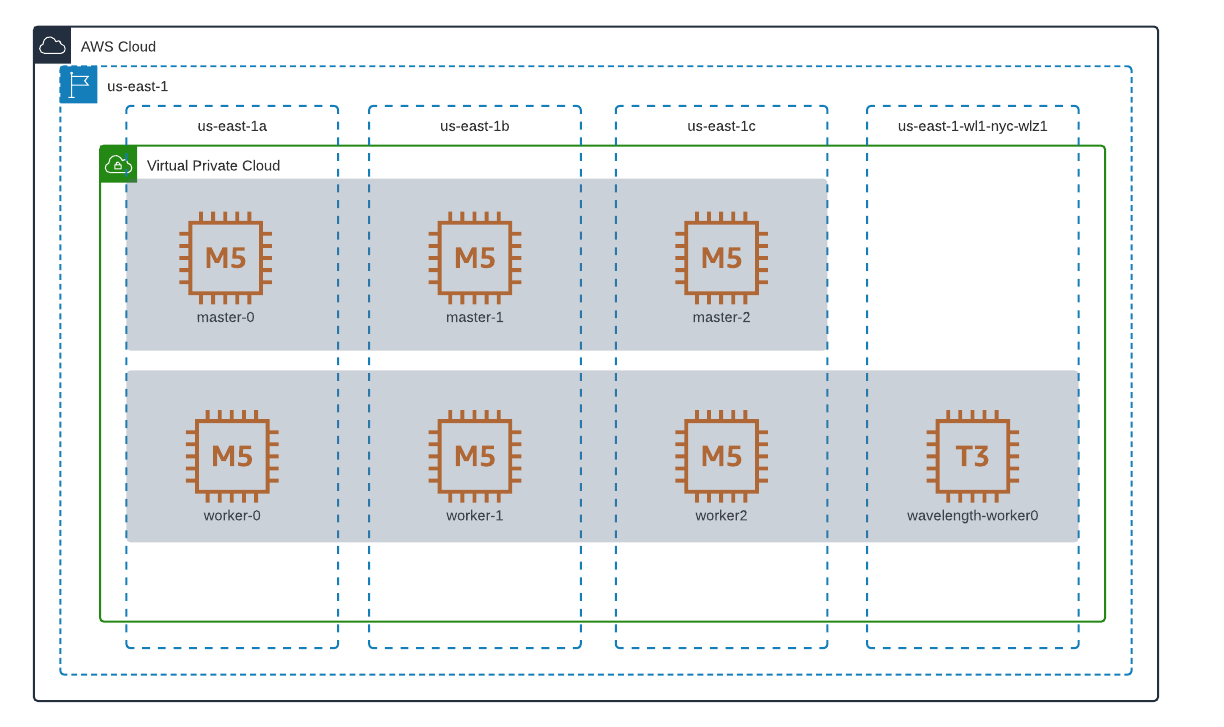
The following are the key concepts for Wavelengths:

* **Wavelength** — A new type of AWS infrastructure designed to run workloads that require ultra-low latency over mobile networks.
* **Wavelength Zone (WZ)** — A zone in the carrier location where the Wavelength infrastructure is deployed. Wavelength Zones are associated with an AWS Region. A Wavelength Zone is a logical extension of the region, and is managed by the control plane in the region.
* **VPC** — A customer virtual private cloud (VPC) that spans Availability Zones, Local Zones, and Wavelength Zones, and has deployed resources such as Amazon EC2 instances in the subnets that are associated with the zones.
* **Subnet** — A subnet that you create in a Wavelength Zone. You can create one or more subnets, and then run and manage AWS services, such as Amazon EC2 instances, in the subnet.
* **Carrier Gateway** — A carrier gateway serves two purposes. It allows inbound traffic from a carrier network in a specific location and allows outbound traffic to the carrier network and internet.
* **Network Border Group** — A unique set of Availability Zones, Local Zones, or Wavelength Zones from which AWS advertises IP addresses.
* **Wavelength Application** — An application that you run on an AWS resource in a Wavelength Zone.

## How Wavelength Works

* Extends the Amazon VPC to include a Wavelength Zone and then creates AWS resources like Amazon EC2 instances in the desired subnets
* Deploys the portions of an application that require ultra-low latency in a Wavelength Zone and then seamlessly connect back to the rest of the application and the full range of cloud services running in the AWS Region
* Allows application traffic to reach application servers running in Wavelength Zones without leaving the mobile network

# Deploying OpenShift

[](https://lucid.app/documents/edit/f7f19b43-aa95-46c0-8782-5ede547520ab/0?callback=close&name=docs&callback_type=back&v=435&s=612)

The above diagram is a high-level overview of deployment architecture for OCP in AWS with wavelength. We have first deployed Red Hat OpenShift using the installer-provisioned infrastructure (IPI) deployment method on an existing VPC with predefined subnets. We did this because if the installer is left to create the underlying infrastructure, it will use the entire VPC CIDR, and we’re left with no option for creating additional subnets for the wavelength zone. Now let us see in detail how this was implemented.

## Creating VPC and Subnets

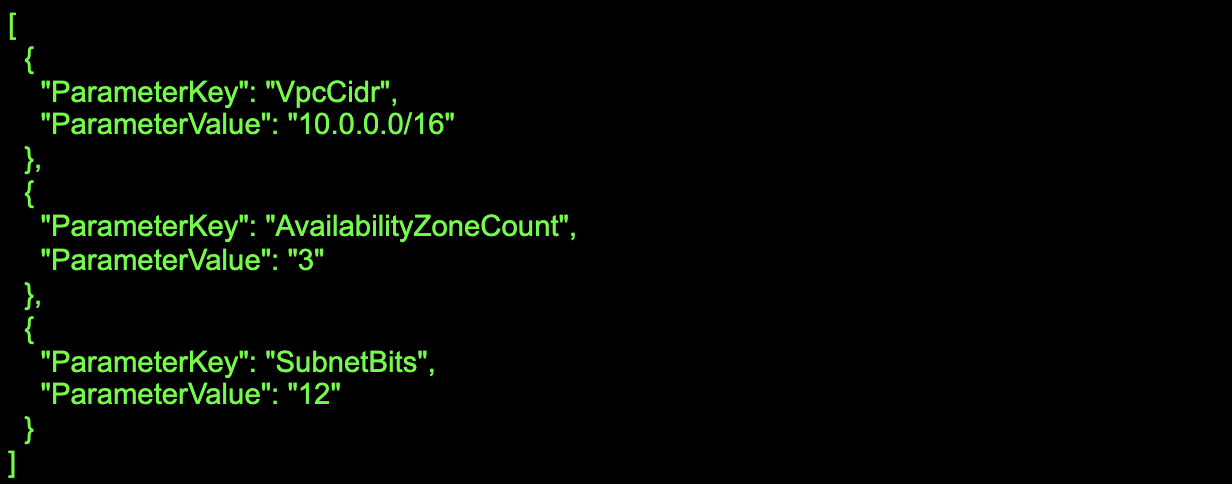
We will start with first creating VPC and subnets in the AWS region. For this, we will use a cloudformation template to deploy required VPC and subnets. We can refer to [this guide](https://docs.openshift.com/container-platform/4.6/installing/installing_aws/installing-aws-user-infra.html) for information on creating VPC and subnets for OCP using cloudformation templates. Keep these in mind while creating VPC and subnets:

1. Subnets bits
2. Availability Zone Count

Following is the template I used for deploying VPC and subnets for this test:

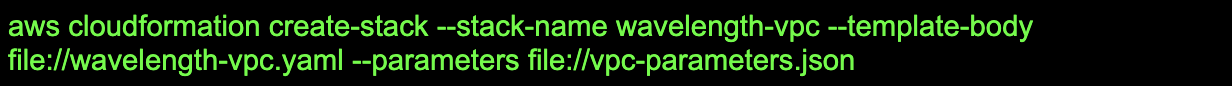
Vpc-parameters.json

| [  {  "ParameterKey": "VpcCidr",  "ParameterValue": "10.0.0.0/16"  },  {  "ParameterKey": "AvailabilityZoneCount",  "ParameterValue": "3"  },  {  "ParameterKey": "SubnetBits",  "ParameterValue": "12"  }  ] |
| --- |



Create the VPC and subnets using aws cli command:

| aws cloudformation create-stack --stack-name wavelength-vpc --template-body file://wavelength-vpc.yaml --parameters file://vpc-parameters.json |
| --- |

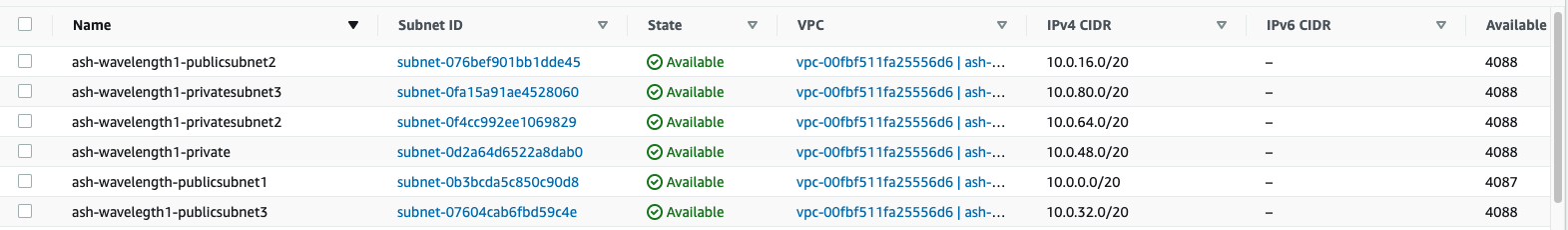


On successful completion of above command, you will have VPC and Subnets:

VPC



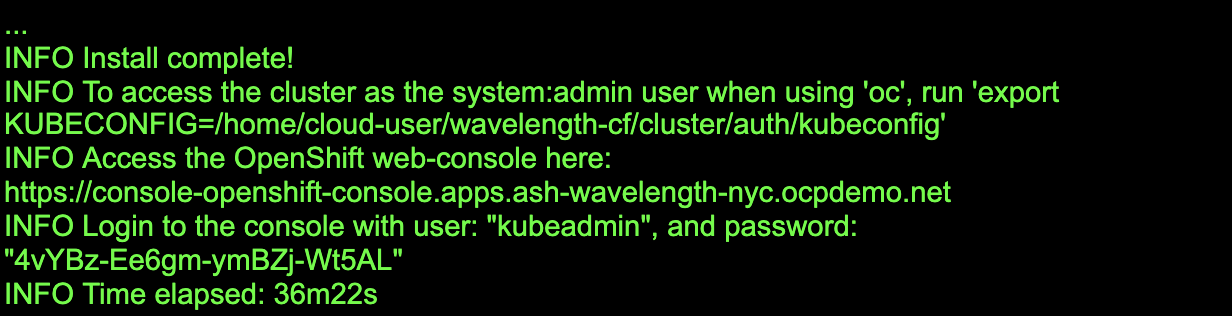
Subnets (names were given later manually)



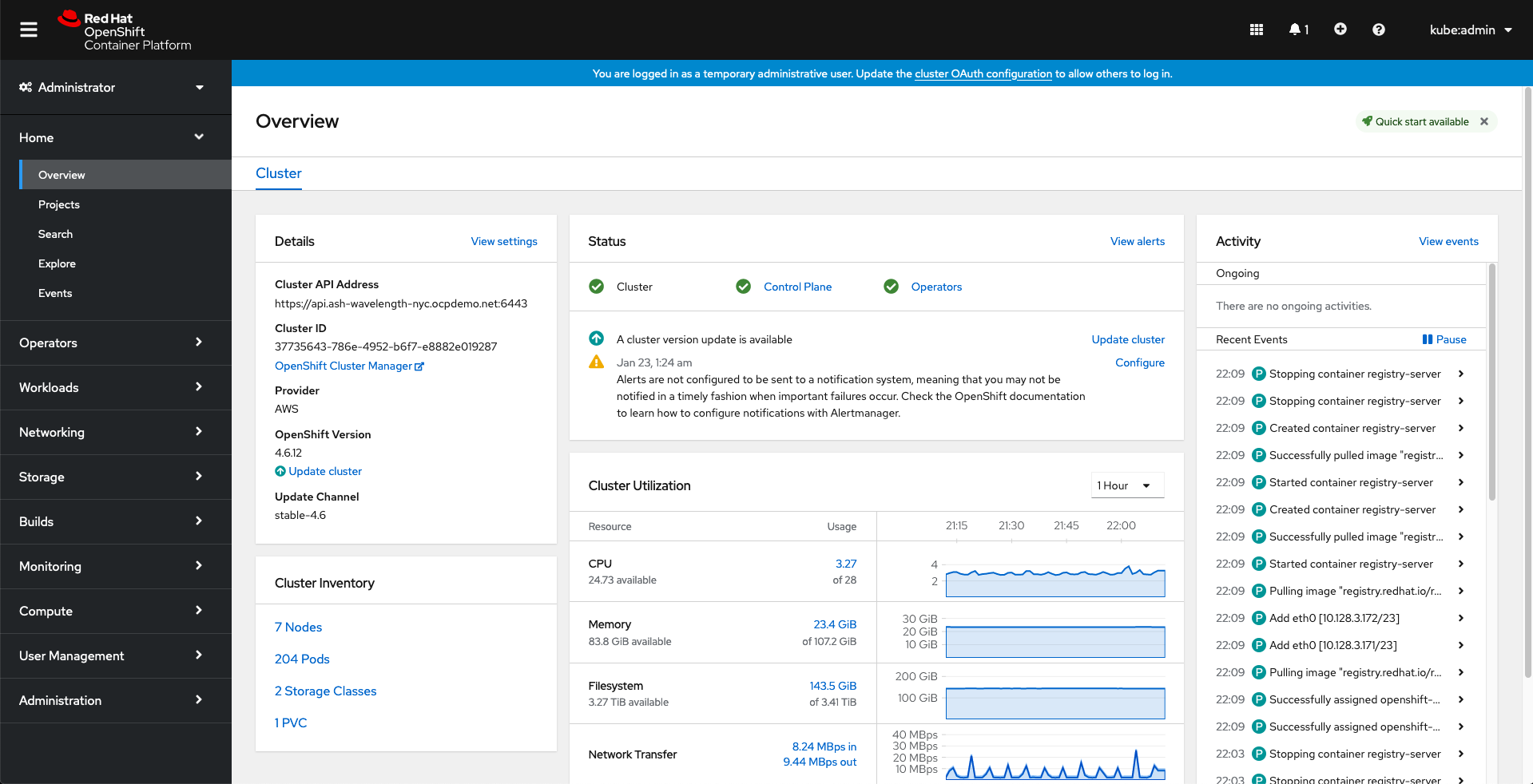
## Installer-Provisioned Infrastructure (IPI) Installation

Once the required VPC and subnets are created, we will proceed to install the OpenShift cluster. We will be using the IPI installation method. For IPI installation with existing VPC, please follow the [Installing a cluster on AWS into an existing VPC](https://docs.openshift.com/container-platform/4.6/installing/installing_aws/installing-aws-vpc.html) document. It will take around 35 to 40 minutes to complete the installation.

| ...  INFO Install complete!  INFO To access the cluster as the system:admin user when using 'oc', run 'export KUBECONFIG=/home/cloud-user/wavelength-cf/cluster/auth/kubeconfig'  INFO Access the OpenShift web-console here: https://console-openshift-console.apps.ash-wavelength-nyc.ocpdemo.net  INFO Login to the console with user: "kubeadmin", and password: "4vYBz-Ee6gm-ymBZj-Wt5AL"  INFO Time elapsed: 36m22s |
| --- |

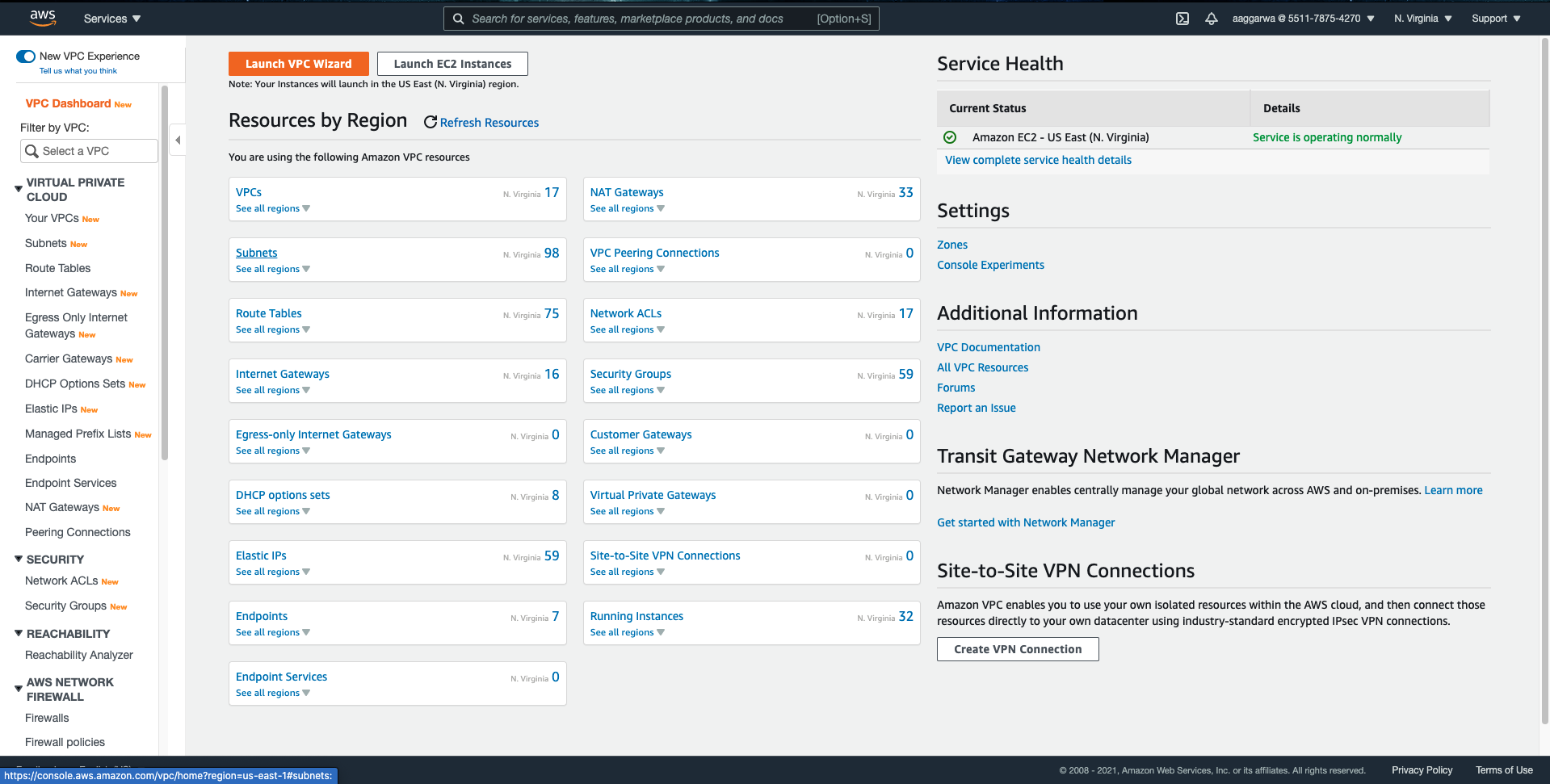


Once the installation is complete, login to the web console to see if the cluster is ready and operational.

