Containers

Deploying Containerized Application on AWS Outposts with Amazon EKS

by Santosh Kumar and Abhishek Nanda | on 01 NOV 2021 | in Amazon Elastic Kubernetes Service, AWS Outposts, Containers, Technical How-To | Permalink | Share

AWS Outposts delivers AWS-designed infrastructure, services, A
Primary use-cases are applications that require low latency, local data processing or need to meet data residency requirements. Outpost connects back to a home Region the customer selects through a connection called the Service Link. It is operated, monitored, and managed by AWS as part of the AWS Region.

Amazon Elastic Kubernetes Service (Amazon EKS) is a managed Kubernetes service that makes it easy to run Kubernetes on AWS and on-premises. Amazon EKS worker nodes can also be deployed on Outposts. With this feature, any latency-constrained workload that needs to run in close proximity to on-premises data and applications can be deployed on Amazon EKS worker nodes launched on Outposts. This allows you to use the same familiar AWS APIs for creating the infrastructure needed for your application, provide ease of managing your application via the Kubernetes API, and at the same time, meet the low latency and data residency requirement of your on-premises workload.

Creating worker nodes on Outposts is exactly same as creating worker nodes in-Region. The difference lies only in creating the VPC subnet on Outposts. The same CreateSubnet API is used for creating a subnet on Outposts (aka Outpost Subnet) along with the OutpostArn parameter. The Availability Zone parameter of CreateSubnet API is set to the same AZ to which the Outpost connects. Any worker node launched on an Outpost subnet will land on Outposts in which the subnet was created. The connectivity of the Amazon EKS control plane, which runs in the AWS Region, and the EKS worker nodes running on Outposts is facilitated by the Outposts service link.

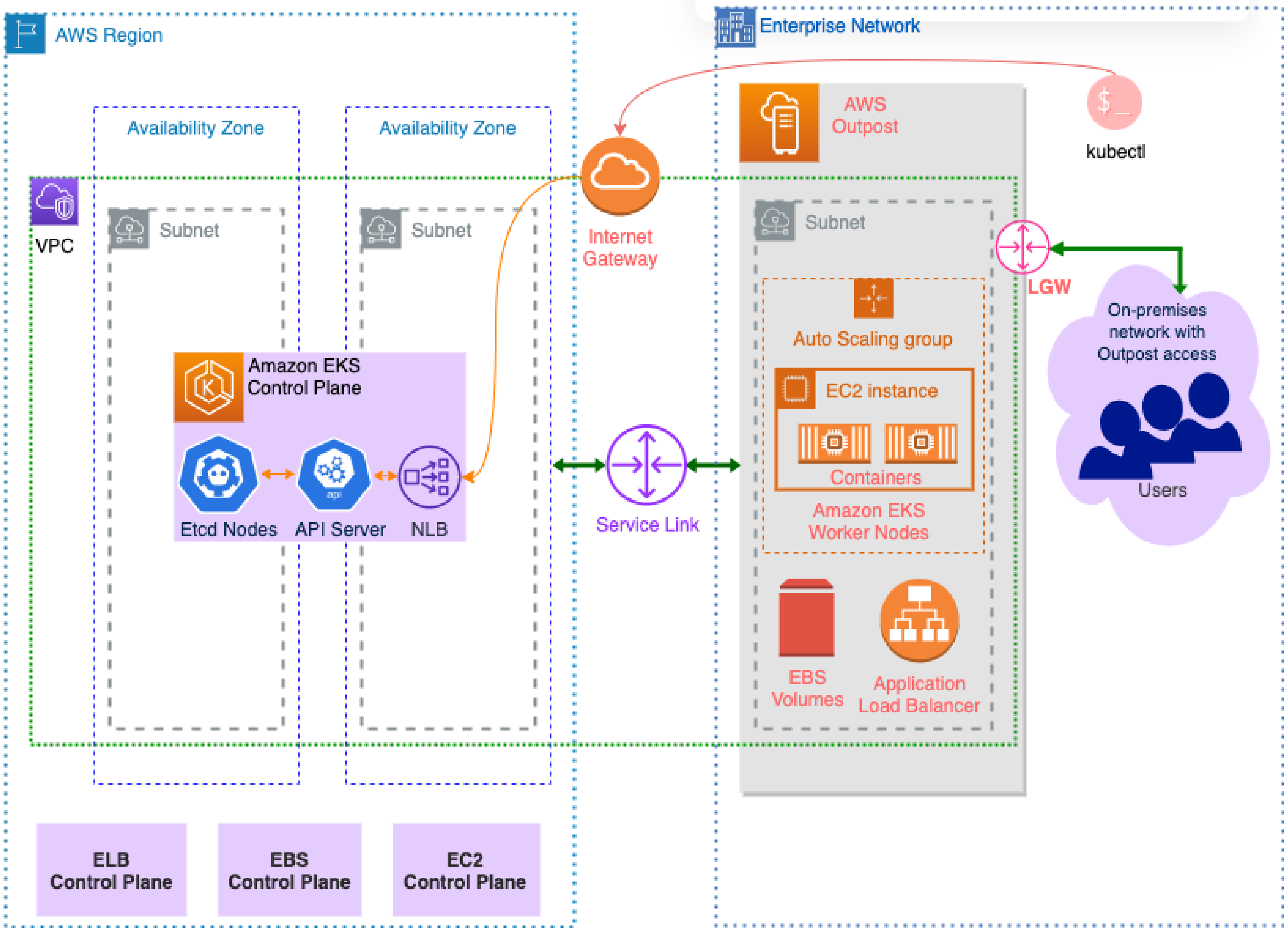
Overview

In this blog, we will walk through the process of deploying a containerized flask application on Amazon EKS hosted on an AWS Outpost. The application will be exposed to the outside network using an Application Load Balancer (ALB) on Outpost and it will serve web contents from a Kubernetes Persistent Volume (PV). We will use eksctl (the official CLI for Amazon EKS) to deploy the EKS cluster and the self-managed node group. We will use AWS CLI to create subnet on Outpost. At a high level, we will use the following steps for deploying the application:

