**AWS EKS topics for review**

**STORAGE :**

**STORAGE CLASSES :** you should define a default storage class for your persistent volume claims

* A **persistent volume** (PV) is a cluster-wide resource that you can use to store data in a way that it persists beyond the lifetime of a pod .

**To create an AWS storage class for your Amazon EKS cluster :**

1. Create an AWS storage class manifest file for your storage class. The gp2-storage-class.yaml example below defines a storage class called gp2 that uses the Amazon EBS gp2 volume type.

**gp2-storage-class.yaml**

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: gp2

annotations:

storageclass.kubernetes.io/is-default-class: "true"

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2

fsType: ext4

1. Use kubectl to create the storage class from the manifest file :

**kubectl create -f gp2-storage-class.yaml**

**To define a default storage class :**

1. List the existing storage classes for your cluster. A storage class must be defined before you can set it as a default.

**kubectl get storageclass**

1. Choose a storage class and set it as your default by setting the storageclass.kubernetes.io/ is-default-class=true annotation.

**kubectl patch storageclass gp2 -p '{"metadata": {"annotations": {"storageclass.kubernetes.io/is-default-class":"true"}}}'**

1. Verify that the storage class is now set as default.

**kubectl get storageclass**

**you will see the following o/p :** gp2 (default) kubernetes.io/aws-ebs 12m

**Resilience in Amazon EKS**

The AWS global infrastructure is built around AWS Regions and Availability Zones. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

Amazon EKS runs Kubernetes control plane instances across multiple Availability Zones to ensure high availability. Amazon EKS automatically detects and replaces unhealthy control plane instances, and it provides automated version upgrades and patching for them.

This control plane consists of at least two API server nodes and three etcd nodes that run across three Availability Zones within a Region. Amazon EKS automatically detects and replaces unhealthy control plane instances, restarting them across the Availability Zones within the Region as needed. Amazon EKS leverages the architecture of AWS Regions in order to maintain high availability.

**Infrastructure security in Amazon EKS**

You use AWS published API calls to access Amazon EKS through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Amazon EKS requires subnets in at least two Availability Zones with public and private subnets so that Kubernetes can create public load balancers in the public subnets that load balance traffic to pods running on nodes that are in private subnets.

When you create a new cluster, Amazon EKS creates an endpoint for the managed Kubernetes API server that you use to communicate with your cluster (using Kubernetes management tools such as kubectl). By default, this API server endpoint is public to the internet, and access to the API server is secured using a combination of AWS Identity and Access Management (IAM) and native Kubernetes Role Based Access Control (RBAC).

You can enable private access to the Kubernetes API server so that all communication between your nodes and the API server stays within your VPC. You can limit the IP addresses that can access your API server from the internet, or completely disable internet access to the API server.

**WHAT IS KUBECONFIG ?**

To access the cluster kubectl uses config file .Default kube config file location is ~/.kube/config .

Kube config organizes information about clusters, users, namespaces, & authentication mechanisms. The kubectl command uses these files to find the information it needs to choose a cluster and communicate with it.

**To create your kubeconfig file with the AWS CLI**

Use the AWS CLI update-kubeconfig command to create or update a kubeconfig for your cluster. Default path is ~/.kube/config , You can specify another path with the --kubeconfig option.

You can specify an IAM role ARN with the --role-arn option to use for authentication when you issue kubectl commands. Otherwise, the IAM entity in your default AWS CLI or SDK credential chain is used. You can view your default AWS CLI or SDK identity by running the aws sts getcaller-identity command.

Certificate : The Base64-encoded certificate data required to communicate with your cluster.

DescribeCluster : Returns descriptive information about an Amazon EKS cluster.

To run the nxt command u need to have eks:DescribeCluster api action permission with the cluster.

**aws eks --region us-west-2 update-kubeconfig --name cluster\_name**

Test your configuration : kubectl get svc

Example kubeconfig file :

You have 2 clusters – 1. Development 2. Scratch

* In development u have – Namespaces frontend & storage , Access to this cluster requires authentication by certificate. In scratch they work in default namespace . Access requires authentication by username and password.