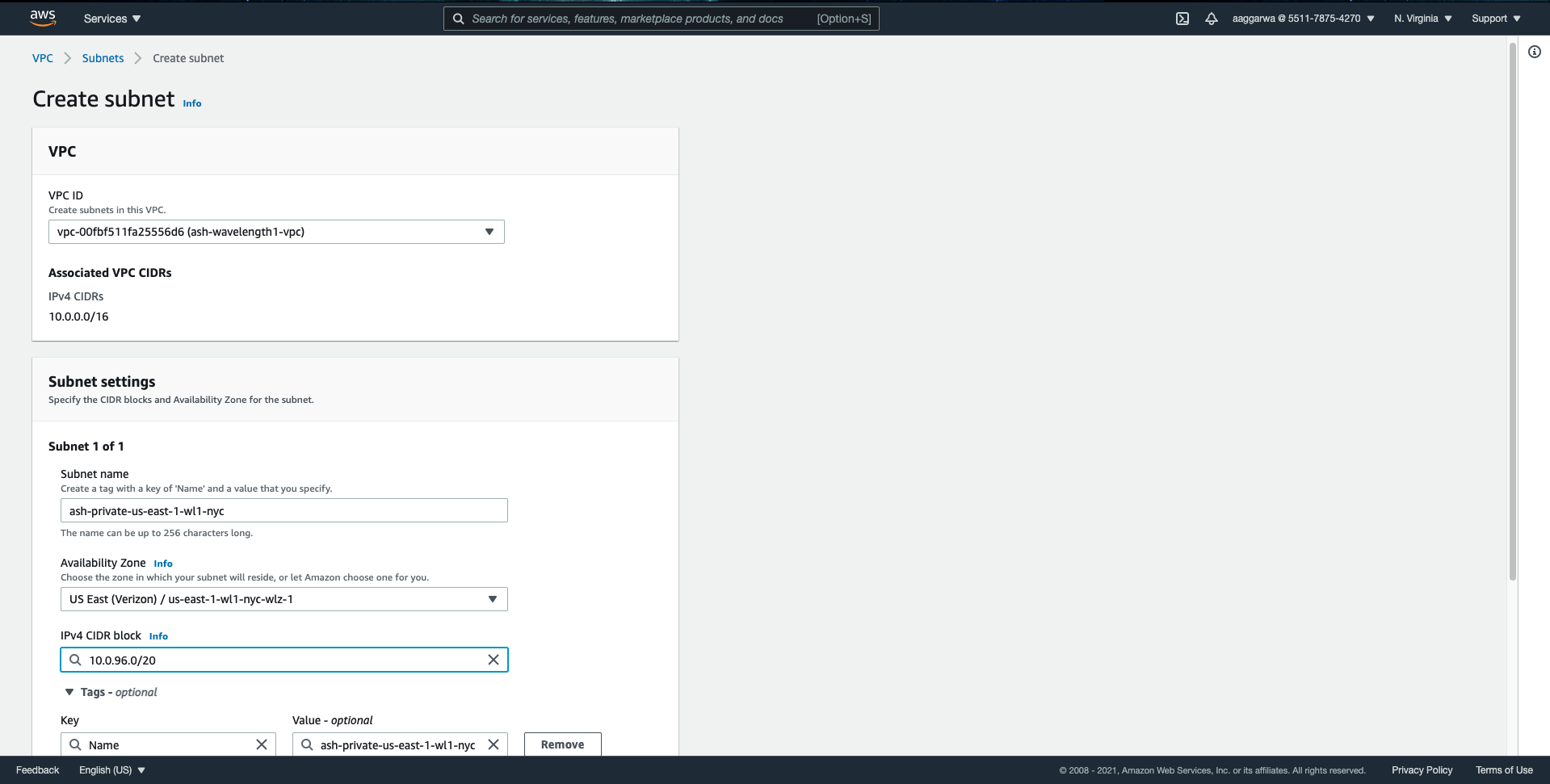


**Creating subnet in VPC for AWS wavelength**

Login to AWS web console and go to ‘Services>VPC’ and click on the subnets:



Create the subnet by providing the VPC ID, Subnet name, Availability Zone, and CIDR for subnet:



Make sure that availability zone is in the wavelength zone. Also, create the Carrier Gateway for wavelength subnet.

**Creating MachineSets for Wavelength Zone**

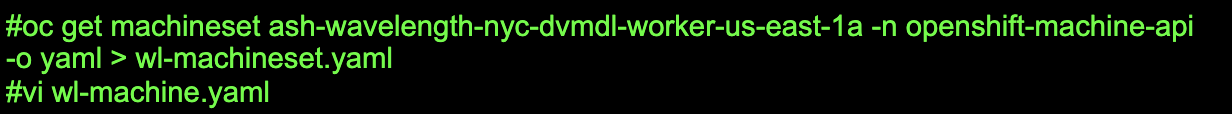
Now we will create a MachineSet for the AWS wavelength zone. (A MachineSet is a group of machines. MachineSets are to the machines as ReplicaSets are to the Pods. MachineSets have a template of machines’ specifications. For more information on MachineSets, please refer to [this document](https://docs.openshift.com/container-platform/4.1/machine_management/creating-machineset.html).)

| # oc get machinesets -n openshift-machine-api  NAME DESIRED CURRENT READY AVAILABLE AGE  ash-wavelength-nyc-dvmdl-worker-us-east-1a 1 1 1 1 6d22h  ash-wavelength-nyc-dvmdl-worker-us-east-1b 1 1 1 1 6d22h  ash-wavelength-nyc-dvmdl-worker-us-east-1c 1 1 1 1 6d22h |
| --- |



We will use the existing MachineSet template and edit it to create wavelength MachineSet:

| #oc get machineset ash-wavelength-nyc-dvmdl-worker-us-east-1a -n openshift-machine-api -o yaml > wl-machineset.yaml  #vi wl-machine.yaml |
| --- |



In the yaml file, you will need to edit following attributes:

| apiVersion: machine.openshift.io/v1beta1  kind: MachineSet  metadata:  labels:  machine.openshift.io/cluster-api-cluster: <infrastructureID> 1  name: <infrastructureID>-<role>-<zone> 2  namespace: openshift-machine-api  spec:  replicas: 1  selector:  matchLabels:  machine.openshift.io/cluster-api-cluster: <infrastructureID> 1  machine.openshift.io/cluster-api-machineset: <infrastructureID>-<role>-<zone> 2  template:  metadata:  labels:  machine.openshift.io/cluster-api-cluster: <infrastructureID> 1  machine.openshift.io/cluster-api-machine-role: <role> 3  machine.openshift.io/cluster-api-machine-type: <role> 3  machine.openshift.io/cluster-api-machineset: <infrastructureID>-<role>-<zone> 2  spec:  metadata:  labels:  node-role.kubernetes.io/<role>: ""  providerSpec:  value:  ami:  id: ami-04a16d506e5b0e246  apiVersion: awsproviderconfig.openshift.io/v1beta1  blockDevices:  - ebs:  iops: 4000  volumeSize: 500  volumeType: gp2  credentialsSecret:  name: aws-cloud-credentials  deviceIndex: 0  iamInstanceProfile:  id: <infrastructureID>-worker-profile 1  instanceType: t3.xlarge 4  kind: AWSMachineProviderConfig  placement:  availabilityZone: us-east-1-wl1-nyc-wlz-1 5  region: us-east-1  securityGroups:  - filters:  - name: tag:Name  values:  - <infrastructureID>-worker-sg 1  subnet:  id: <subnet-id> 6  tags:  - name: kubernetes.io/cluster/<infrastructureID>  value: owned  userDataSecret:  name: worker-user-data |
| --- |



1. Specify the infrastructure ID that is based on the cluster ID that you set when we provisioned the cluster.**Note If you have copied the MachineSet from an existing AZ, then you don't have to modify the infrastructure ID or cluster ID.**

# oc get -o jsonpath='{.status.infrastructureName}{"\n"}' infrastructure cluster

1. Specify the infrastructure ID, node label, and zone
2. Specify the node label to add
3. Instance type has to be t3.xlarge because Wavelength supports the following instance types for edge workloads: t3.medium, t3.xlarge, and r5.2xlarge. **Note, we have tested only with t3.xlarge . We believe r5.2xlarge should also work fine!**.
4. Availability zone has to wavelength zone
5. Subnet ID of the subnet created for wavelength zone