1. Create an EKS cluster without a node group using eksctl. This will create a VPC with two public subnets and two private subnets along with the EKS cluster and other needed infrastructure.
2. Create an Amazon Elastic Container Registry (Amazon ECR) repository.
3. Create a Docker image for the flask application and push it to the ECR repository.
4. After the EKS cluster creation completes, create a new private subnet in the same VPC (created by eksctl for EKS cluster) with the Outpost ARN in order to extend the VPC to the Outpost.
5. Create a self-managed node group in the Outpost subnet using eksctl.
6. Deploy EBS CSI driver in the EKS cluster and create Persistent Volumes and Persistent Volume Claims by dynamically provisioning EBS volumes in Outpost.
7. Deploy AWS Load Balancer Controller in the EKS cluster.

8. Deploy the flask application on EKS using Kubernetes deployment and service objects.
9. Expose the Kubernetes service to the outside network using a Kubernetes ingress object.
10. In order to establish connectivity between VPC (created by eksctl) and on-premises network, configure the VPC to route on-premises CIDRs destined traffic to Outpost's Local Gateway (LGW).

The following is a high-level diagram that shows components of the application deployed by Amazon EKS on AWS Outpost:



Considerations

- We will be creating a self-managed node group using eksctl. At the time of this post, EKS managed node groups are not supported on Outposts.
- The node group spec YAML will be updated to use the subnet which we created on the Outpost in order to launch EKS worker nodes on the Outpost.
- We will be updating the storage class spec YAML to create an EBS volume of type "gp2" as by default EBS CSI driver creates a "gp3" type EBS volume, which is currently not supported with AWS Outposts.
- We will use the us-west-2 Region to deploy the EKS cluster. You must use the parent Region that your Outpost is homed to.

Prerequisites

1. An active AWS Account.
2. An installed and active AWS Outpost
3. eksctl version *0.51.0-rc.0 or later* for creating EKS cluster with 1.20 version or higher.
4. Latest AWS CLI with the IAM user having admin permission or with necessary permissions to execute the setup.
5. Helm 3.0 for applying helm charts for EBS CSI driver and AWS Load Balancer Controller.
6. kubectl version 1.20 or later
7. A workstation with Docker installed to build and push the Docker image to ECR

Solution tutorial

Create an Amazon EKS cluster and node group. Then deploy the Amazon EBS CSI driver and AWS Load Balancer Controller

Step 1: Set up environment variables

Set up the following environment variables which we will use throughout this blog. Be sure to replace the environment variables AWS Region, Outpost ID, EKS Cluster Name, the worker node instance type supported on your Outpost, and the SSH Key pair (to be used while launching worker nodes) in the following command as per your environment configuration.

Bash
```bash
# Set necessary environment variables. Be sure to modify these as per your requireme
# Parent AWS Region of your Outposts
export REGION='us-west-2'
# AZ in which EKS Cluster will be created. Must specify at-least 2 AZs.
export AZ='us-west-2b,us-west-2c'
export OUTPOST_ID="op-000111222888xxxyy"
export EKS_CLUSTER_NAME="outpost-eks"
# EC2 instance type supported on Outpost to be used for EKS Worker nodes
export OUTPOST_NG_INSTANCE_TYPE="r5.2xlarge"
# SSH Key to be used while launching EKS Worker nodes
export SSH_KEYPAIR_NAME="EKSonOutpostLabKey"
export KUBERNETES_VERSION="1.20"

export AWS_DEFAULT_REGION=$REGION
export ACCOUNTID=$(aws sts get-caller-identity --query 'Account' --output text --reg
export OUTPOST_AZ=$(aws outposts list-outposts --query "Outposts[?OutpostId==\`${OUT
export OUTPOST_ARN="arn:aws:outposts:$REGION:$ACCOUNTID:outpost/$OUTPOST_ID"
aws ec2 create-key-pair --key-name $SSH_KEYPAIR_NAME --query 'KeyMaterial' --output
```

Step 2: Create an Amazon EKS cluster using eksctl with “–without-nodegroup” option

Execute the following command to create the EKS cluster without the node groups. We will create node groups separately. It may take few min for this command to complete execution. Please be patient while eksctl finishes the cluster creation.

Note: Ensure that the VPC CIDR does not overlap with the Customer owned IP (CoIP) pool of your Outpost.

Bash
```
# Create EKS Cluster
# Be sure to modify the VPC CIDR below so that it doesn't overlap with CoIP pool
eksctl create cluster --name $EKS_CLUSTER_NAME --vpc-cidr '172.30.0.0/16' --version $K
```

Step 3: Verify the EKS cluster connectivity from your workstation

After creating the EKS cluster, eksctl creates a kubectl config file in ~/.kube or adds the new cluster's configuration within an existing config file in ~/.kube. So, once the EKS cluster transitions to ACTIVE state, we can connect to EKS cluster using kubectl. Execute the following kubectl command to verify EKS cluster connectivity. The following is the expected output.

Bash
```
$ kubectl get svc

NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.100.0.1    <none>        443/TCP   18m
```

Step 4: Retrieve the VPC ID, create Outpost Subnet, and tag it as per EKS requirements

Retrieve the VPC ID of the EKS cluster created using eksctl and define a CIDR for the Outpost subnet. Then create a subnet on Outposts using AWS CLI and tag the subnet for automatic subnet discovery by AWS Load Balancer Controller. For more information on how automatic subnet discovery works for AWS Load Balance Controller with EKS, please refer to this article.

