Table of Contents

Ref. & Source code :

- https://github.com/vishalk17/eks_learn
- https://github.com/vishalk17/devops/

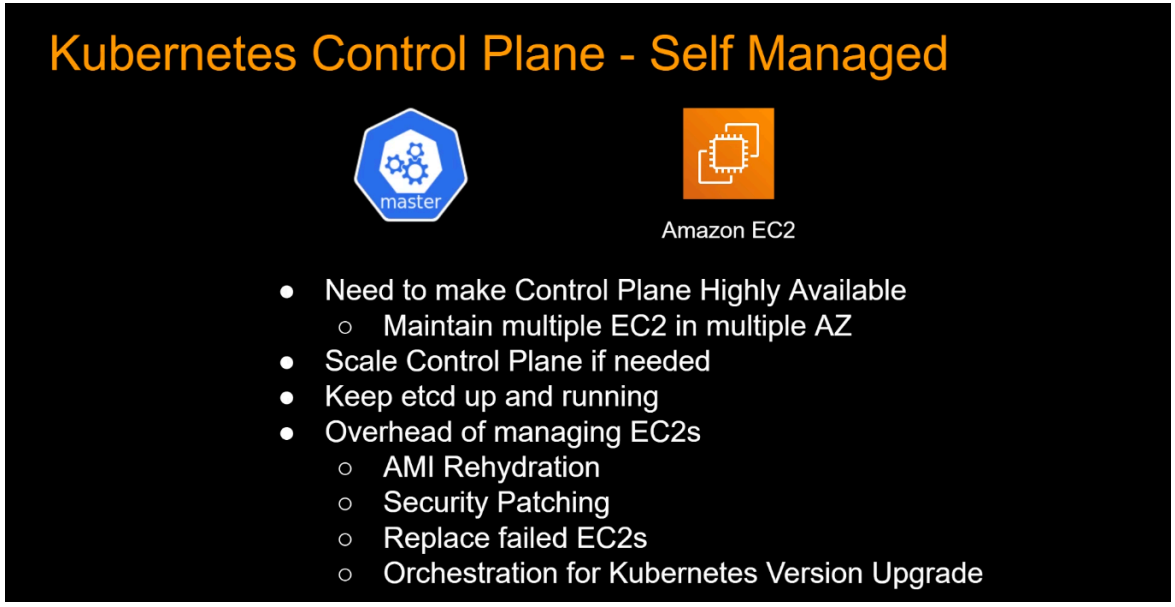
1.0 : What is EKS ?

EKS (Elastic Kubernetes Service) is a **managed Kubernetes service** provided by **AWS (Amazon Web Services)**. It allows users to run **Kubernetes clusters** in the cloud **without managing the control plane** (i.e., the master nodes). AWS handles cluster provisioning, scaling, patching, and high availability.

1.0.1 : Key Features of EKS

1. **Fully Managed** – AWS takes care of the Kubernetes control plane, including updates and scaling.
2. **High Availability** – EKS runs across multiple **Availability Zones (AZs)** for redundancy.
3. **Integrated with AWS Services** – Works well with IAM, VPC, ALB, CloudWatch, and other AWS services.
4. **Supports EC2 and Fargate** – You can run workloads on both EC2 instances and AWS Fargate (serverless).
5. **Secure** – Uses **IAM roles**, **VPC networking**, and **encryption** for security.