Create a file named config-demo:

apiVersion: v1

kind: Config

preferences: {}

clusters:

- cluster:

name: development

- cluster:

name: scratch

users:

- name: developer

- name: experimenter

contexts:

- context:

name: dev-frontend

- context:

name: dev-storage

- context:

name: exp-scratch

kubectl config --kubeconfig=config-demo set-cluster development --server=https://1.2.3.4 --certificate-authority=fake-ca-file

kubectl config --kubeconfig=config-demo set-cluster scratch --server=https://5.6.7.8 --insecure-skip-tls-verify

Add user details to your configuration file:

kubectl config --kubeconfig=config-demo set-credentials developer --client-certificate=fake-cert-file --client-key=fake-key-seefile

kubectl config --kubeconfig=config-demo set-credentials experimenter --username=exp --password=some-password

Add context details to your configuration file:

kubectl config --kubeconfig=config-demo set-context dev-frontend --cluster=development --namespace=frontend --user=developer

kubectl config --kubeconfig=config-demo set-context dev-storage --cluster=development --namespace=storage --user=developer

kubectl config --kubeconfig=config-demo set-context exp-scratch --cluster=scratch --namespace=default --user=experimenter

**open your config demo to see added details :**

kubectl config --kubeconfig=config-demo view

apiVersion: v1

clusters:

- cluster:

certificate-authority: fake-ca-file

server: https://1.2.3.4

name: development

- cluster:

insecure-skip-tls-verify: true

server: https://5.6.7.8

name: scratch

contexts:

- context:

cluster: development

namespace: frontend

user: developer

name: dev-frontend

- context:

cluster: development

namespace: storage

user: developer

name: dev-storage

- context:

cluster: scratch

namespace: default

user: experimenter

name: exp-scratch

current-context: ""

kind: Config

preferences: {}

users:

- name: developer

user:

client-certificate: fake-cert-file

client-key: fake-key-file

- name: experimenter

user:

password: some-password

username: exp

Sometimes you may want to use Base64-encoded data embedded here instead of separate certificate files; in that case you need add the suffix -data to the keys, for example, certificate-authority-data, client-certificate-data, client-key-data.

To see only the configuration information associated with the current context, use the --minify flag.

kubectl config --kubeconfig=config-demo view --minify

**CLUSTER AUTOSCALER :**

The Kubernetes Cluster Autoscaler automatically adjusts the number of nodes in your cluster when pods fail to launch due to lack of resources or when nodes in the cluster are underutilized and their pods can be rescheduled onto other nodes in the cluster.

**Create an Amazon EKS cluster : if u already have a cluster then goto autoscaler considerations**

If u don’t have a cluster created then follow the nxt steps –

If u want ur application to run in multiple azs then you can either create multiple node grps each assigned to 1 az and then enable balance similar node grps feature or create a single node grp that spans across multiple azs .

* Single node grp in multiple azs :

**eksctl create cluster --name my-cluster --version 1.17 --managed --asg-access**

* Cluster with a dedicated managed node group for each Availability Zone :

1. Create an Amazon EKS cluster with no node groups with the following eksctl command. Note the availability zones in which the cluster is created in .

**eksctl create cluster --name my-cluster --version 1.17 --without-nodegroup**

1. For each Availability Zone in your cluster, use the following eksctl command to create a node group.

**eksctl create nodegroup --cluster my-cluster --node-zones region-codea --name regioncodea --asg-access --nodes-min 1 --nodes 5 --nodes-max 10 –managed**

**Cluster Autoscaler node group considerations :**

The Cluster Autoscaler requires additional IAM and resource tagging considerations

**Node group IAM policy**

Cluster autoscaler requires the following permission to make calls to aws apis on ur behalf.

If cluster is created using the previous eksctl command then the required IAM permissions are automatically provided and attached to your node IAM roles.

If you did not use eksctl, you must create an IAM policy with the following document and attach it to your node IAM roles.

{

"Version": "2012-10-17",

"Statement": [

{

"Action": [

"autoscaling:DescribeAutoScalingGroups",

"autoscaling:DescribeAutoScalingInstances",

"autoscaling:DescribeLaunchConfigurations",

"autoscaling:DescribeTags",

"autoscaling:SetDesiredCapacity",

"autoscaling:TerminateInstanceInAutoScalingGroup",

"ec2:DescribeLaunchTemplateVersions"

],

"Resource": "\*",

"Effect": "Allow"

} ] }

**Auto Scaling group tags :**

The Cluster Autoscaler requires the following tags on your node group Auto Scaling groups so that they can be auto-discovered.

If you used the previous eksctl commands to create your node groups, these tags are automatically applied. Otherwise manually tag ur resources with the following tags.

k8s.io/cluster-autoscaler/<cluster name> owned

k8s.io/cluster-autoscaler/enabled true

**Deploy cluster autoscaler :**

kubectl apply -f https://raw.githubusercontent.com/kubernetes/autoscaler/master/ cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-autodiscover.yaml

Add the cluster-autoscaler.kubernetes.io/safe-to-evict annotation to the deployment with the following command.

kubectl -n kube-system annotate deployment.apps/cluster-autoscaler clusterautoscaler.kubernetes.io/safe-to-evict="false"

Edit the Cluster Autoscaler deployment with the following command.

kubectl -n kube-system edit deployment.apps/cluster-autoscaler

Edit the cluster-autoscaler container command to replace with your cluster's name, and add the following options.