Bash
```
VPCID=$(aws eks describe-cluster --name $EKS_CLUSTER_NAME --query cluster.resourcesVpc

CIDR=$(aws ec2 describe-vpcs --vpc-ids $VPCID --query 'Vpcs[*].CidrBlock' --output tex

SUBNET_ID=$(aws ec2 create-subnet --cidr-block $CIDR --vpc-id $VPCID --outpost-arn $OU

aws ec2 create-tags --resources $SUBNET_ID --tags Key=kubernetes.io/cluster/${EKS_CLUS
```

Step 5: Associate a new route table with the Outpost Subnet

Once the subnet has been created on the Outpost, it by default inherits the main route table. As a best practice, we should create our worker nodes in the private subnet. So, we would be updating the route table association of the Outpost subnet to use one of the private subnets' route table created by eksctl.

Bash
```
NAT_GW_ID=$(aws cloudformation describe-stack-resource --stack-name eksctl-${EKS_CLUST

ROUTE_TABLE_ID=$(aws ec2 create-route-table --vpc-id $VPCID --query 'RouteTable.RouteT

aws ec2 create-route --route-table-id $ROUTE_TABLE_ID --destination-cidr-block '0.0.0.

aws ec2 associate-route-table --subnet-id $SUBNET_ID --route-table-id $ROUTE_TABLE_ID
```

Step 6: Create an IAM policy with necessary permissions to be used with node group IAM Instance profile

In order for the AWS Load Balance Controller to work correctly, the EKS node group IAM instance profile requires IAM permission related to CoIP. Create the IAM policy with the necessary permissions to be associated with IAM instance profile of the node group.

Bash
```
cat << EOF > iamPolicy.yaml
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:GetCoipPoolUsage",
                "ec2:AllocateAddress",
                "ec2:AssociateAddress",
                "ec2:DisassociateAddress",
                "ec2:ReleaseAddress",
                "ec2:AttachVolume",
                "ec2:DetachVolume",
                "ec2:DeleteVolume",
                "ec2:CreateVolume",
                "outposts:GetOutpostInstanceTypes",
                "logs:CreateLogGroup",
```

Step 7: Create a node group using eksctl

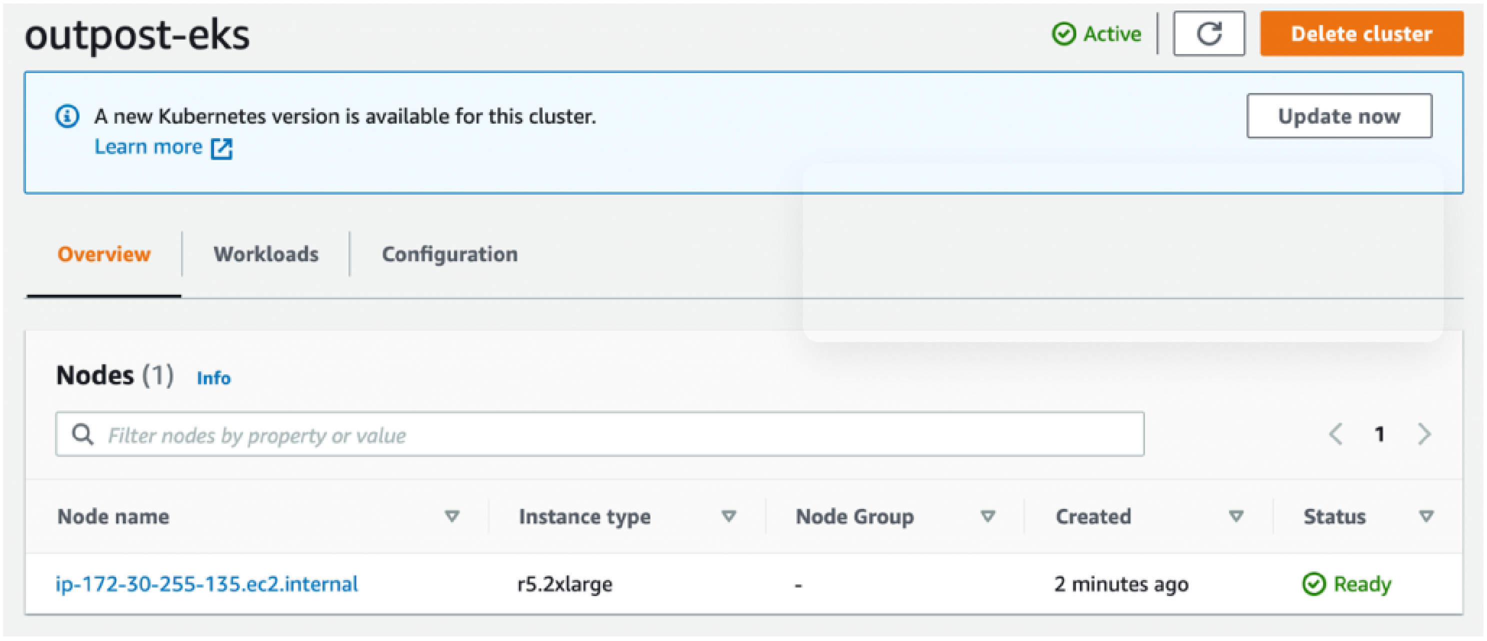
The default EBS volume type used for node group creation using eksctl is gp3, however Outpost supports only gp2 type volumes at this time, so we will explicitly specify to use gp2 volumes type for worker nodes in eksctl ClusterConfig.

