AWS Compute Blog

Building Modern Applications with Amazon EKS on Amazon Outposts

by Emma White | on 04 OCT 2021 | in AWS Outposts, Best Practices, Technical How-To | Permalink | Comments | Share

This post is written by Brad Kirby, Principal Outposts Specialist, and Chris Lunsford, Senior Outposts SA.

Customers are modernizing applications by deconstructing monolithic architectures and migrating application components into container–based, service-oriented, and microservices architectures. Modern applications improve scalability, reliability, and development efficiency by allowing services to be owned by smaller, more focused teams.

This post shows how you can combine Amazon Elastic Kubernetes Service (Amazon EKS) with AWS Outposts to deploy managed Kubernetes application environments on-premises alongside your existing data and applications.

For a brief introduction to the subject matter, the watch this video, which demonstrates:

- The benefits of application modernization
- How containers are an ideal enabling technology for microservices architectures
- How AWS Outposts combined with Amazon container services enables you to unwind complex service interdependencies and modernize on-premises applications with low latency, local data processing, and data residency requirements

Understanding the Amazon EKS on AWS Outposts architecture

Amazon EKS

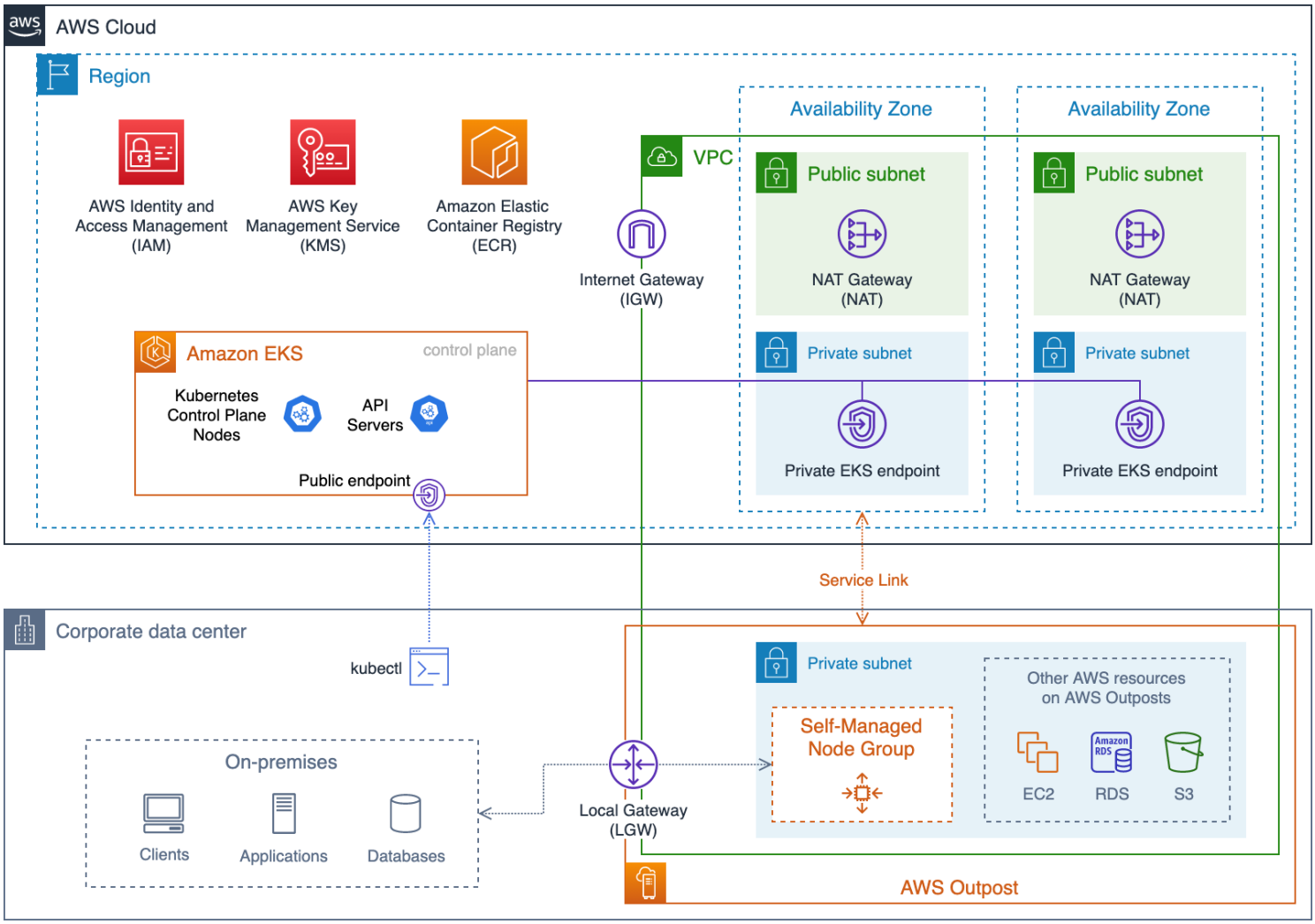
Many organizations chose Kubernetes as their container orchestration platform because of its openness, flexibility, and a growing pool of Kubernetes literate IT professionals. Amazon EKS enables you to run Kubernetes clusters on AWS without needing to install and manage the Kubernetes control plane. The control plane, including the API servers, scheduler, and cluster store services, runs within a managed environment in the AWS Region. Kubernetes clients (like kubectl) and cluster worker nodes communicate with the managed control plane via public and private EKS service endpoints.