## Testing Communication

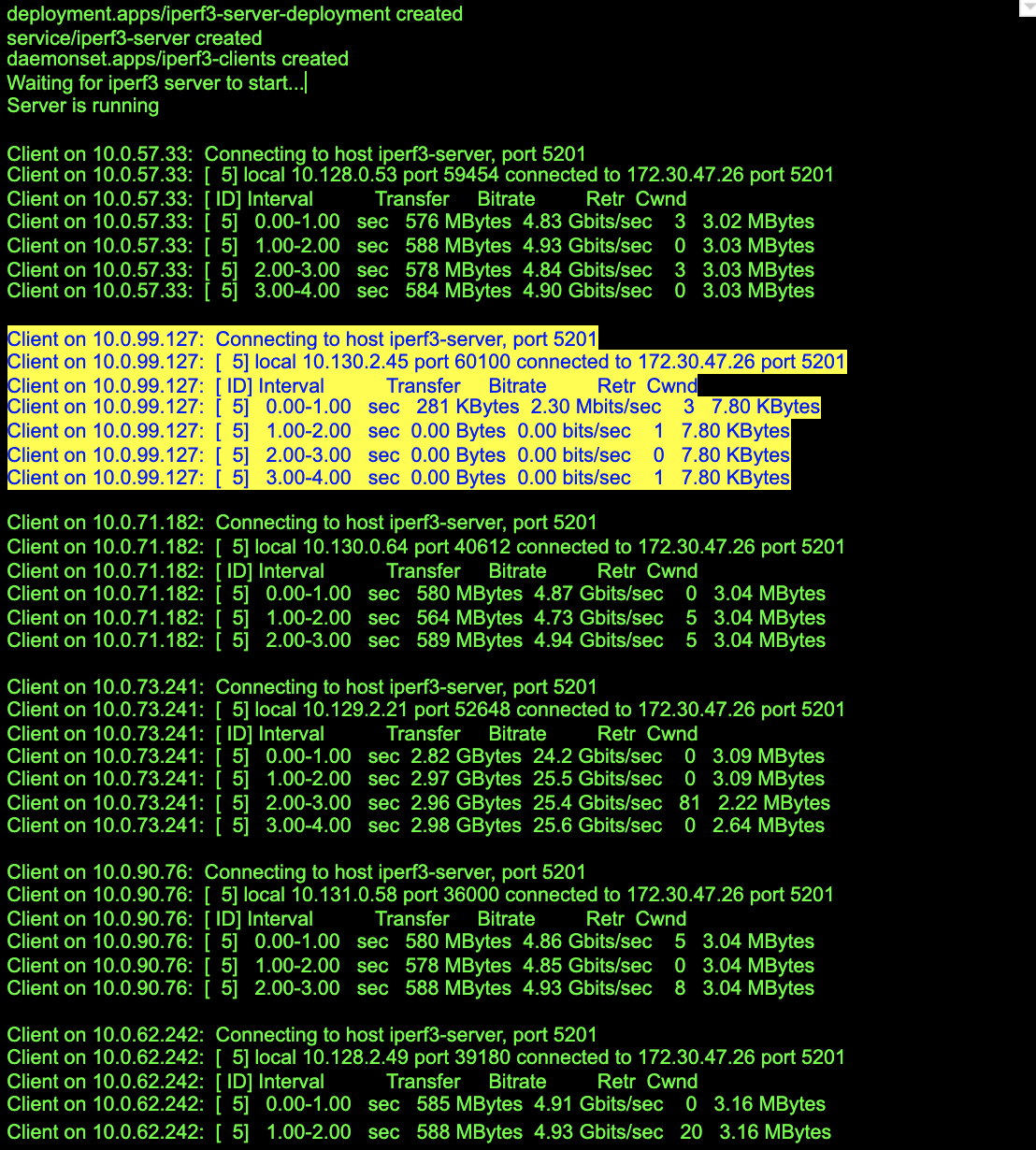
Now we will test the communication between pods running on a wavelength node and a normal node. We can use iperf (<https://github.com/Pharb/kubernetes-iperf3>) for this. To run this test, clone the above repository, login to OpenShift cluster, and run the following command:

| #sh iperf3.sh |
| --- |



The output should look similar to this:

| deployment.apps/iperf3-server-deployment created  service/iperf3-server created  daemonset.apps/iperf3-clients created  Waiting for iperf3 server to start...  Server is running  Client on 10.0.57.33: Connecting to host iperf3-server, port 5201  Client on 10.0.57.33: [ 5] local 10.128.0.53 port 59454 connected to 172.30.47.26 port 5201  Client on 10.0.57.33: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.57.33: [ 5] 0.00-1.00 sec 576 MBytes 4.83 Gbits/sec 3 3.02 MBytes  Client on 10.0.57.33: [ 5] 1.00-2.00 sec 588 MBytes 4.93 Gbits/sec 0 3.03 MBytes  Client on 10.0.57.33: [ 5] 2.00-3.00 sec 578 MBytes 4.84 Gbits/sec 3 3.03 MBytes  Client on 10.0.57.33: [ 5] 3.00-4.00 sec 584 MBytes 4.90 Gbits/sec 0 3.03 MBytes  Client on 10.0.99.127: Connecting to host iperf3-server, port 5201  Client on 10.0.99.127: [ 5] local 10.130.2.45 port 60100 connected to 172.30.47.26 port 5201  Client on 10.0.99.127: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.99.127: [ 5] 0.00-1.00 sec 281 KBytes 2.30 Mbits/sec 3 7.80 KBytes  Client on 10.0.99.127: [ 5] 1.00-2.00 sec 0.00 Bytes 0.00 bits/sec 1 7.80 KBytes  Client on 10.0.99.127: [ 5] 2.00-3.00 sec 0.00 Bytes 0.00 bits/sec 0 7.80 KBytes  Client on 10.0.99.127: [ 5] 3.00-4.00 sec 0.00 Bytes 0.00 bits/sec 1 7.80 KBytes  Client on 10.0.71.182: Connecting to host iperf3-server, port 5201  Client on 10.0.71.182: [ 5] local 10.130.0.64 port 40612 connected to 172.30.47.26 port 5201  Client on 10.0.71.182: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.71.182: [ 5] 0.00-1.00 sec 580 MBytes 4.87 Gbits/sec 0 3.04 MBytes  Client on 10.0.71.182: [ 5] 1.00-2.00 sec 564 MBytes 4.73 Gbits/sec 5 3.04 MBytes  Client on 10.0.71.182: [ 5] 2.00-3.00 sec 589 MBytes 4.94 Gbits/sec 5 3.04 MBytes  Client on 10.0.73.241: Connecting to host iperf3-server, port 5201  Client on 10.0.73.241: [ 5] local 10.129.2.21 port 52648 connected to 172.30.47.26 port 5201  Client on 10.0.73.241: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.73.241: [ 5] 0.00-1.00 sec 2.82 GBytes 24.2 Gbits/sec 0 3.09 MBytes  Client on 10.0.73.241: [ 5] 1.00-2.00 sec 2.97 GBytes 25.5 Gbits/sec 0 3.09 MBytes  Client on 10.0.73.241: [ 5] 2.00-3.00 sec 2.96 GBytes 25.4 Gbits/sec 81 2.22 MBytes  Client on 10.0.73.241: [ 5] 3.00-4.00 sec 2.98 GBytes 25.6 Gbits/sec 0 2.64 MBytes  Client on 10.0.90.76: Connecting to host iperf3-server, port 5201  Client on 10.0.90.76: [ 5] local 10.131.0.58 port 36000 connected to 172.30.47.26 port 5201  Client on 10.0.90.76: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.90.76: [ 5] 0.00-1.00 sec 580 MBytes 4.86 Gbits/sec 5 3.04 MBytes  Client on 10.0.90.76: [ 5] 1.00-2.00 sec 578 MBytes 4.85 Gbits/sec 0 3.04 MBytes  Client on 10.0.90.76: [ 5] 2.00-3.00 sec 588 MBytes 4.93 Gbits/sec 8 3.04 MBytes  Client on 10.0.62.242: Connecting to host iperf3-server, port 5201  Client on 10.0.62.242: [ 5] local 10.128.2.49 port 39180 connected to 172.30.47.26 port 5201  Client on 10.0.62.242: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.62.242: [ 5] 0.00-1.00 sec 585 MBytes 4.91 Gbits/sec 0 3.16 MBytes  Client on 10.0.62.242: [ 5] 1.00-2.00 sec 588 MBytes 4.93 Gbits/sec 20 3.16 MBytes |
| --- |