Bash
```
cat << EOF > cluster-config.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: $EKS_CLUSTER_NAME
  region: $REGION
  version: "$EKS_VERSION"

vpc:
  subnets:
    private:
      outpost-one:
        id: $SUBNET_ID

nodeGroups:
  - name: outpostng1
    instanceType: $OUTPOST_NG_INSTANCE_TYPE
    desiredCapacity: 1
```

Once node group creation is successful, a worker node will be launched on your Outpost and will join the in-Region EKS cluster. You can check the nodes in the cluster using the following command:

Bash
```
$ kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
ip-172-30-255-236.us-west-2.compute.internal  Ready    <none>   18m   v1.20.4-eks-6b7
```

You can also use the AWS Management Console to check the self-managed nodes of your EKS cluster.



outpost-eks

Active | Delete cluster

A new Kubernetes version is available for this cluster. Learn more

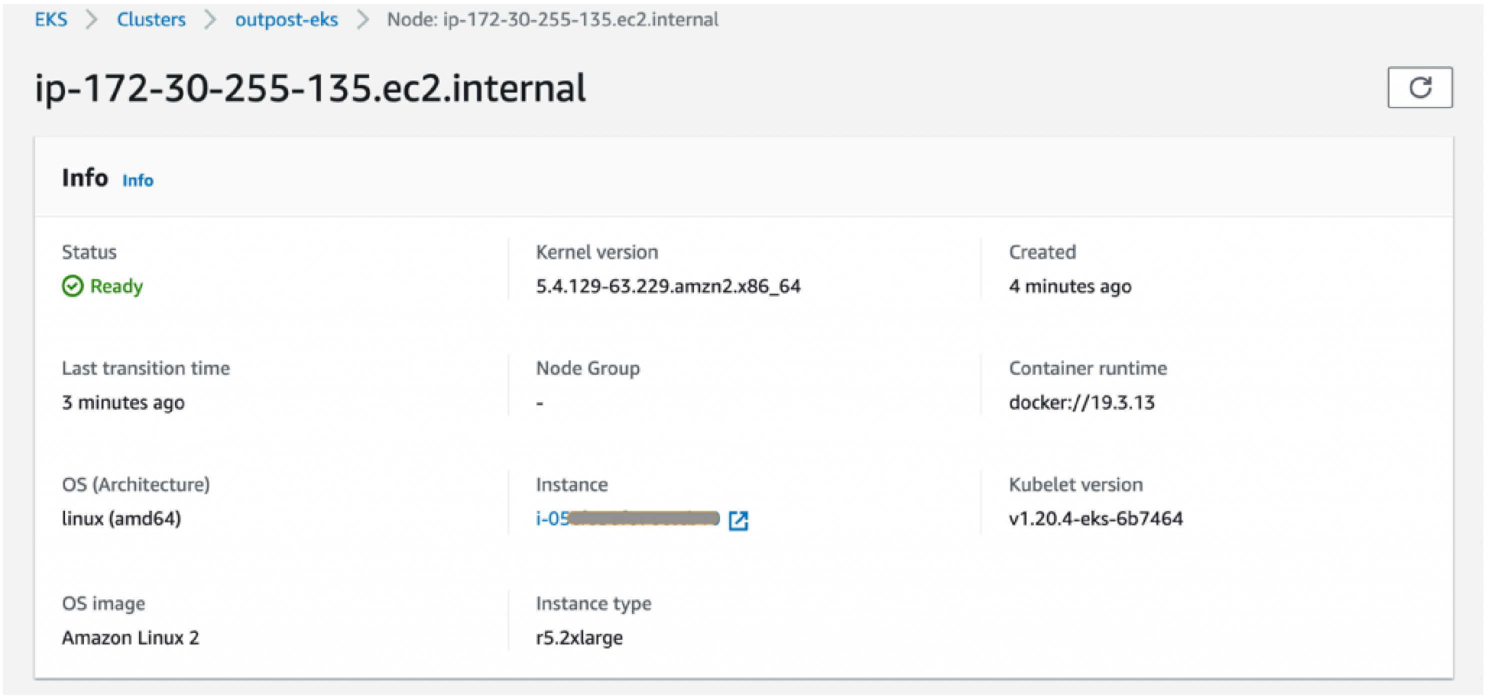
Update now

Overview | Workloads | Configuration

Nodes (1) Info

Filter nodes by property or value

Node name	Instance type	Node Group	Created	Status
ip-172-30-255-135.ec2.internal	r5.2xlarge	-	2 minutes ago	Ready



EKS > Clusters > outpost-eks > Node: ip-172-30-255-135.ec2.internal

ip-172-30-255-135.ec2.internal

Info Info

Status Ready	Kernel version 5.4.129-63.229.amzn2.x86_64	Created 4 minutes ago
Last transition time 3 minutes ago	Node Group -	Container runtime docker://19.3.13
OS (Architecture) linux (amd64)	Instance i-05[redacted]	Kubelet version v1.20.4-eks-6b7464
OS image Amazon Linux 2	Instance type r5.2xlarge	

Step 8: Deploy the Amazon EBS CSI driver

We will use a dynamically provisioned persistent volume (PV) to store the static content of the app. In order to use the Amazon Elastic Block Store (Amazon EBS) volume as a PV, the Amazon EBS CSI driver must be deployed on the EKS cluster. Follow the steps in the Amazon EBS CSI driver documentation (steps 1, 2, and 3) to deploy the Amazon EBS CSI driver to the EKS cluster. Be sure to deploy an EBS CSI driver version 0.7.0 or later as Outpost support was added in 0.7.0 version.

Note: Please replace the EKS cluster name “my-cluster” with “$EKS_CLUSTER_NAME” and Account ID “111122223333” with “$ACCOUNTID“ in step 2 while deploying the EBS CSI driver.

Step 9: Create the Storage Class and Persistent Volume Claim (PVC) to be used in the pod definition

Use the following commands to create the storage class and PVC in the EKS cluster.

Bash
```bash
cat << EOF | kubectl apply -f -
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
parameters:
  type: gp2
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
```

Step 10: Deploy AWS Load Balance Controller

In order to expose the application to external users, we will use AWS Load Balancer Controller. To deploy the AWS Load Balancer Controller in the EKS cluster, please follow the steps in AWS Load Balancer Controller documentation (steps 1, 2, 3, 5, and 6).