• --balance-similar-node-groups

• --skip-nodes-with-system-pods=false

spec: containers:

- command:

- ./cluster-autoscaler

- --v=4 - --stderrthreshold=info

- --cloud-provider=aws

- --skip-nodes-with-local-storage=false

- --expander=least-waste

- --node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/

cluster-autoscaler/clustername

- --balance-similar-node-groups

- --skip-nodes-with-system-pods=false

* Cluster Autoscaler has an option to skip-nodes-with-system-pods which is true by default. If you set this option to false, it can drain and therefore scale-down all the nodes even if pods from the kube-system namespace are there.
* balance similar node grps will find the node grps with same instance type or labels and maintain the same size between all of those node grps.

Open the Cluster Autoscaler releases page in a web browser and find the latest Cluster Autoscaler version that matches your cluster's Kubernetes major and minor version. Record the version(1.17.n).

Set the Cluster Autoscaler image tag:

**kubectl -n kube-system set image deployment.apps/cluster-autoscaler clusterautoscaler=us.gcr.io/k8s-artifacts-prod/autoscaling/cluster-autoscaler:v1.15.n**

**View your Cluster Autoscaler logs :**

kubectl -n kube-system logs -f deployment.apps/cluster-autoscaler

**Amazon EKS control plane logging :**

Amazon EKS control plane logging provides audit and diagnostic logs directly from the Amazon EKS control plane to CloudWatch Logs in your account. You can start using Amazon EKS control plane logging by choosing which log types you want to enable for each new or existing Amazon EKS cluster. When enabled, logs are automatically sent from the Amazon EKS cluster to CloudWatch Logs in the same account.

When you use Amazon EKS control plane logging, you're charged standard Amazon EKS pricing for each cluster that you run. You are charged the standard CloudWatch Logs data ingestion and storage costs for any logs sent to CloudWatch Logs from your clusters. You are also charged for any AWS resources, such as Amazon EC2 instances or Amazon EBS volumes, that you provision as part of your cluster.

**LOG TYPES :**

Kubernetes API server component logs (api) : Your cluster's API server is the control plane component that exposes the Kubernetes API.

Audit (audit) : Kubernetes audit logs provide a record of the individual users, administrators, or system components that have affected your cluster.

Authenticator (authenticator) : Authenticator logs are unique to Amazon EKS. These logs represent the control plane component that Amazon EKS uses for Kubernetes Role Based Access Control (RBAC) authentication using IAM credentials.

Controller manager (controllerManager) : The controller manager manages the core control loops that are shipped with Kubernetes.

Scheduler (scheduler) : The scheduler component manages when and where to run pods in your cluster.

**Enabling and disabling control plane logs**

CloudWatch Logs ingestion, archive storage, and data scanning rates apply to enabled control plane logs

**Using console :**

Open amazon eks 🡺cluster 🡺logging 🡺choose update 🡺for each individual log type choose enable or disable .

**Using AWS CLI :**

Aws version should not be less than 1.16.139 check version using  **aws –version**

Update your cluster's control plane log export configuration with the following AWS CLI command.

**aws eks --region region-code update-cluster-config --name prod \ --logging '{"clusterLogging":[{"types": ["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}'**

Monitor the status of your log configuration update. Your update is complete when the status appears as Successful.

**aws eks --region region-code describe-update --name prod --updateid 883405c8-65c6-4758-8cee-2a7c1340a6d9**

**Viewing cluster control plane logs :**

Open cloud watch console 🡺choose the cluster 🡺Choose the log stream to view. The following list describes the log stream name format for each log type.

**Note** As log stream data grows, the log stream names are rotated. When multiple log streams exist for a particular log type, you can view the latest log stream by looking for the log stream name with the latest Last Event Time.

• Kubernetes API server component logs (api) – kube-apiserver-nnn...

• Audit (audit) – kube-apiserver-audit-nnn...

• Authenticator (authenticator) – authenticator-nnn...

• Controller manager (controllerManager) – kube-controller-manager-nnn...

• Scheduler (scheduler) – kube-scheduler-nnn...

**DASHBOARD :**

**Pre-requisites :**

* A cluster should be present to deploy our dashboard on .
* Security grps for control plane elastic interface and nodes should follow the recommended settings.
* kubectl should be installed and configured to communicate with cluster

**STEP – 1 :** Deploy the Kubernetes Metrics Server

The Kubernetes Dashboard uses the metrics server to gather metrics for your cluster, such as CPU and memory usage over time.