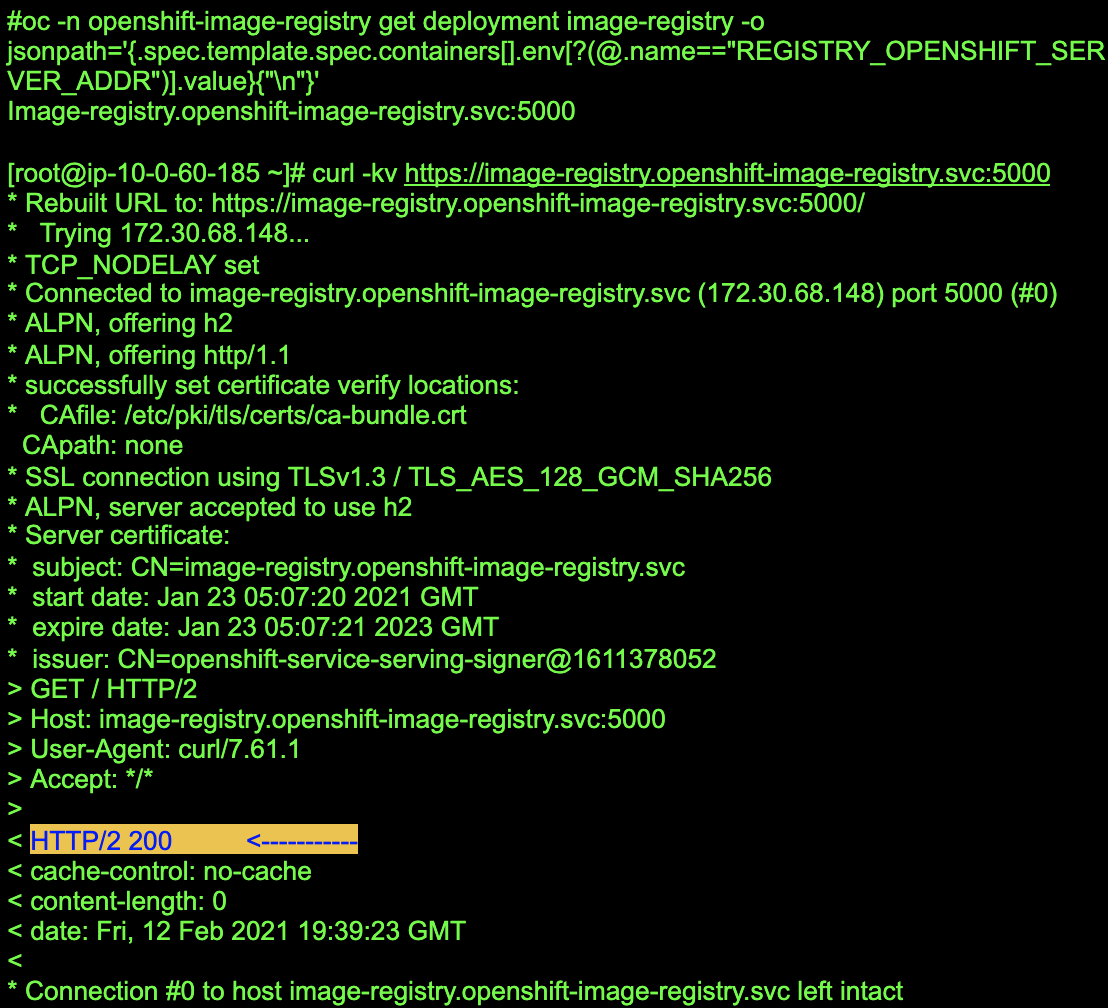


The highlighted result is of a wavelength node. We can observe that the pods in the wavelength zone are unable to communicate with other pods running in normal zones. The reason behind this is MTU. The default MTU in wavelength is 1300 whereas in a normal zone, it is 1500.When a packet is larger than the MTU size that is transmitted over HTTP, the physical network router is able to break the packet into multiple packets to transmit the data. However, when a packet is larger than the MTU size that is transmitted over HTTPS, the router is forced to drop the packet.

## Verifying Maximum Transmission Unit (MTU) Issue

To verify the MTU issue, we can check by following this procedure. We can try to connect to the internal URL of an application running on OpenShift. In this case, we can use the internal registry for communication.

| #oc -n openshift-image-registry get deployment image-registry -o jsonpath='{.spec.template.spec.containers[].env[?(@.name=="REGISTRY\_OPENSHIFT\_SERVER\_ADDR")].value}{"\n"}'  Image-registry.openshift-image-registry.svc:5000  [root@ip-10-0-60-185 ~]# curl -kv <https://image-registry.openshift-image-registry.svc:5000>  \* Rebuilt URL to: https://image-registry.openshift-image-registry.svc:5000/  \* Trying 172.30.68.148...  \* TCP\_NODELAY set  \* Connected to image-registry.openshift-image-registry.svc (172.30.68.148) port 5000 (#0)  \* ALPN, offering h2  \* ALPN, offering http/1.1  \* successfully set certificate verify locations:  \* CAfile: /etc/pki/tls/certs/ca-bundle.crt  CApath: none  \* SSL connection using TLSv1.3 / TLS\_AES\_128\_GCM\_SHA256  \* ALPN, server accepted to use h2  \* Server certificate:  \* subject: CN=image-registry.openshift-image-registry.svc  \* start date: Jan 23 05:07:20 2021 GMT  \* expire date: Jan 23 05:07:21 2023 GMT  \* issuer: CN=openshift-service-serving-signer@1611378052  > GET / HTTP/2  > Host: image-registry.openshift-image-registry.svc:5000  > User-Agent: curl/7.61.1  > Accept: \*/\*  >  < HTTP/2 200 <-----------  < cache-control: no-cache  < content-length: 0  < date: Fri, 12 Feb 2021 19:39:23 GMT  <  \* Connection #0 to host image-registry.openshift-image-registry.svc left intact |
| --- |



| #oc -n openshift-image-registry get deployment image-registry -o jsonpath='{.spec.template.spec.containers[].env[?(@.name=="REGISTRY\_OPENSHIFT\_SERVER\_ADDR")].value}{"\n"}'  Image-registry.openshift-image-registry.svc:5000  [root@ip-10-0-99-126~]# curl -kv https://image-registry.openshift-image-registry.svc:5000  \* Rebuilt URL to: https://image-registry.openshift-image-registry.svc:5000/  \* Trying 172.30.68.148...  \* Connected to image-registry.openshift-image-registry.svc (172.30.68.148) port 5000 (#0)  \* successfully set certificate verify locations:  \* CAfile: /etc/pki/tls/certs/ca-bundle.crt  CApath: none  \* SSL connection using TLSv1.3 / TLS\_AES\_128\_GCM\_SHA256  \* ALPN, server accepted to use h2  \* Server certificate:  \* subject: CN=image-registry.openshift-image-registry.svc  \* start date: Jan 23 05:07:20 2021 GMT  \* expire date: Jan 23 05:07:21 2023 GMT  \* issuer: CN=openshift-service-serving-signer@1611378052  > GET / HTTP/2  > Host: image-registry.openshift-image-registry.svc:5000  > User-Agent: curl/7.61.1  > Accept: \*/\*  …..  …..  ….. |
| --- |



## Changing MTU for OpenShift SDN

To overcome the MTU issue, we have to change the MTU size of the SDN. This way, the overall MTU size is not more than 1300, which is the MTU size for nodes in the wavelength zone. The procedure to change MTU for OpenShift SDN is as follows:

* The SDN MTU can be configured in the object cluster/networks.operator.openshift.io, and the actual configuration can be retrieved by:

| oc get networks.operator.openshift.io cluster -o yaml |
| --- |



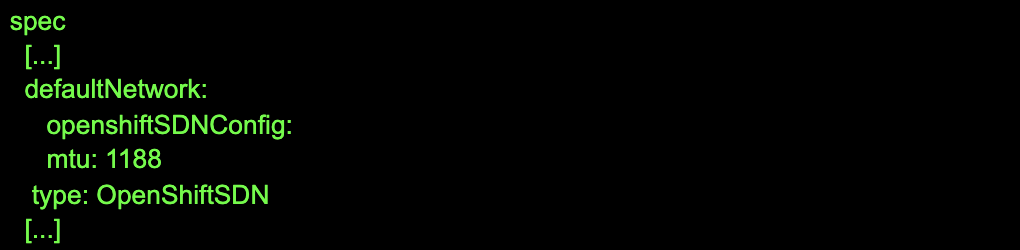
* Edit the network object to configure the new MTU:

| oc edit networks.operator.openshift.io cluster |
| --- |



* Move to the key spec.defaultNetwork and configure it with the desired MTU and save the changes:

| spec  [...]  defaultNetwork:  openshiftSDNConfig:  mtu: 1188  type: OpenShiftSDN  [...] |
| --- |



* The network operator maintains a configmap with the current network configuration applied to the cluster. Delete it so the operator re-reads the modified network configuration and recreates the configuration:

| oc -n openshift-network-operator delete configmap applied-cluster |
| --- |



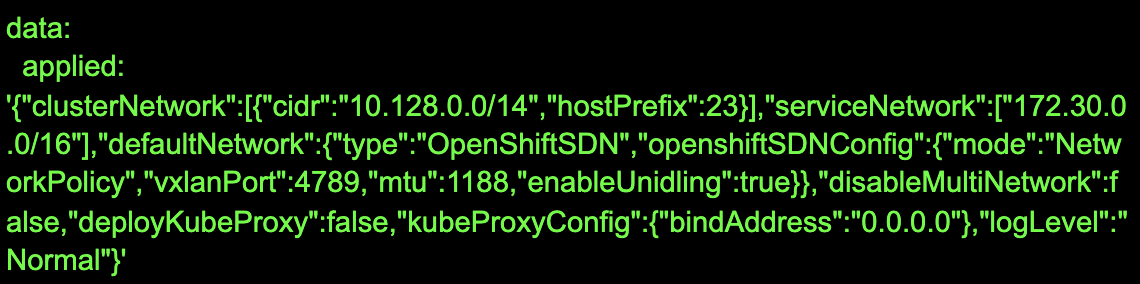
* Check that the configmap is created with the new MTU:

| oc -n openshift-network-operator get configmap applied-cluster -o yaml |
| --- |