1.1.0 : Self Managed kubernetes control Plane :



1.1.1 : Key Challenges:

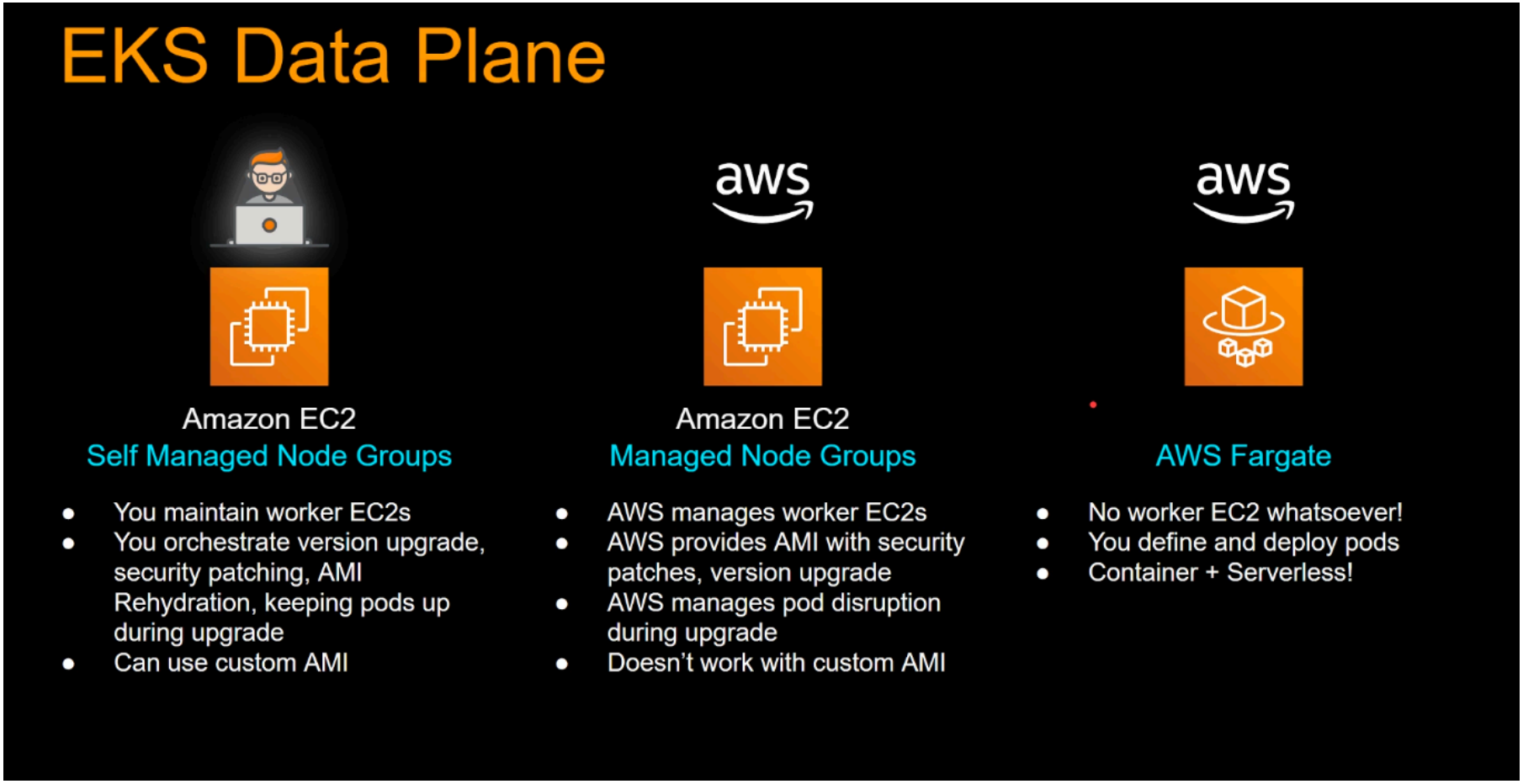
1. **High Availability of Control Plane**
   - Requires multiple EC2 instances across **multiple Availability Zones (AZs)**.
2. **Scaling the Control Plane**
   - Manual intervention needed to scale up/down based on workload demand.
3. **etcd Maintenance**
   - Ensuring **etcd**, the Kubernetes key-value store, is always running and consistent and high available
4. **Operational Overhead of Managing EC2 Instances**
   - **AMI Rehydration**: Keeping Amazon Machine Images (AMIs) updated.
   - **Security Patching**: Regular updates to avoid vulnerabilities.
   - **Replacing Failed EC2s**: Monitoring and replacing unhealthy instances.
   - **Kubernetes Version Upgrades**: Handling upgrades manually across all nodes.