**TO DEPLOY METRICS SERVER :**

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/ v0.3.6/components.yaml

**Verify if the metrics server is running using :**

kubectl get deployment metrics-server -n kube-system

**STEP - 2** : Deploy Dashboard

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/ deploy/recommended.yaml

**STEP – 3 :** Create an EKS-Service admin account and cluster role binding to securely connect to the dashboard with admin-level permissions.

eks-admin-service-account.yaml

apiVersion: v1

kind: ServiceAccount

metadata:

name: eks-admin

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1beta1   
kind: ClusterRoleBinding

metadata:

name: eks-admin

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: eks-admin

namespace: kube-system

**kubectl apply -f eks-admin-service-account.yaml**

**STEP – 4 : Connect to Dashboard**

* Retrieve an authentication token for the eks-admin service account to use it to connect to dashboard.

**kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep eksadmin | awk '{print $1}')**

* Start the kubectl proxy – **kubectl proxy**
* To access the dashboard endpoint, open the following link with a web browser: http:// localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/ proxy/#!/login
* Choose token and paste the token retrieved.

**STEP – 5 : NEXT STEPS**

After you have connected to your Kubernetes Dashboard, you can view and control your cluster using your eks-admin service account.

**Prometheus :**

Kubernetes API server has many metrics exposed used for monitoring & analysis. The metrics end point is /metrics http api ,this is also exposed in the control plane .

**Viewing the raw metrics**

By using –raw flag u can pass any http path and get the raw response .

Eg : kubectl get --raw /metrics

rest\_client\_requests\_total{code="200",host="127.0.0.1:21362",method="POST"} 4994

The output is in the Prometheus format where we will have metric name, tag and value .

if you want to analyze these metrics over time then deploy Prometheus in the cluster.

**Prometheus** is a monitoring and time series database that scrapes exposed endpoints and aggregates data, allowing you to filter, graph, and query the results.

**Deploying Prometheus using Helm:**

1. Create a Prometheus namespace : **kubectl create namespace Prometheus**
2. Deploy Prometheus : helm install prometheus stable/prometheus \

--namespace prometheus\

--set

alertmanager.persistentVolume.storageClass="gp2",server.persistentVolume.storageClass="gp2"

1. Verify that all of the pods in the prometheus namespace are in the READY state:

kubectl get pods -n Prometheus

1. Use kubectl to port forward the Prometheus console to your local machine

kubectl --namespace=prometheus port-forward deploy/prometheus-server 9090

**Kubectl port**-**forward** allows you to access and interact with internal **Kubernetes** cluster processes from your localhost.

1. Point a web browser to localhost:9090 to view the Prometheus console.
2. Choose a metric and then click on execute to see the metrics also choose status and targets to see all the target endpoints.

**USING HELM WITH EKS:**

The Helm package manager for Kubernetes helps you install and manage applications on your Kubernetes cluster.

**To install the Helm binaries on your local system :**

1. for linux :

curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 >

get\_helm.sh

chmod 700 get\_helm.sh

./get\_helm.sh

1. To pick up the new binary in your PATH, Close your current terminal window and open a new one.
2. Confirm that Helm is running with the following command : **helm help**
3. Now, we can run any Helm commands (such as helm install chart\_name) to install, modify, delete, or query Helm charts in your cluster.

**Vertical Pod Autoscaler :**

The Kubernetes Vertical Pod Autoscaler automatically adjusts the CPU and memory reservations for your pods to help "right size" your applications.

**Metrics Server** :

1. Install the metrics server : It provides metrics that are required by the Vertical Pod Autoscaler.

Check if u have already deployed a metrics server using –

**kubectl -n kube-system get deployment/metrics-server** (if it returns not found then next step)

1. kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/ v0.3.6/components.yaml
2. verify using => kubectl get deployment metrics-server -n kube-system

**Deploy the Vertical Pod Autoscaler :**

1. Download the vertical pod autoscaler source using :

git clone <https://github.com/kubernetes/autoscaler.git>

1. cd autoscaler/vertical-pod-autoscaler/
2. If you have already deployed another version of the Vertical Pod Autoscaler, remove it using :