AWS Outposts

The AWS Outposts service delivers AWS infrastructure and services to on-premises locations from the AWS Global Cloud Infrastructure. An Outpost functions as an extension of the Availability Zone (AZ) where it is anchored. AWS operates, monitors, and manages AWS Outposts infrastructure as part of its parent Region. Each Outpost connects back to its anchor AZ via a Service Link connection (a set of encrypted VPN tunnels). AWS Outposts extend Virtual Private Cloud (VPC) environments from the Region to on-premises locations and enables you to deploy VPC subnets to Outposts in your data center and co-location spaces. The Outposts Local Gateway (LGW) routes traffic between Outpost VPC subnets and the on-premises network.

Amazon EKS on AWS Outposts

You deploy Amazon EKS worker nodes to Outposts using *self-managed node groups*. The worker nodes run on Outposts and register with the Kubernetes control plane in the AWS Region. The worker nodes, and containers running on the nodes, can communicate with AWS services and resources running on the Outpost and in the region (via the Service Link) and with on-premises networks (via the Local Gateway).



You use the same AWS and Kubernetes tools and APIs to work with EKS on Outposts nodes that you use to work with EKS nodes in the Region. You can use eksctl, the AWS Management Console, AWS CLI, or infrastructure as code (IaC) tools like AWS CloudFormation or HashiCorp Terraform to create self-managed node groups on AWS Outposts.

Amazon EKS self-managed node groups on AWS Outposts

Like many customers, you might use managed node groups when you deploy EKS worker nodes in your Region, and you may be wondering, "what are self-managed node groups?"

Self-managed node groups, like managed node groups, use standard Amazon Elastic Compute Cloud (Amazon EC2) instances in EC2 Auto Scaling groups to deploy, scale-up, and scale-down EKS worker nodes using Amazon EKS optimized Amazon Machine Images (AMIs). Amazon configures the EKS optimized AMIs to work with the EKS service. The images include Docker, kubelet, and the AWS IAM Authenticator. The AMIs also contain a specialized bootstrap

script /etc/eks/bootstrap.sh that allows worker nodes to discover and connect to your cluster control plane and add Kubernetes labels that identify the nodes in the cluster.