Note: Please replace the EKS cluster name “my-cluster” with “$EKS_CLUSTER_NAME” and Account ID “111122223333” with “$ACCOUNTID“ in step 3.

In order to ensure AWS Load Balancer Controller creates ALB in the Outpost subnet, ensure only the Outpost subnet has the necessary tags applied as mentioned in our public documentation. Use the following command to remove tags “kubernetes.io/role/elb” and “kubernetes.io/role/internal-elb“ if any, from in-Region subnets.

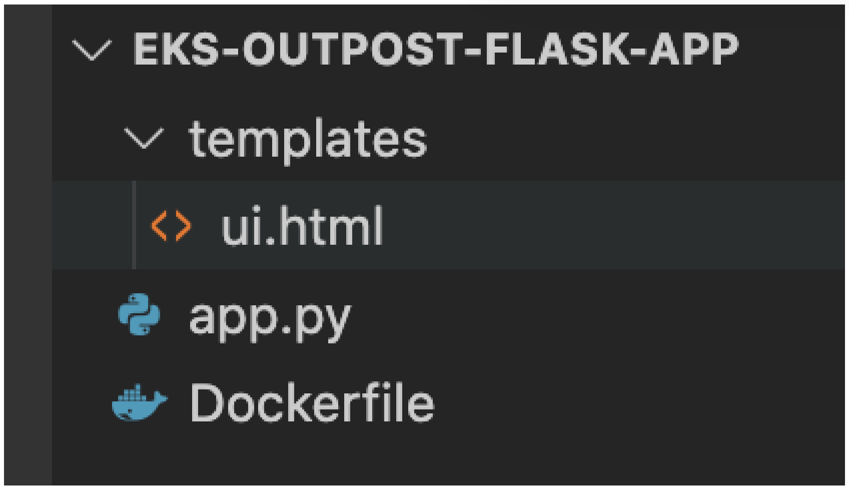
Bash
```bash
IN_REGION_SUBNETS=$(aws ec2 describe-subnets --filters Name=vpc-id,Values=$VPCID --out

for Subnet in $IN_REGION_SUBNETS; do
  aws ec2 delete-tags --resources $Subnet --tags Key='kubernetes.io/role/elb' Key='kub
done
```

Docker image creation

In order to demonstrate hosting a containerized application with EKS on Outposts, we will use a sample flask application that reads file from the Persistent Volume (EBS) and shows its content in html format in a browser. In this section, we will create a Docker image of the application and push it to an ECR repository.

Step 1: Create the directory structure for the application

We have all the necessary infrastructure components deployed to host our application, now we will create our flask application. The following is the source code directory structure for the application.



These commands can be used to create the directory structure in the current working directory:

Bash
```bash
mkdir -p EKS-Outpost-Flask-App/templates
cd EKS-Outpost-Flask-App
```

Step 2: Application source code for the app and Dockerfile

Create app.py and file in the home directory of the application, which will read a file from a predefined location on the PV and set the content in the ui.html file, which will be rendered to end users by the web server.

Bash
```bash
# Create app.py
cat << 'EOF' > app.py
import flask
import os
from datetime import datetime

directory=os.environ.get('dirLocation')
POD_NAME=os.environ.get('POD_NAME')
filePath=directory+"/"+POD_NAME+"_info.txt"
print("Dir"+directory)
print("File"+filePath)
g=""
with open(filePath) as fp:
        for cnt, line in enumerate(fp):
            g=g+line+"\n"
fp.close()
print("Content from the persistent volume:\n" + g)

app=flask.Flask(  name  )
```

Step 3: Create the Dockerfile

Create the Dockerfile to be used for creating the Docker image of the application.

Bash

```
cat << 'EOF' > Dockerfile
FROM alpine:3.14
RUN apk add python3 py-pip && \
python3 -m ensurepip && \
pip install --upgrade pip && \
pip install flask
ENV FLASK_APP app.py
WORKDIR /app
COPY . /app/
CMD ["python3", "app.py"]
EOF
```

Step 4: Create an ECR repository and login to ECR using Docker CLI

Create an ECR repository named "eks-outpost-flask-app" and then login into the ECR repository.

Bash

```
# Creating an ECR repository
aws ecr create-repository --repository-name eks-outpost-flask-app --region $REGION

# Authenticate to the ECR repository.
aws ecr get-login-password --region $REGION | docker login --username AWS --password-s
```

Step 5. Create the Docker image and push it to ECR repository

Create the Docker image using the Dockerfile, push the image to the ECR repository, and tag it with "flask".

Bash

```
docker build -t $ACCOUNTID.dkr.ecr.$REGION.amazonaws.com/eks-outpost-flask-app:flask .
docker push $ACCOUNTID.dkr.ecr.$REGION.amazonaws.com/eks-outpost-flask-app:flask
```

Deploy the application

Step 1: Create the pod definition and pod

Now we have our application and the infrastructure ready for deployment. We will use a Kubernetes deployment to deploy our application using the Docker image we created. The Kubernetes deployment will have two containers deployed per pod. One initContainer that will write the pod related details such as "PodIP", "Pod_Name", "Node_Name", and the "Pod_Namespace" to a file in the Persistent Volume. The main container will run our flask application that reads these details from the file and exposes to the end users.

Bash
```
cd ../
cat << EOF > eks-outpost-flask-app-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: eks-outpost-flask-app-deployment
  labels:
    app: flask
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask
  template:
    metadata:
        name: pod-info
        labels:
            app: flask
```

Step 2: Wait for the application to become available

Watch the pods in the default namespace and wait for the application pod to transition to a running state.