./hack/vpa-down.sh

1. Deploy the Vertical Pod Autoscaler to your cluster with the following command :

./hack/vpa-up.sh

1. Verify using : kubectl get pods -n kube-system

**Test your Vertical Pod Autoscaler installation :**

1. Deploy the hamster.yaml Vertical Pod Autoscaler a sample application .

kubectl apply -f examples/hamster.yaml

1. Get the pods from the hamster example application

kubectl get pods -l app=hamster

1. Describe one of the pods to view its CPU and memory reservation

kubectl describe pod hamster-c7d89d6db-rglf5

1. The Vertical Pod Autoscaler vpa-recommender deployment analyzes the hamster pods to see if the CPU and memory requirements are appropriate. If adjustments are needed, the vpa-updater relaunches the pods with updated values.
2. Describe the hamster-vpa resource to view the new recommendation.

kubectl describe vpa/hamster-vpa

In the yaml file you can see the following in recommendations part :

Recommendation:

Container Recommendations:

Container Name: hamster

Lower Bound:

Cpu: 550m

Memory: 262144k

Target:

Cpu: 587m

Memory: 262144k

Uncapped Target:

Cpu: 587m

Memory: 262144k

Upper Bound:

Cpu: 21147m

Memory: 387863636

You can also delete the application using ,

**kubectl delete -f examples/hamster.yaml**

**Horizontal Pod Autoscaler :**

The Kubernetes Horizontal Pod Autoscaler automatically scales the number of pods in a deployment, replication controller, or replica set based on that resource's CPU utilization. This can help your applications scale out to meet increased demand or scale in when resources are not needed, thus freeing up your nodes for other applications. When you set a target CPU utilization percentage, the Horizontal Pod Autoscaler scales your application in or out to try to meet that target.

* You do not need to deploy or install the Horizontal Pod Autoscaler on your cluster to begin scaling your applications

**Install the metrics server :**

It needs a metrics source to be installed in the cluster to work .

**Run a Horizontal Pod Autoscaler test application :**

1. Deploy a simple Apache web server application with the following command.

kubectl apply -f <https://k8s.io/examples/application/php-apache.yaml>

1. Create a Horizontal Pod Autoscaler resource for the php-apache deployment

kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10

1. kubectl describe hpa
2. Create a load for the web server by running a container.

kubectl run -it --rm load-generator --image=busybox /bin/sh --generator=run-pod/v1

1. To watch the deployment scale out, periodically - kubectl get hpa

As long as actual CPU percentage is higher than the target percentage, then the replica count increases.

1. Stop the load. In the terminal window you're generating the load in (from step 4), stop the load by holding down the Ctrl+C keys. You can see the replicas scale back to 1. The default timeframe for scaling back down is five minutes.
2. kubectl delete deployment.apps/php-apache service/php-apache horizontalpodautoscaler.autoscaling/php-apache

**ALB Ingress Controller on Amazon EKS**

The AWS ALB Ingress Controller for Kubernetes is a controller that triggers the creation of an Application Load Balancer (ALB) and the necessary supporting AWS resources whenever an Ingress resource is created on the cluster with the kubernetes.io/ingress.class: alb annotation. The Ingress resource configures the ALB to route HTTP or HTTPS traffic to different pods within the cluster.

The ALB Ingress controller supports the following traffic modes:

**Instance :** Registers nodes within your cluster as targets for the ALB. Traffic reaching the ALB is routed to NodePort for your service and then proxied to your pods. This is the default traffic mode. You can also explicitly specify it with the alb.ingress.kubernetes.io/target-type: instance annotation.

**IP :** Registers pods as targets for the ALB. Traffic reaching the ALB is directly routed to pods for your service. You must specify the alb.ingress.kubernetes.io/target-type: ip annotation to use this traffic mode.

**To deploy the ALB Ingress Controller to an Amazon EKS cluster**

• All subnets in your VPC should be tagged accordingly so that Kubernetes can discover them.

kubernetes.io/cluster/cluster name - shared

• Public subnet for external load balancers -

kubernetes.io/role/elb - 1

• Private subnets for internal load balancers –

kubernetes.io/role/internal-elb - 1

**Create an IAM OIDC provider and associate it with your cluster :**

[ IAM OIDC identity providers are entities in IAM that describe an external identity provider (IdP) service that supports the [OpenID Connect](http://openid.net/connect/) (OIDC) standard  You use an IAM OIDC identity provider when you want to establish trust between an OIDC-compatible IdP and your AWS account when you need access to aws account but u don’t want to create custom sign-in code or manage your own user identities the opt this oidc ]

**eksctl utils associate-iam-oidc-provider \ --region region-code \ --cluster prod \ --approve**

**Download an IAM policy for the ALB Ingress Controller pod** that allows it to make calls to AWS APIs on your behalf.

curl -o iam-policy.json <https://raw.githubusercontent.com/kubernetes-sigs/aws-albingress-controller/v1.1.8/docs/examples/iam-policy.json>

**Create an IAM policy called ALBIngressControllerIAMPolicy using the policy downloaded**

aws iam create-policy \ --policy-name ALBIngressControllerIAMPolicy \ --policy-document <file://iam-policy.json>

**Create a Kubernetes service account named alb-ingress-controller** in the kube-system namespace, a cluster role, **and a cluster role binding for the ALB Ingress Controller** to use with the following command.