* The object data should look like the following (notice the field ‘mtu: 1188’):

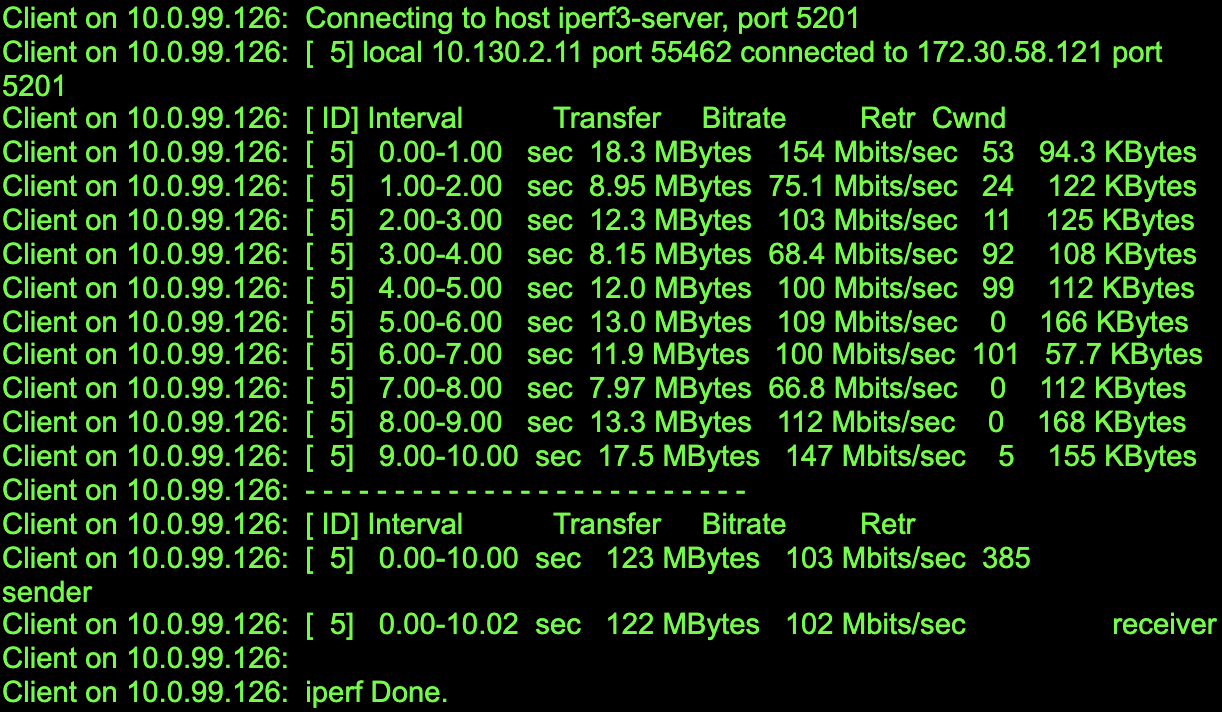
| data:  applied: '{"clusterNetwork":[{"cidr":"10.128.0.0/14","hostPrefix":23}],"serviceNetwork":["172.30.0.0/16"],"defaultNetwork":{"type":"OpenShiftSDN","openshiftSDNConfig":{"mode":"NetworkPolicy","vxlanPort":4789,"mtu":1188,"enableUnidling":true}},"disableMultiNetwork":false,"deployKubeProxy":false,"kubeProxyConfig":{"bindAddress":"0.0.0.0"},"logLevel":"Normal"}' |
| --- |



## Check Connectivity After MTU Change

We will use iperf again to test the communication between pods:

| Client on 10.0.99.126: Connecting to host iperf3-server, port 5201  Client on 10.0.99.126: [ 5] local 10.130.2.11 port 55462 connected to 172.30.58.121 port 5201  Client on 10.0.99.126: [ ID] Interval Transfer Bitrate Retr Cwnd  Client on 10.0.99.126: [ 5] 0.00-1.00 sec 18.3 MBytes 154 Mbits/sec 53 94.3 KBytes  Client on 10.0.99.126: [ 5] 1.00-2.00 sec 8.95 MBytes 75.1 Mbits/sec 24 122 KBytes  Client on 10.0.99.126: [ 5] 2.00-3.00 sec 12.3 MBytes 103 Mbits/sec 11 125 KBytes  Client on 10.0.99.126: [ 5] 3.00-4.00 sec 8.15 MBytes 68.4 Mbits/sec 92 108 KBytes  Client on 10.0.99.126: [ 5] 4.00-5.00 sec 12.0 MBytes 100 Mbits/sec 99 112 KBytes  Client on 10.0.99.126: [ 5] 5.00-6.00 sec 13.0 MBytes 109 Mbits/sec 0 166 KBytes  Client on 10.0.99.126: [ 5] 6.00-7.00 sec 11.9 MBytes 100 Mbits/sec 101 57.7 KBytes  Client on 10.0.99.126: [ 5] 7.00-8.00 sec 7.97 MBytes 66.8 Mbits/sec 0 112 KBytes  Client on 10.0.99.126: [ 5] 8.00-9.00 sec 13.3 MBytes 112 Mbits/sec 0 168 KBytes  Client on 10.0.99.126: [ 5] 9.00-10.00 sec 17.5 MBytes 147 Mbits/sec 5 155 KBytes  Client on 10.0.99.126: - - - - - - - - - - - - - - - - - - - - - - - - -  Client on 10.0.99.126: [ ID] Interval Transfer Bitrate Retr  Client on 10.0.99.126: [ 5] 0.00-10.00 sec 123 MBytes 103 Mbits/sec 385 sender  Client on 10.0.99.126: [ 5] 0.00-10.02 sec 122 MBytes 102 Mbits/sec receiver  Client on 10.0.99.126:  Client on 10.0.99.126: iperf Done. |
| --- |



As we can see from the output, we can communicate between pods. Now we will be able to deploy applications on a worker node running in wavelength zones.

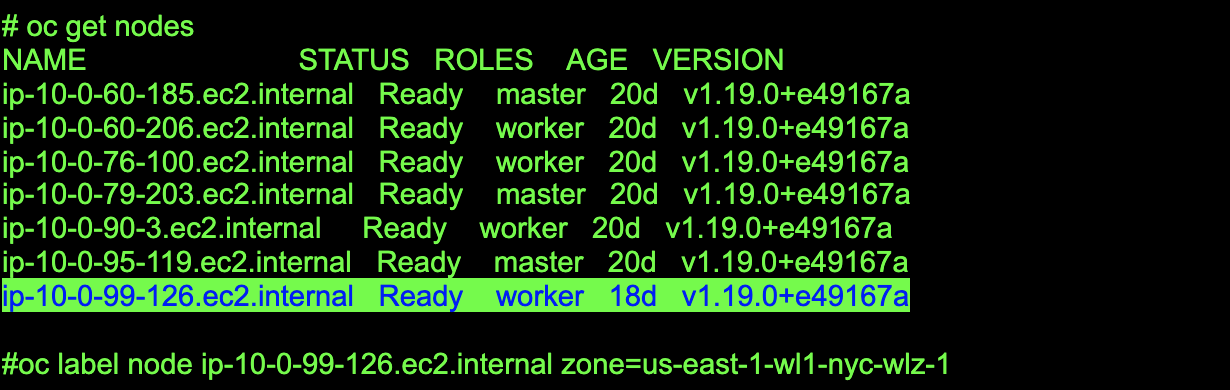
## Deploying an Application

To test further, we can deploy an application on the worker node in wavelength zone.