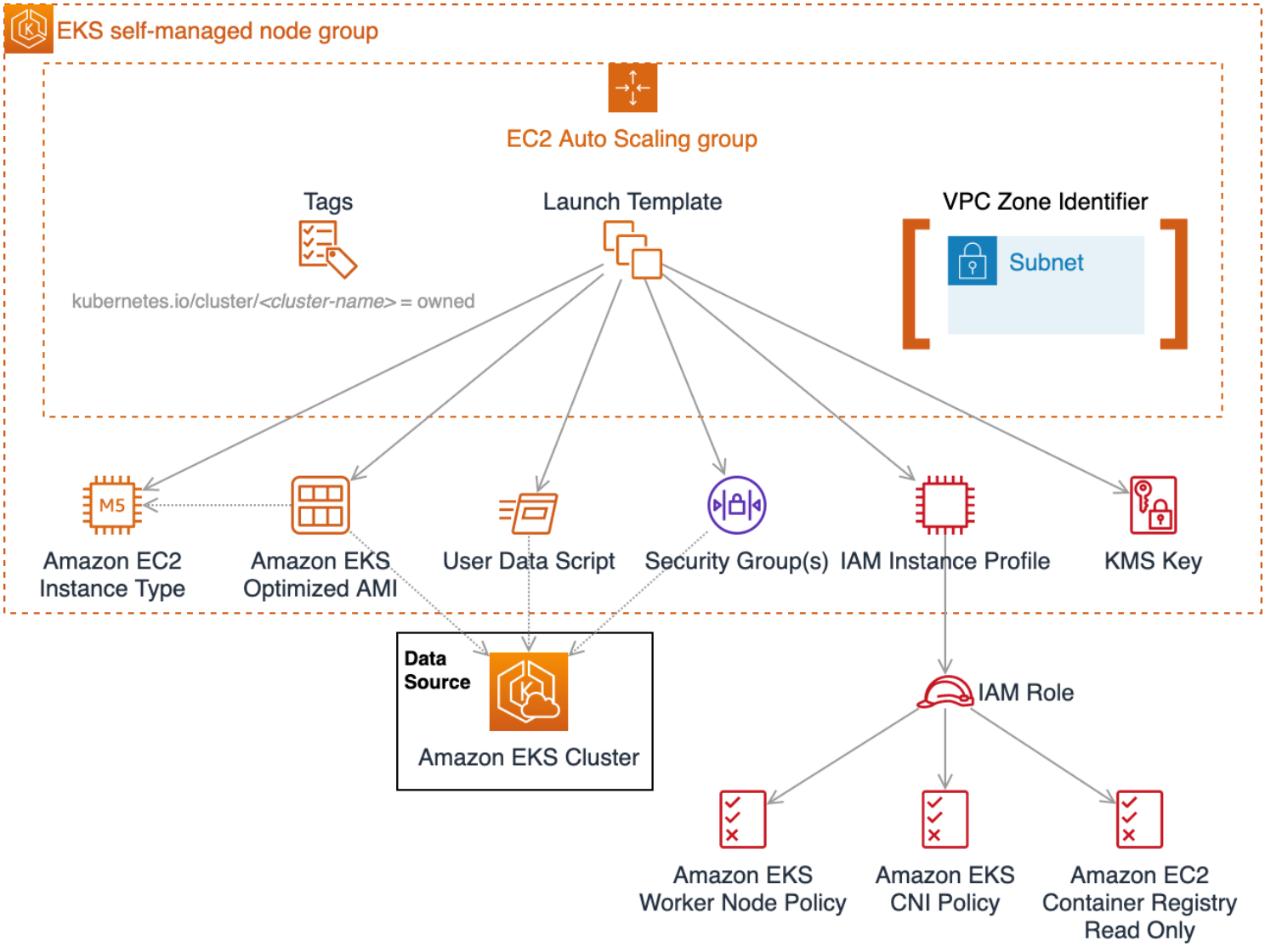
What makes the node groups *self-managed*? Managed node groups have additional features that simplify updating nodes in the node group. With self-managed nodes, you must implement a process to update or replace your node group when you want to update the nodes to use a new Amazon EKS optimized AMI.

You create self-managed node groups on AWS Outposts using the same process and resources that you use to deploy EC2 instances using EC2 Auto Scaling groups. All instances in a self-managed node group must:

- Be the same Instance type
- Be running the same Amazon Machine Image (AMI)
- Use the same Amazon EKS node IAM role
- Be tagged with the io/cluster/<cluster-name>=owned tag

Additionally, to deploy on AWS Outposts, instances must:

- Use encrypted EBS volumes
- Launch in Outposts subnets



Kubernetes authenticates all API requests to a cluster. You must configure an EKS cluster to allow nodes from a self-managed node group to join the cluster. Self-managed nodes use the *node IAM role* to authenticate with an EKS cluster. Amazon EKS clusters use the AWS IAM Authenticator for Kubernetes to authenticate requests and Kubernetes native Role Based Access Control (RBAC) to authorize requests. To enable self-managed nodes to register with a cluster, configure the AWS IAM Authenticator to recognize the node IAM role and assign the role to the system:bootstrappers and system:nodes RBAC groups.