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/rbac-role.yaml>

**Create an IAM role for the ALB Ingress Controller and attach the role** to the service account created in the previous step.

eksctl create iamserviceaccount \

--region region-code \

--name alb-ingress-controller \

--namespace kube-system \

--cluster prod \

--attach-policy-arn arn:aws:iam::111122223333:policy/ALBIngressControllerIAMPolicy \

--override-existing-serviceaccounts \

--approve

**Deploy the ALB Ingress Controller** with the following command :

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/alb-ingress-controller.yaml>

Open the ALB Ingress Controller **deployment** **manifest for editing** with the following command.

kubectl edit deployment.apps/alb-ingress-controller -n kube-system

**Add a line for the cluster name after the --ingress-class=alb line**, If you're running the ALB Ingress Controller on Fargate, then you must also **add the lines for the VPC ID, and AWS Region name** of your cluster.

spec:

containers:

- args:

- --ingress-class=alb

- --cluster-name=prod

- --aws-vpc-id=vpc-03468a8157edca5bd

- --aws-region=region-code

**Confirm that the ALB Ingress Controller is running :** kubectl get pods -n kube-system

**To deploy a sample application :**

Amazon EC2 nodes only ,

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/2048/2048-namespace.yaml>

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/2048/2048-deployment.yaml>

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/2048/2048-service.yaml>

kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingresscontroller/v1.1.8/docs/examples/2048/2048-ingress.yaml>

**Fargate : create a fargate profile**

* eksctl create fargateprofile --cluster prod --region region-code --name albsample-app --namespace 2048-game
* Download and apply the manifest files to create the Kubernetes namespace, deployment, and service with the following commands.

kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/aws-albingress-controller/v1.1.8/docs/examples/2048/2048-namespace.yaml kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/aws-albingress-controller/v1.1.8/docs/examples/2048/2048-deployment.yaml kubectl apply -f <https://raw.githubusercontent.com/kubernetes-sigs/aws-albingress-controller/v1.1.8/docs/examples/2048/2048-service.yaml>

* Download the ingress manifest file with the following command.

curl -o 2048-ingress.yaml <https://raw.githubusercontent.com/kubernetes-sigs/awsalb-ingress-controller/v1.1.8/docs/examples/2048/2048-ingress.yaml>

* Edit the 2048-ingress.yaml file. Under the existing alb.ingress.kubernetes.io/ scheme: internet-facing line , add the line alb.ingress.kubernetes.io/targettype: ip.

Apply the ingress manifest file with the following command

**kubectl apply -f 2048-ingress.yaml**

* **kubectl get ingress/2048-ingress -n 2048-game**
* Open a browser and navigate to the ADDRESS URL from the previous command output to see the sample application

**Pod security policy**

The Kubernetes pod security policy admission controller validates pod creation and update requests against a set of rules. By default, Amazon EKS clusters ship with a fully permissive security policy with no restrictions.

**Amazon EKS default pod security policy :**

Amazon EKS clusters with Kubernetes version 1.13 and higher have a default pod security policy named eks.privileged. This policy has no restriction on what kind of pod can be accepted into the system, which is equivalent to running Kubernetes with the PodSecurityPolicy controller disabled.

* We can view default policy using ,

kubectl get psp eks.privileged

* Describe the policy using ,

kubectl describe psp eks.privileged

**YAML file for the eks.privileged pod security policy**

---

apiVersion: policy/v1beta1

kind: PodSecurityPolicy

metadata:

name: eks.privileged

annotations:

kubernetes.io/description: 'privileged allows full unrestricted access to pod features, as if the PodSecurityPolicy controller was not enabled.' seccomp.security.alpha.kubernetes.io/allowedProfileNames: '\*'

labels:

kubernetes.io/cluster-service: "true"

eks.amazonaws.com/component: pod-security-policy

spec:

privileged: true ("privileged" container is given access to all devices on the host)

allowPrivilegeEscalation: true (this will allow any authenticated user to access pod as a root user)

allowedCapabilities:

- '\*'

volumes:

- '\*'

hostNetwork: true (Controls whether the pod may use the node network namespace)

hostPorts: (allowed ports on host network)