Bash
```
$ kubectl get pods --watch
NAME                                               READY   STATUS     RESTARTS   AGE
eks-outpost-flask-app-deployment-58cb5c975c-d8zbd  0/1     Init:0/1   0          3s
eks-outpost-flask-app-deployment-58cb5c975c-sff4v  0/1     Init:0/1   0          3s
eks-outpost-flask-app-deployment-58cb5c975c-d8zbd  0/1     PodInitializing   0
eks-outpost-flask-app-deployment-58cb5c975c-sff4v  0/1     PodInitializing   0
eks-outpost-flask-app-deployment-58cb5c975c-sff4v  1/1     Running           0
eks-outpost-flask-app-deployment-58cb5c975c-d8zbd  1/1     Running           0
```

Step 3: Deploy the service and ingress

As we have already deployed our application pod "pod-info" in using Kubernetes deployment, we will now create a service and an ingress to expose the application to the outside world using an external ALB.

Note: be sure to replace "ipv4pool-coip-xxxxxxxx" with the CoIP pool id associated with your Outpost. You may use the following command to describe the CoIP pool in your account and get the CoIP pool id: "aws ec2 describe-coip-pools –region $REGION".

Bash
```
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: flask-app
spec:
  selector:
    app: flask
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: NodePort
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-app-ingress
```

Enable communication from the on-premises network

After deploying the Kubernetes service and the ingress objects in the previous section, our flask application is running in the EKS cluster and can be accessed from within the VPC network. In order to access it from the on-premises network, you need to associate the eksctl VPC with the LGW route table of the Outpost and update the route table of Outpost subnet to route on-premises CIDRs destined traffic to LGW.

Note: Replace "<On-premises CIDR>" in the following command with the CIDR range of the on-premises network from where the flask application will be accessed.

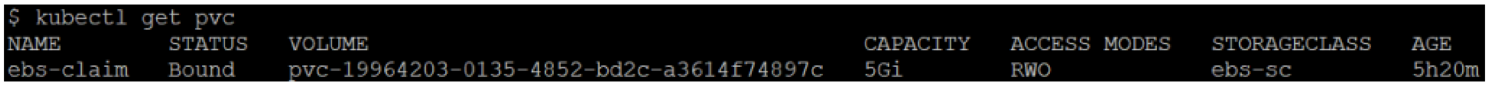
Bash
```
# Get LGW ID for the Outpost
LGW_ID=$(aws ec2 describe-local-gateways --query "LocalGateways[?OutpostArn==\`$OUTPOS

LGW_RT_ID=$(aws ec2 describe-local-gateway-route-tables --query "LocalGatewayRouteTabl
# Associate VPC with LGW Route Table and update the Outpost Subnet Route Table
aws ec2 create-local-gateway-route-table-vpc-association --local-gateway-route-table-i

aws ec2 create-route --route-table-id $ROUTE_TABLE_ID --destination-cidr-block <On-pre
```

Verification

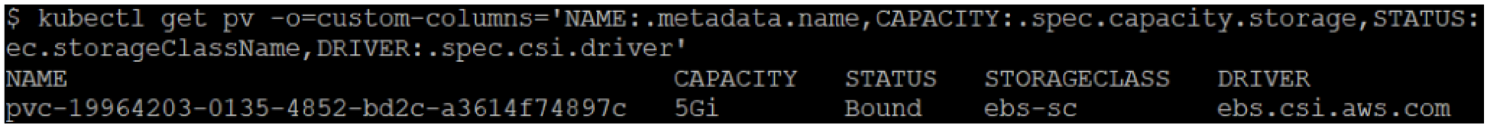
After executing the commands mentioned in preceding sections, check if the Persistent Volume was created on the Outpost using the following commands:

Bash
```bash
kubectl get pvc

kubectl get pv -o=custom-columns='NAME:.metadata.name,CAPACITY:.spec.capacity.storage,

aws ec2 describe-volumes --volume-ids $(kubectl get pv -o jsonpath='{.items[*].spec.cs
```

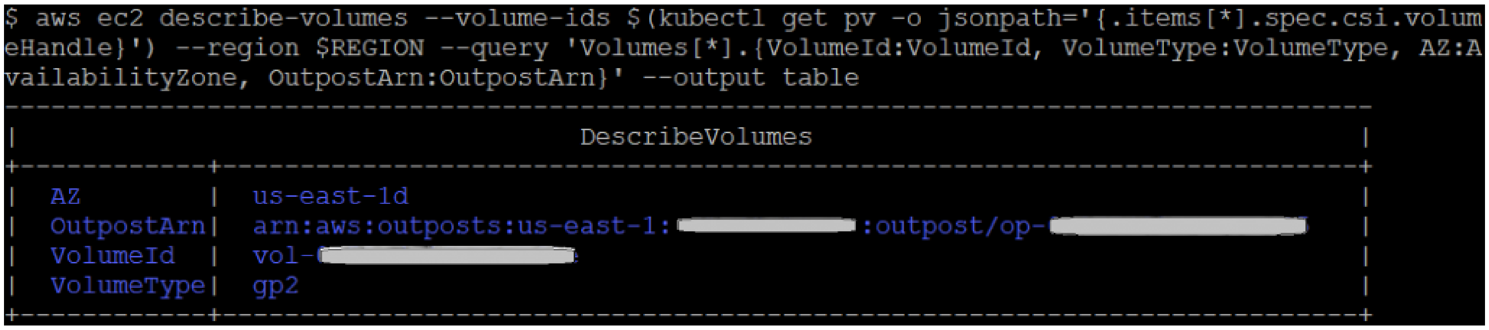
Expected output:



```
$ kubectl get pvc
NAME        STATUS    VOLUME                                      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
ebs-claim   Bound     pvc-19964203-0135-4852-bd2c-a3614f74897c   5Gi        RWO            ebs-sc         5h20m
```