In the following tutorial, we take you through the steps required to deploy EKS worker nodes on an Outpost and register those nodes with an Amazon EKS cluster running in the Region. We created a sample Terraform module aws-eks-self-managed-node-group to help you get started quickly. If you are interested, you can dive into the module sample code to see the detailed configurations for self-managed node groups.

Deploying Amazon EKS on AWS Outposts (Terraform)

To deploy Amazon EKS nodes on an AWS Outposts deployment, you must complete two high-level steps:

Step 1: Create a self-managed node group

Step 2: Configure Kubernetes to allow the nodes to register

We use Terraform in this tutorial to provide visibility (through the sample code) into the details of the configurations required to deploy self-managed node groups on AWS Outposts.

Prerequisites

To follow along with this tutorial, you should have the following prerequisites:

- An AWS account.
- An operational AWS Outposts deployment (Outpost) associated with your AWS account.
- AWS Command Line Interface (CLI) version 1.25 or later installed and configured on your workstation.
- HashiCorp Terraform version 14.6 or later installed on your workstation.
- Familiarity with Terraform HashiCorp Configuration Language (HCL) syntax and experience using Terraform to deploy AWS resources.
- An existing VPC that meets the requirements for an Amazon EKS cluster. For more information, see Cluster VPC considerations. You can use the Getting started with Amazon EKS guide to walk you through creating a VPC that meets the requirements.
- An existing Amazon EKS cluster. You can use the Creating an Amazon EKS cluster guide to walk you through creating the cluster.
- *Tip:* We recommend creating and using a new VPC and Amazon EKS cluster to complete this tutorial. Procedures like modifying the aws-auth ConfigMap on the cluster may impact other nodes, users, and workloads on the cluster. Using a new VPC and Amazon EKS cluster will help minimize the risk of impacting other applications as you complete the tutorial.
- *Note:* Do not reference subnets deployed on AWS Outposts when creating Amazon EKS clusters in the Region. The Amazon EKS cluster control plane runs in the Region and attaches to VPC subnets in the Region availability zones.

- A symmetric KMS key to encrypt the EBS volumes of the EKS worker nodes. You can use the alias/aws/ebs AWS managed key for this prerequisite.

Using the sample code

The source code for the `amazon-eks-self-managed-node-group` Terraform module is available in AWS-Samples on GitHub.

Setup

To follow along with this tutorial, complete the following steps to setup your workstation:

1. Open your terminal.
2. Make a new directory to hold your EKS on Outposts Terraform configurations.
3. Change to the new directory.
4. Create a new file named `providers.tf` with the following contents:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }

    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 1.13.3"
    }
  }
}
```

5. Keep your terminal open and do all the work in this tutorial from this directory.

Step 1: Create a self-managed node group

To create a self-managed node group on your Outpost

1. Create a new Terraform configuration file named `self-managed-node-group.tf`.
2. Configure the aws Terraform provider with your AWS CLI profile and Outpost parent Region:

```
provider "aws" {
  region  = "us-west-2"
  profile = "default"
}
```

3. Configure the `aws-eks-self-managed-node-group` module with the following (minimum) arguments:
    - `eks_cluster_name` the name of your EKS cluster
    - `instance_type` an instance type supported on your AWS Outposts deployment
    - `desired_capacity`, `min_size`, and `max_size` as desired to control the number of nodes in your node group (ensure that your Outpost has sufficient resources available to run the desired number nodes of the specified instance type)
    - **subnets** the subnet IDs of the Outpost subnets where you want the nodes deployed
    - (Optional) Kubernetes `node_labels` to apply to the nodes
    - Allow `ebs_encrypted` and configure the `ebs_kms_key_arn` with KMS key you want to use to encrypt the nodes' EBS volumes (required for Outposts deployments)

Example:

```
module "eks_self_managed_node_group" {
  source = "github.com/aws-samples/amazon-eks-self-managed-node-group"

  eks_cluster_name = "cmluns-eks-cluster"
  instance_type    = "m5.2xlarge"
  desired_capacity = 1
  min_size         = 1
  max_size         = 1
  subnets          = ["subnet-0afb721a5cc5bd01f"]

  node_labels = {
    "node.kubernetes.io/outpost"    = "op-0d4579457ff2dc345"
    "node.kubernetes.io/node-group" = "node-group-a"
  }

  ebs_encrypted   = true
  ebs_kms_key_arn = "arn:aws:kms:us-west-2:799838960553:key/0e8f15cc-d3fc-4da4-ae03-
}
```

You may configure other optional arguments as appropriate for your use case – see the module README for details.

Step 2: Configure Kubernetes to allow the nodes to register

Use the following procedures to configure Terraform to manage the AWS IAM Authenticator (`aws-auth`) ConfigMap on the cluster. This adds the node-group IAM role to the IAM authenticator and Kubernetes RBAC groups.

Configure the Terraform Kubernetes Provider to allow Terraform to interact with the Kubernetes control plane.

Note: If you add a node group to a cluster with existing node groups, mapped IAM roles, or mapped IAM users, the aws-auth ConfigMap may already be configured on your cluster. If the ConfigMap exists, you must download, edit, and replace the ConfigMap on the cluster using kubectl. We do not provide a procedure for this operation as it may affect workloads running on your cluster. Please see the section Managing users or IAM roles for your cluster in the Amazon EKS User Guide for more information.

To check if your cluster has the **aws-auth** ConfigMap configured

1. Run the `aws eks --region <region> update-kubeconfig --name <cluster-name>` command to update your workstation's `~/.kube/config` with the information needed to connect to your cluster, substituting your `<region>` and `<cluster-name>`.

```
❯ aws eks --region us-west-2 update-kubeconfig --name cmluns-eks-cluster
Updated context arn:aws:eks:us-west-2:799838960553:cluster/cmluns-eks-cluster in ~/
```

2. Run the `kubectl describe configmap -n kube-system aws-auth`

- If you receive an error stating, `Error from server (NotFound): configmaps "aws-auth" not found`, then *proceed* with the following procedures to use Terraform to apply the ConfigMap.

```
❯ kubectl describe configmap -n kube-system aws-auth
Error from server (NotFound): configmaps "aws-auth" not found
```

- If you do not receive the preceding error, and `kubectl` returns an `aws-auth ConfigMap`, then **you should *not* use Terraform to manage the ConfigMap**.

To configure the Terraform Kubernetes provider

1. Create a new Terraform configuration file named `aws-auth-config-map.tf`.
2. Add the aws_eks_cluster Terraform data source, and configure it to look up your cluster by name.

```
data "aws_eks_cluster" "selected" {
  name = "cmluns-eks-cluster"
}
```

3. Add the aws_eks_cluster_auth Terraform data source, and configure it to look up your cluster by name.

```
data "aws_eks_cluster_auth" "selected" {
  name = "cmluns-eks-cluster"
}
```

4. Configure the `kubernetes` provider with your cluster host (endpoint address), `cluster_ca_certificate`, and the token from the `aws_eks_cluster` and `aws_eks_cluster_auth` data sources.

```
    provider "kubernetes" {
      load_config_file       = false
      host                   = data.aws_eks_cluster.selected.endpoint
      cluster_ca_certificate = base64decode(data.aws_eks_cluster.selected.certificate_a
      token                  = data.aws_eks_cluster_auth.selected.token
    }
```

To configure the AWS IAM Authenticator on the cluster

1. Open the `aws-auth-config-map.tf` Terraform configuration file you created in the last step.
2. Add a kubernetes_config_map Terraform resource to add the `aws-auth ConfigMap` to the `kube-system`
3. Configure the **data** argument with a YAML format heredoc string that adds the Amazon Resource Name (ARN) for your IAM role to the mapRoles