- min: 0

max: 65535

hostIPC: true (controls whether pods can share hostipc )

hostPID: true (controls whether pods can share host process id )

runAsUser:

rule: 'RunAsAny'

seLinux: rule: 'RunAsAny'

supplementalGroups:

rule: 'RunAsAny'

fsGroup:

rule: 'RunAsAny'

readOnlyRootFilesystem: false

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: eks:podsecuritypolicy:privileged

labels:

kubernetes.io/cluster-service: "true"

eks.amazonaws.com/component: pod-security-policy

rules:

- apiGroups:

- policy

resourceNames:

- eks.privileged

resources:

- podsecuritypolicies

verbs:

- use

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: eks:podsecuritypolicy:authenticated

annotations:

kubernetes.io/description: 'Allow all authenticated users to create privileged pods.'

labels:

kubernetes.io/cluster-service: "true"

eks.amazonaws.com/component: pod-security-policy

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: eks:podsecuritypolicy:privileged

subjects:

- kind: Group

apiGroup: rbac.authorization.k8s.io

name: system:authenticated

**Delete the pod security policy : this will delete policy , cluster role and its binding**

kubectl delete -f privileged-podsecuritypolicy.yaml

**To restore :**

Copy the yml file in privileged-podsecuritypolicy.yaml and then ,

kubectl apply -f privileged-podsecuritypolicy.yaml

**KUBERNETES POD SECURITY POLICY :**

Spec will have the policy rules :

1. Running of privileged containers : field name – privileged

  By default a container is not allowed to access any devices on the host, but a "privileged" container is given access to all devices on the host.

1. Privilege Escalation : AllowPrivilegeEscalation, DefaultAllowPrivilegeEscalation

AllowPrivilegeEscalation : Gates whether or not a user is allowed to set the security context of a container to allowPrivilegeEscalation=true. This defaults to allowed so as to not break setuid binaries.

DefaultAllowPrivilegeEscalation :  Sets the default for the allowPrivilegeEscalation option. The default behavior without this is to allow privilege escalation so as to not break setuid binaries

1. Capabilities :  privileges traditionally associated with the superuser

**AllowedCapabilities -**  Provides a list of capabilities that are allowed to be added to a container. The default set of capabilities are implicitly allowed. The empty set means that no additional capabilities may be added beyond the default set. \* can be used to allow all capabilities.

**RequiredDropCapabilities** - The capabilities which must be dropped from containers. These capabilities are removed from the default set, and must not be added.

**DefaultAddCapabilities** - The capabilities which are added to containers by default, in addition to the runtime defaults.

1. **Volumes :** Provides a list of allowed volume types. The allowable values correspond to the volume sources that are defined when creating a volume. \* means all types.
2. **HostNetwork :** Controls whether the pod may use the node network namespace. It could be used to snoop on network activity of other pods on the same node.
3. **HostPorts :** Provides a list of ranges of allowable ports in the host network namespace.
4. **HostIPC :** Controls whether the pod containers can share the host IPC(Interprocess Communication) namespace.
5. **HostPID :** Controls whether the pod containers can share the host process ID namespace.
6. **Users and groups :**

**RunAsUser :** Controls which user ID the containers can run with.

MustRunAs, MustRunAsNonRoot, RunAsAny

**RunAsGroup :** Controls which primary group ID the containers can run with.

MustRunAs,MayRunAs,RunAsAny

**SupplementalGroups :** Controls which group IDs containers add.

MustRunAs,MayRunAs,RunAsAny

1. **SELinux : SELinux** is a labeling system for processes and files

**MustRunAs :**  Requires seLinuxOptions to be configured. Uses seLinuxOptions as the default. Validates against seLinuxOptions.

**RunAsAny :** No default provided. Allows any seLinuxOptions to be specified.

1. **FSGroup :** Controls the supplemental group applied to some volumes.

**MustRunAs :** Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.

**MayRunAs:** Allows FSGroups to be left unset without providing a default. Validates against all ranges if FSGroups is set.

**RunAsAny :**  No default provided. Allows any fsGroup ID to be specified

1. **ReadOnlyRootFilesystem** - Requires that containers must run with a read-only root filesystem

**WINDOWS SUPPORT**

**Considerations :**

* Kubernetes version 1.14 or later
* Amazon EC2 instance types C3, C4, D2, I2, M4,R3 are not supported for windows workloads.
* Amazon EKS clusters must contain one or more Linux nodes to run core system pods that only run on Linux, such as coredns and the VPC resource controller.
* Windows nodes support one elastic network interface per node. The number of pods that you can run per Windows node is equal to the number of IP addresses available per elastic network interface for the node's instance type, minus one.
* Group Managed Service Accounts (GMSA) for Windows pods and containers is not supported for version 1.16 and earlier.