### Preparing wavelength node

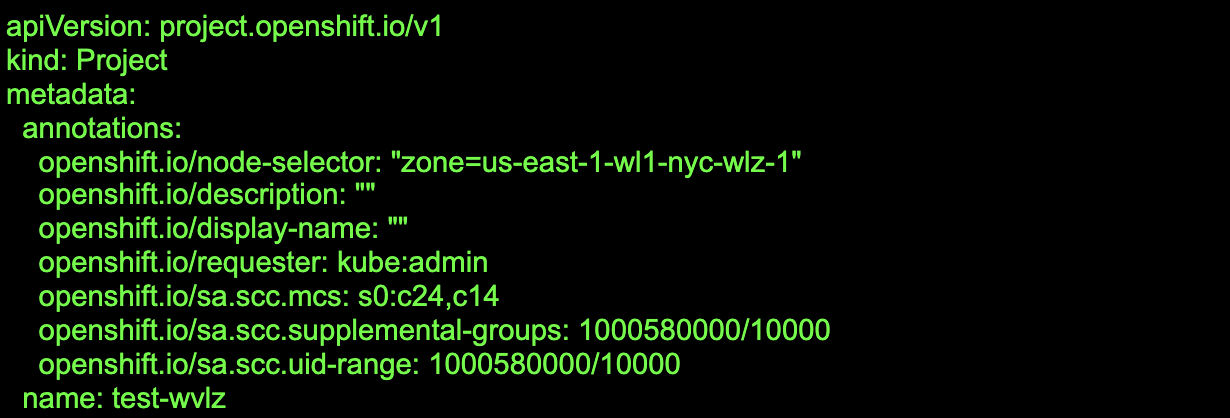
To make sure that our application intended to run on the wavelength zone only runs on nodes running in wavelength zones, we will label it:

| # oc get nodes  NAME STATUS ROLES AGE VERSION  ip-10-0-60-185.ec2.internal Ready master 20d v1.19.0+e49167a  ip-10-0-60-206.ec2.internal Ready worker 20d v1.19.0+e49167a  ip-10-0-76-100.ec2.internal Ready worker 20d v1.19.0+e49167a  ip-10-0-79-203.ec2.internal Ready master 20d v1.19.0+e49167a  ip-10-0-90-3.ec2.internal Ready worker 20d v1.19.0+e49167a  ip-10-0-95-119.ec2.internal Ready master 20d v1.19.0+e49167a  ip-10-0-99-126.ec2.internal Ready worker 18d v1.19.0+e49167a  #oc label node ip-10-0-99-126.ec2.internal zone=us-east-1-wl1-nyc-wlz-1 |
| --- |



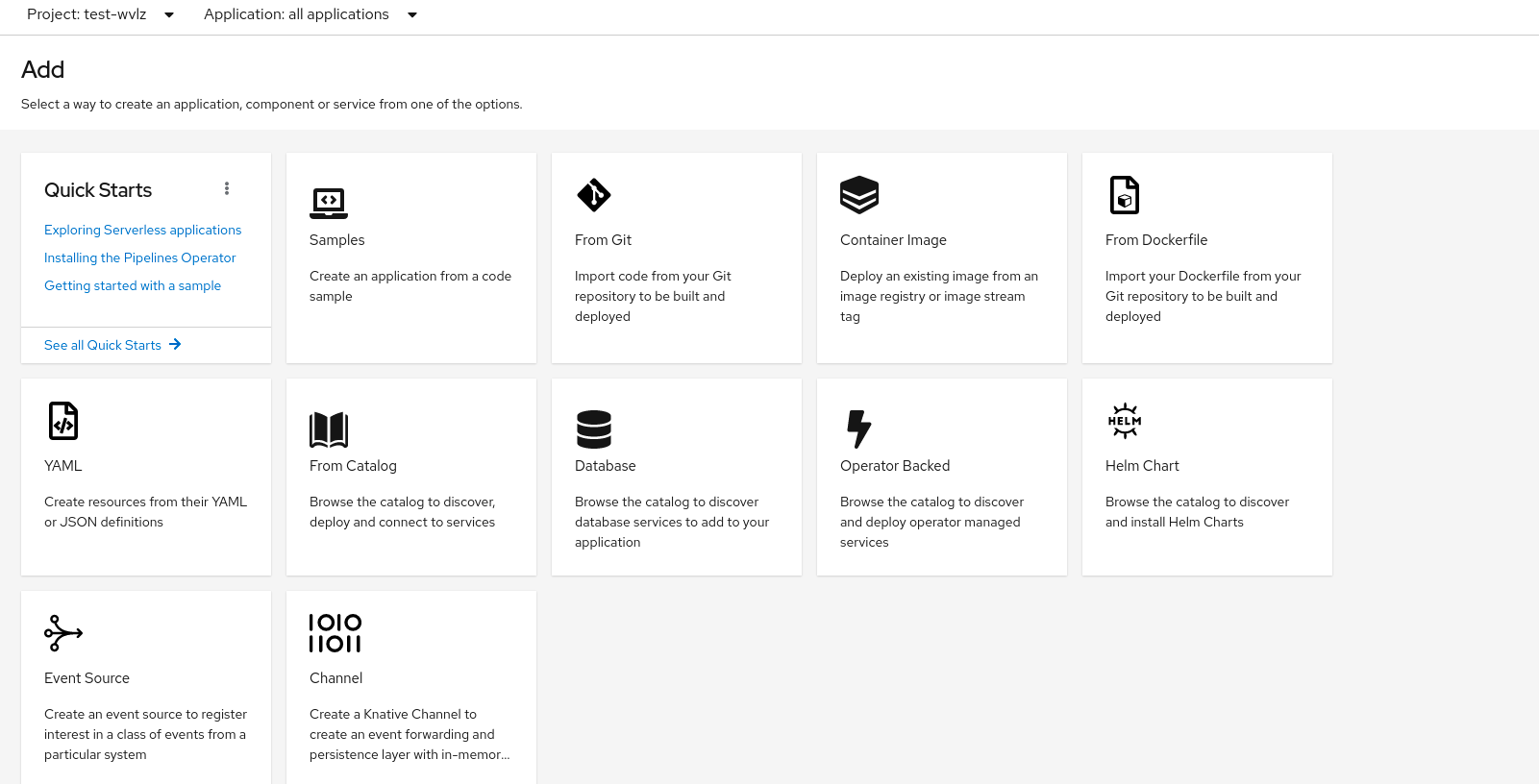
Once the node is labeled, we will create a project and annotate the node selector with the wavelength zone so that any application deployed in this project runs on a wavelength node:

| apiVersion: project.openshift.io/v1  kind: Project  metadata:  annotations:  openshift.io/node-selector: "zone=us-east-1-wl1-nyc-wlz-1"  openshift.io/description: ""  openshift.io/display-name: ""  openshift.io/requester: kube:admin  openshift.io/sa.scc.mcs: s0:c24,c14  openshift.io/sa.scc.supplemental-groups: 1000580000/10000  openshift.io/sa.scc.uid-range: 1000580000/10000  name: test-wvlz |
| --- |

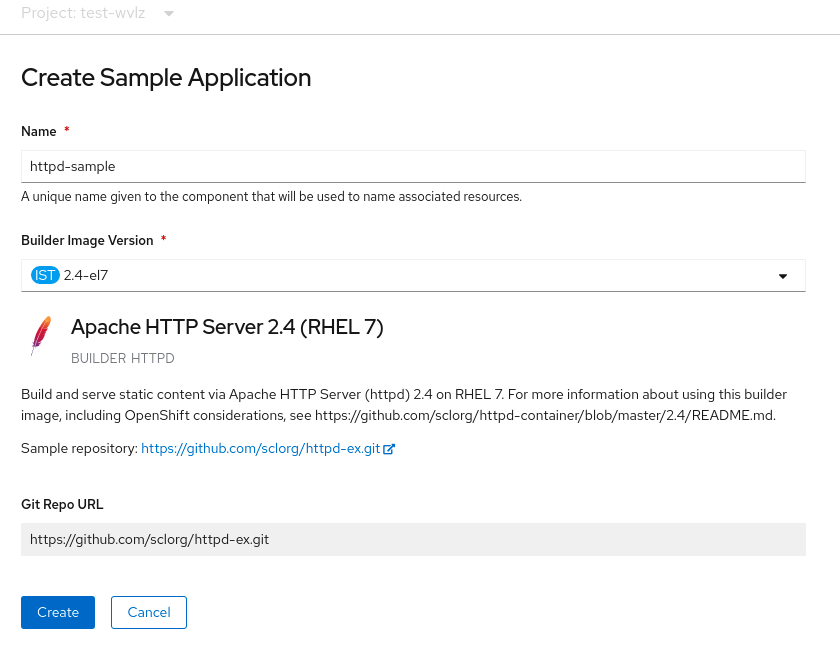


Now we will deploy a sample application in this namespace using the OpenShift web UI.