```
resource "kubernetes_config_map" "aws_auth" {
  metadata {
    name      = "aws-auth"
    namespace = "kube-system"
  }

  data = {
    mapRoles = <<-EOT
      - rolearn: ${module.eks_self_managed_node_group.role_arn}
        username: system:node:{{EC2PrivateDNSName}}
        groups:
          - system:bootstrappers
          - system:nodes
    EOT
  }
}
```

Apply the configuration and verify node registration

You have created the Terraform configurations to deploy an EKS self-managed node group to your Outpost and configure Kubernetes to authenticate the nodes. Now you apply the configurations and verify that the nodes register with your Kubernetes cluster.

To apply the Terraform configurations

1. Run the `terraform init` command to download the providers, self-managed node group module, and prepare the directory for use.
2. Run the `terraform apply`
3. Review the resources that will be created.
4. Enter `yes` to confirm that you want to create the resources.

```
❯ terraform apply

Terraform used the selected providers to generate the following execution plan. R
  + create

Terraform will perform the following actions:

<-- Output omitted for brevity -->

Plan: 9 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

5. Press **Enter**.

```
<-- Output omitted for brevity -->

Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
```

6. Run the `aws eks --region <region> update-kubeconfig --name <cluster-name>` command to update your workstation's `~/.kube/config` with the information required to connect to your cluster – substituting your `<region>` and `<cluster-name>`.

```
❯ aws eks --region us-west-2 update-kubeconfig --name cmluns-eks-cluster
Updated context arn:aws:eks:us-west-2:799838960553:cluster/cmluns-eks-cluster in
```

7. Run the `kubectl` get nodes command to view the status of your cluster nodes.
8. Verify your new nodes show up in the list with a **STATUS** of Ready.

```
❯ kubectl get nodes
NAME                                           STATUS   ROLES    AGE   VERSION
ip-10-253-46-119.us-west-2.compute.internal    Ready    <none>   37s   v1.18.9-eks
```

9. (Optional) If your nodes are registering, and their **STATUS** does not show **Ready**, run the `kubectl get nodes --watch` command to watch them come online.
10. (Optional) Run the `kubectl get nodes --show-labels` command to view the node list with the labels assigned to each node. The nodes created by your AWS Outposts node group will have the labels you assigned in Step 1.

To verify the Kubernetes system pods deploy on the worker nodes

1. Run the `kubectl get pods --namespace kube-system`
2. Verify that each pod shows **READY** 1/1 with a **STATUS** of Running.

```
❯ kubectl get pods --namespace kube-system
NAME                       READY   STATUS    RESTARTS   AGE
aws-node-84xlc             1/1     Running   0          2m16s
coredns-559b5db75d-jxqbp   1/1     Running   0          5m36s
coredns-559b5db75d-vdccc   1/1     Running   0          5m36s
kube-proxy-fspw4           1/1     Running   0          2m16s
```

The nodes in your AWS Outposts node group should be registered with the EKS control plane in the Region and the Kubernetes system pods should successfully deploy on the new nodes.

Clean up

One of the nice things about using infrastructure as code tools like Terraform is that they automate stack creation and deletion. Use the following procedure to remove the resources you created in this tutorial.

To clean up the resources created in this tutorial

1. Run the **terraform destroy**
2. Review the resources that will be destroyed.

3. Enter **yes** to confirm that you want to destroy the resources.

```
❯ terraform destroy

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

Plan: 0 to add, 0 to change, 9 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

4. Press **Enter**.

```
Destroy complete! Resources: 9 destroyed.
```

5. Clean up any resources you created for the prerequisites.

Conclusion

In this post, we discussed how containers sit at the heart of the application modernization process, making it easy to adopt microservices architectures that improve application scalability, availability, and performance. We also outlined the challenges associated with modernizing on-premises applications with low latency, local data processing, and data residency requirements. In these cases, AWS Outposts brings cloud services, like Amazon EKS, close to the workloads being refactored. We also walked you through deploying Amazon EKS worker nodes on-premises on AWS Outposts.

Now that you have deployed Amazon EKS worker nodes on in a test VPC using Terraform, you should adapt the Terraform module(s) and resources to prepare and deploy your production Kubernetes clusters on-premises on Outposts. If you don't have an Outpost and you want to get started modernizing your on-premises applications with Amazon EKS and AWS Outposts, contact your local AWS account Solutions Architect (SA) to help you get started.

TAGS: Amazon EKS, AWS Outposts