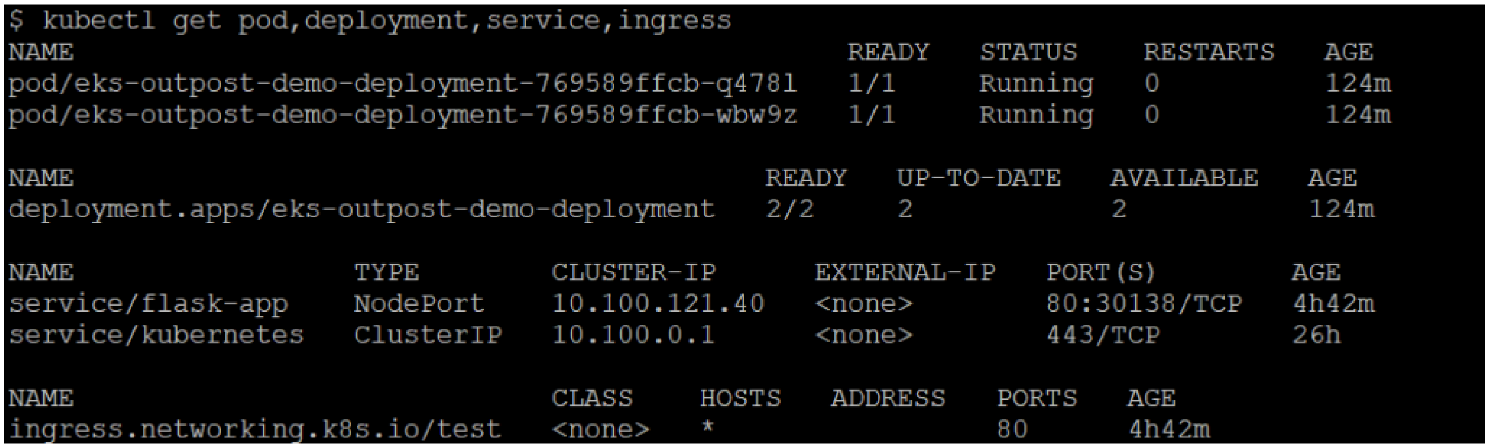
```
$ kubectl get pv -o=custom-columns='NAME:.metadata.name,CAPACITY:.spec.capacity.storage,STATUS:
ec.storageClassName,DRIVER:.spec.csi.driver'
NAME                                       CAPACITY   STATUS   STORAGECLASS   DRIVER
pvc-19964203-0135-4852-bd2c-a3614f74897c   5Gi        Bound    ebs-sc         ebs.csi.aws.com
```



```
$ aws ec2 describe-volumes --volume-ids $(kubectl get pv -o jsonpath='{.items[*].spec.csi.volum
eHandle}') --region $REGION --query 'Volumes[*].{VolumeId:VolumeId, VolumeType:VolumeType, AZ:A
vailabilityZone, OutpostArn:OutpostArn}' --output table
-----------------------------------------------------------------------------------------------
|                                       DescribeVolumes                                       |
+-------------+-------------------------------------------------------------------------------+
|  AZ         |  us-east-1d                                                                   |
|  OutpostArn |  arn:aws:outposts:us-east-1:[redacted]:outpost/op-[redacted]                  |
|  VolumeId   |  vol-[redacted]                                                               |
|  VolumeType |  gp2                                                                          |
+-------------+-------------------------------------------------------------------------------+
```

Use the following command to verify if the Kubernetes objects pods, deployment, service, and ingress was created properly:

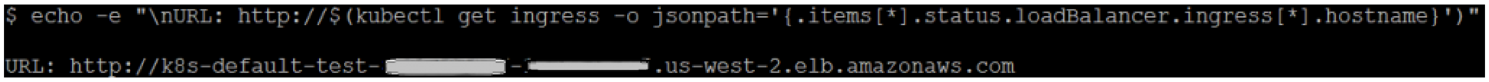
Bash
```bash
kubectl get pod,deployment,service,ingress
```



```
$ kubectl get pod,deployment,service,ingress
NAME                                              READY   STATUS    RESTARTS   AGE
pod/eks-outpost-demo-deployment-769589ffcb-q4781  1/1     Running   0          124m
pod/eks-outpost-demo-deployment-769589ffcb-wbw9z  1/1     Running   0          124m

NAME                                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/eks-outpost-demo-deployment  2/2     2            2           124m

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/flask-app      NodePort    10.100.121.40   <none>        80:30138/TCP   4h42m
service/kubernetes     ClusterIP   10.100.0.1      <none>        443/TCP        26h

NAME                              CLASS    HOSTS   ADDRESS   PORTS   AGE
ingress.networking.k8s.io/test    <none>   *                 80      4h42m
```