**Enabling Windows support :**

**Eksctl :** eksctl version is 0.26.0 or later

1. eksctl utils install-vpc-controllers --cluster cluster\_name –approve
2. launch node grp into the cluster

After you add Windows support to your cluster, you must specify node selectors on your applications so that the pods land on a node with the appropriate operating system.

**For linux** :

nodeSelector:

kubernetes.io/os: linux

kubernetes.io/arch: amd64

**For windows** :

nodeSelector:

kubernetes.io/os: windows

kubernetes.io/arch: amd64

**using windows client :**

1. deploy vpc resource controller
2. deploy vpc admission controller
3. cluster role binding :

eks-kube-proxywindows-crb.yaml

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1beta1

metadata:

name: eks:kube-proxy-windows

labels: k8s-app: kube-proxy eks.amazonaws.com/component: kube-proxy

subjects:

- kind: Group name: "eks:kube-proxy-windows"

roleRef:

kind: ClusterRole

name: system:node-proxier

apiGroup: rbac.authorization.k8s.io

1. Apply the config to cluster : kubectl apply -f eks-kube-proxy-windows-crb.yaml
2. Node selector for application .

**Deploy a Windows sample application :**

windows-server-iis.yaml

apiVersion: apps/v1 kind: Deployment metadata: name: windows-server-iis spec: selector: matchLabels: app: windows-server-iis tier: backend track: stable replicas: 1 template: metadata: labels: app: windows-server-iis tier: backend track: stable spec: containers: - name: windows-server-iis image: mcr.microsoft.com/windows/servercore:1809 ports: - name: http containerPort: 80 imagePullPolicy: IfNotPresent command: - powershell.exe - -command - "Add-WindowsFeature Web-Server; Invoke-WebRequest -UseBasicParsing -Uri 'https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.6/ ServiceMonitor.exe' -OutFile 'C:\\ServiceMonitor.exe'; echo '

# 

' > C:\\inetpub\\wwwroot\ \default.html; C:\\ServiceMonitor.exe 'w3svc'; " nodeSelector: kubernetes.io/os: windows --- apiVersion: v1 kind: Service metadata: name: windows-server-iis-service namespace: default spec: ports: - port: 80 protocol: TCP targetPort: 80 selector: app: windows-server-iis tier: backend track: stable sessionAffinity: None type: LoadBalancer

1. **Deploy** kubectl apply -f windows-server-iis.yaml

**MANAGED VS UNMANAGED NODE GROUPS :**

**Managed Node groups :**

It’s also another managed Kubernetes service where you don’t have to maintain or create the cluster control plane. EKS runs a cluster control plane in multiple AZ to ensure it maintains high availability, and automatically replaces unhealthy instances.

**Self managed node groups :**

 with a self-managed Kubernetes service we have more command over the cluster control plane.

**RBAC :**

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

1 role

2 cluster role

**ARM support :**

Amazon EKS optimized Arm Amazon Linux AMIs Arm instances deliver significant cost savings for scale-out and Arm-based applications such as web servers, containerized microservices, caching fleets, and distributed data stores.

When adding Arm nodes to your cluster, review the following considerations. Considerations   
• You can only deploy Arm AMIs in 1.15 or later clusters.

• If your cluster was deployed before August 17, 2020, then you must do a one-time upgrade of critical cluster add-on manifests so that Kubernetes can pull the correct image for each hardware architecture in use in your cluster. If you deployed your cluster on or after August 17, 2020, then your coredns, kube-proxy, and Amazon VPC CNI Plugin for Kubernetes add-ons are already multiarchitecture capable.

• Applications deployed to Arm nodes must be compiled for Arm.

• You can't use any of the Cluster Storage Interface drivers with Arm.

• If you have any daemonsets deployed in an existing cluster, or you want to deploy them to a new cluster that you also want to deploy Arm nodes in, then ensure that your daemonset can run on all hardware architectures in your cluster.

• You can run Arm node groups and x86 node groups in the same cluster. If you do, consider deploying multi-architecture container images to a container repository such as Amazon Elastic Container Registry and then adding node selectors to your manifests so that Kubernetes knows what harware architecture a pod can be deployed to.

**SERVICE ACCOUNT :**

**Service user** – If you use the Amazon EKS service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Amazon EKS features to do your work, you might need additional permissions.

**Service administrator** – If you're in charge of Amazon EKS resources at your company, you probably have full access to Amazon EKS. It's your job to determine which Amazon EKS features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users.

**IAM administrator** – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Amazon EKS. To view example Amazon EKS identity-based policies that you can use in IAM, see Amazon EKS identity-based policy examples (p. 262).