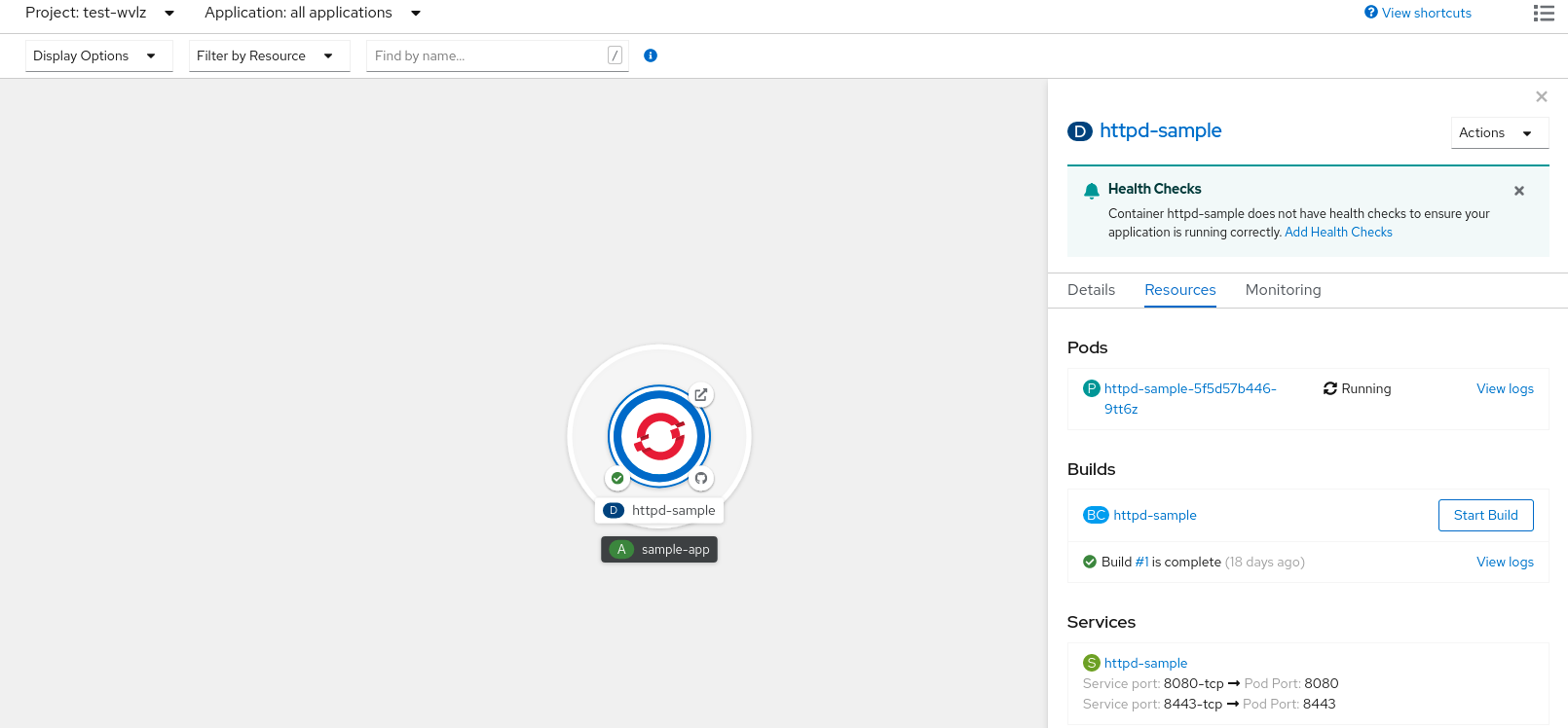
1. Go to the developer perspective of the project test-wvlz and click on add:



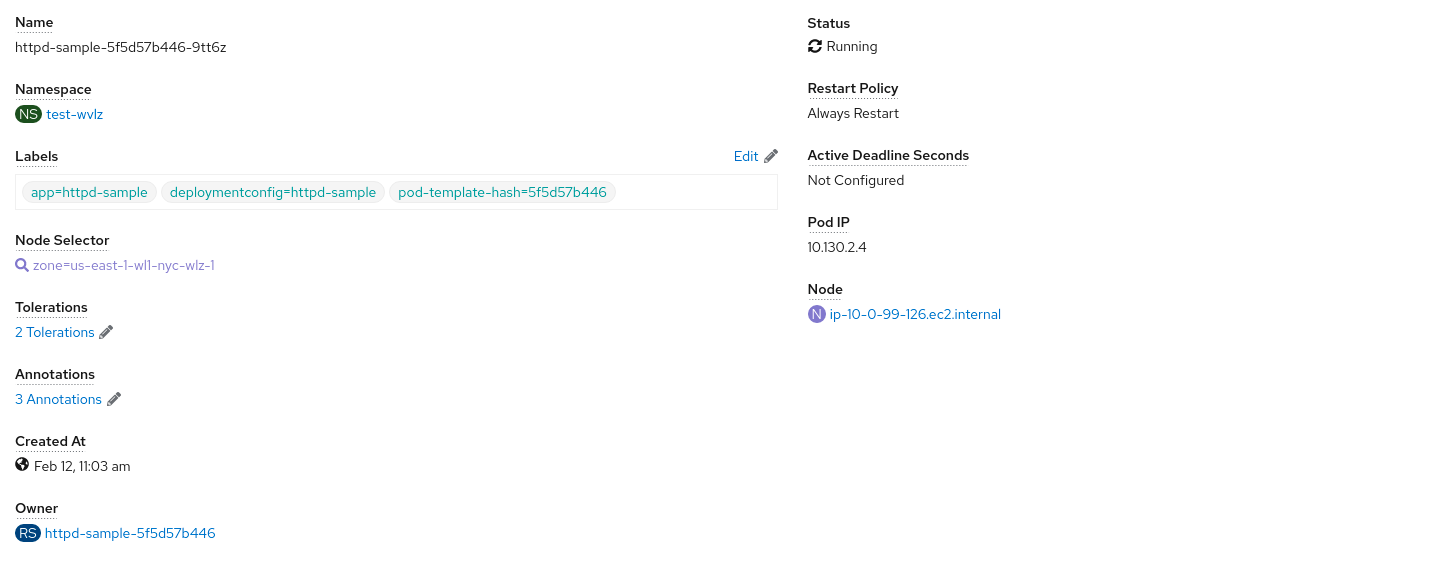
1. Click on Samples, select HTTP and click on create:



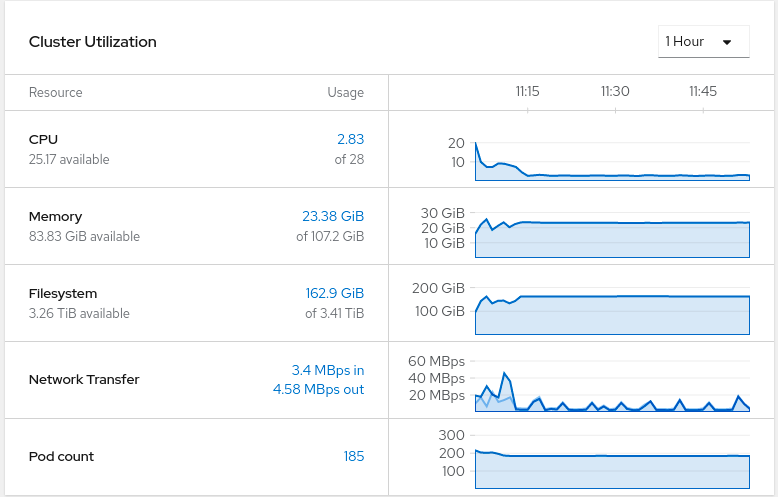
1. The application will be deployed:



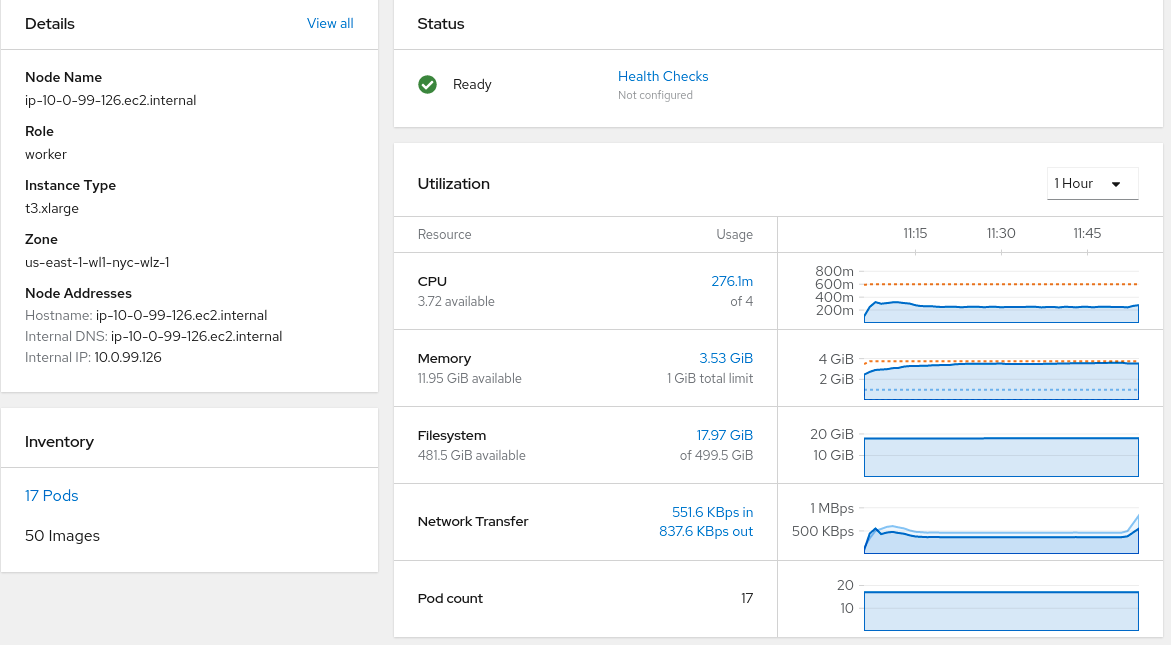
1. Click on the pod to see on which node it is running. It should be running on ip-10-0-99-126.ec2.internal, which is our wavelength node:



1. Usage details:
   1. We can also observe the resource utilization of the whole cluster:



* 1. We can also observe the resource utilization for the worker node on wavelength:



**Conclusion**

The Installer Provisioned Infrastructure deployment method can be used to install a stretched OpenShift cluster with centralized masters running in regular AWS zones (us-east-1[a,b,c]) and worker nodes running on an AWS wavelength zone (us-east-1-wl1-nyc-wlz1). This architecture significantly reduces the footprint of an OpenShift cluster in edge locations allowing edge compute infrastructure to devote more resources to applications.

Based on specific use case requirements, the deployment can be further configured to move any additional OpenShift components such as load balancers also in wavelength zones or open up node ports to allow direct communication with workloads running on wavelength zones.