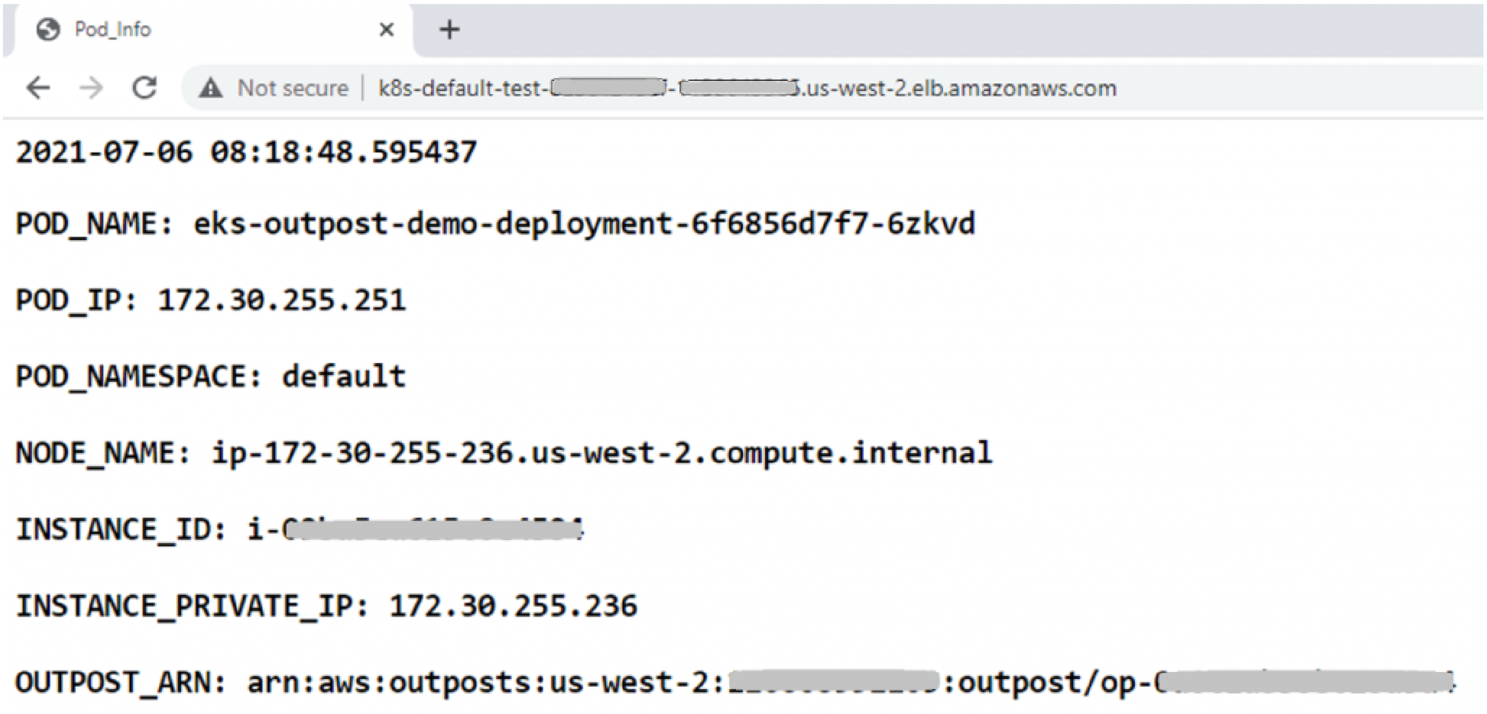
In order to check if the application has been deployed properly and working as expected, get the ALB URL using the following command and access it from a workstation in your on-premises network from where you have connectivity to the Outpost.

Bash
```bash
echo -e "\nURL: http://$(kubectl get ingress -o jsonpath='{.items[*].status.loadBalanc
```



```
$ echo -e "\nURL: http://$(kubectl get ingress -o jsonpath='{.items[*].status.loadBalancer.ingress[*].hostname}')"
URL: http://k8s-default-test-[redacted]-[redacted].us-west-2.elb.amazonaws.com
```

If all the preceding steps go as expected, you will see the application web page similar to the following screenshot:


Pod_Info
Not secure
k8s-default-test-[redacted]-[redacted].us-west-2.elb.amazonaws.com

2021-07-06 08:18:48.595437

POD_NAME: eks-outpost-demo-deployment-6f6856d7f7-6zkvd

POD_IP: 172.30.255.251

POD_NAMESPACE: default

NODE_NAME: ip-172-30-255-236.us-west-2.compute.internal

INSTANCE_ID: i-[redacted]

INSTANCE_PRIVATE_IP: 172.30.255.236

OUTPOST_ARN: arn:aws:outposts:us-west-2:[redacted]:outpost/op-[redacted]

Cleaning up

Use the following commands to cleanup resources created in this blog.

```
Bash
# Delete Kubernetes resources
kubectl delete deploy/eks-outpost-flask-app-deployment svc/flask-app ingress/flask-a

# Delete deployed helm charts
helm uninstall aws-ebs-csi-driver -n kube-system
helm uninstall aws-load-balancer-controller -n kube-system

# Delete ECR repo
aws ecr batch-delete-image --repository-name eks-outpost-flask-app --image-ids image
aws ecr delete-repository --repository-name eks-outpost-flask-app --region $REGION

# Delete NodeGroup
eksctl delete nodegroup --cluster=$EKS_CLUSTER_NAME --name outpostng1 --region $REGI

# Detach VPC from Outpost LGW Route Table
LGW_RT_A_ID=$(aws ec2 describe-local-gateway-route-table-vpc-associations --query "L
aws ec2 delete-local-gateway-route-table-vpc-association --local-gateway-route-table
```

Conclusion

AWS Outposts provides a seamless solution to host hybrid applications running on Kubernetes. The Kubernetes workloads running on Outposts can communicate with the on-premises data and systems providing ultra low-latency experience. In-region Amazon EKS control plane can manage the Kubernetes workloads hosted on EC2 instances running on Outposts. The same toolsets (such as docker, eksctl, kubectl, helm, aws-cli, AWS SDKs, and AWS Management Console) used for managing in-region cluster and worker nodes can be utilized to manage the Amazon EKS workload on AWS Outposts.

The blog demonstrates how a containerized application can be deployed on a Kubernetes cluster using simple Kubernetes deployment YAML manifest. In this post, we discussed how to deploy a flask application on AWS Outposts using Amazon EKS. The Amazon EKS cluster control plane was deployed in the region and the nodegroup was deployed on Outposts. We created an eksctl cluster config to use the Outposts subnet created using the aws-cli and created our nodegroup to launch worker nodes on Outposts. We used the EBS CSI driver and AWS Load Balancer Controller to showcase the functionalities of Persistent Volumes, Persistent Volume Claims, and exposing the sample application via Application Load Balancer using ingress YAML manifest on AWS Outposts. The EBS volumes and Application Load Balancer were launched on the Outposts to facilitate low latency. We also demonstrated how the eksctl cluster config YAML and the storageclass yaml can be tweaked to use the EBS gp2 volumes on Outposts.

TAGS: docker, EKS, IAM, Kubernetes, outposts