1.2.0 : AWS Managed kubernetes control Plane:

How EKS Solves These Issues:

- **AWS EKS** manages the **Kubernetes control plane**, removing the need for users to handle EC2 instances manually.
- **Automatic High Availability** across multiple AZs.
- **etcd is fully managed** by AWS and highly available.
- **Patching & upgrades** are simplified since AWS maintains the control plane.

1.3.0: EKS Manages Control Plane but what about data planes (worker nodes) ?



When AWS says **"AWS manages pod disruption during upgrades"**, it means:

1️⃣ What Happens During an Upgrade?

==========

- When you **upgrade a Kubernetes version** or apply **security patches**, worker nodes may need to **restart** or be **replaced**.
- This can cause **downtime** for applications if not handled properly.

2️⃣ How AWS Handles It in Managed Node Groups

- AWS **automatically**:
  ✅ **Drains** the node (moves running pods to another node).
  ✅ **Replaces** the node with an upgraded one.
  ✅ **Ensures minimal pod disruption** by doing upgrades **one node at a time** (rolling update).

1.4.0: Comparison between self-managed Kubernetes and EKS?

Aspect	Self-Managed Kubernetes (on EC2)	AWS EKS (Managed)
Control Plane Management	You manage the control plane (master nodes) — setting up, scaling, and maintaining it.	AWS manages the control plane for you. You focus on worker nodes and workloads.
High Availability	Need to configure and maintain multiple EC2 instances across different AZs for redundancy.	Built-in multi-AZ high availability. AWS takes care of it automatically.
etcd Maintenance	You need to ensure etcd is running smoothly and consistently, including backups and restores.	Fully managed by AWS. You don't have to worry about etcd directly.
Scaling	Manually scale the control plane and worker nodes.	Automatic scaling for the control plane; you manage scaling for worker nodes.
Security	Handle security patching for control plane nodes and OS-level patches.	AWS handles security patches for the control plane. You only manage the worker nodes.

Version Upgrades	Manually upgrade Kubernetes versions, including control plane and worker nodes.	Simplified version upgrades. AWS manages the control plane upgrades; you still manage the worker nodes.
Cost	Potentially lower cost but higher operational overhead. You pay for EC2 instances used for control plane.	You pay for EKS ($0.10 per hour per cluster) plus the cost of worker nodes, but save on management time.
Reliability	Depends on your setup and monitoring. Might need more effort to ensure high availability.	Higher reliability due to AWS's SLA and infrastructure management.
Customization	More flexibility in configuring the control plane and underlying infrastructure.	Less control over the control plane, but can customize worker nodes and configurations.
Operational Overhead	High — you handle everything from setup to scaling to maintenance.	Low — AWS manages the complex parts, freeing you up to focus on applications and workloads.

1.5.0: EKS in AWS Ecosystem

Amazon Elastic Kubernetes Service (**EKS**) is **AWS's managed Kubernetes offering** that integrates with various AWS services to simplify container orchestration. Here's how it fits into the AWS ecosystem:

1. Control Plane (Managed by AWS)

✅ AWS manages the **Kubernetes API server, etcd storage, and upgrades**
✅ Highly Available & Auto-Scales across **multiple AZs**

👉 **You don't need to manage the control plane**

2. Data Plane (Worker Nodes - Your Responsibility)

- **Self-Managed Nodes (EC2)** → You provision and manage EC2 instances manually

==========

- **Managed Node Groups (EC2)** → AWS manages EC2 nodes for you
- **AWS Fargate (Serverless)** → No EC2 instances, just run pods

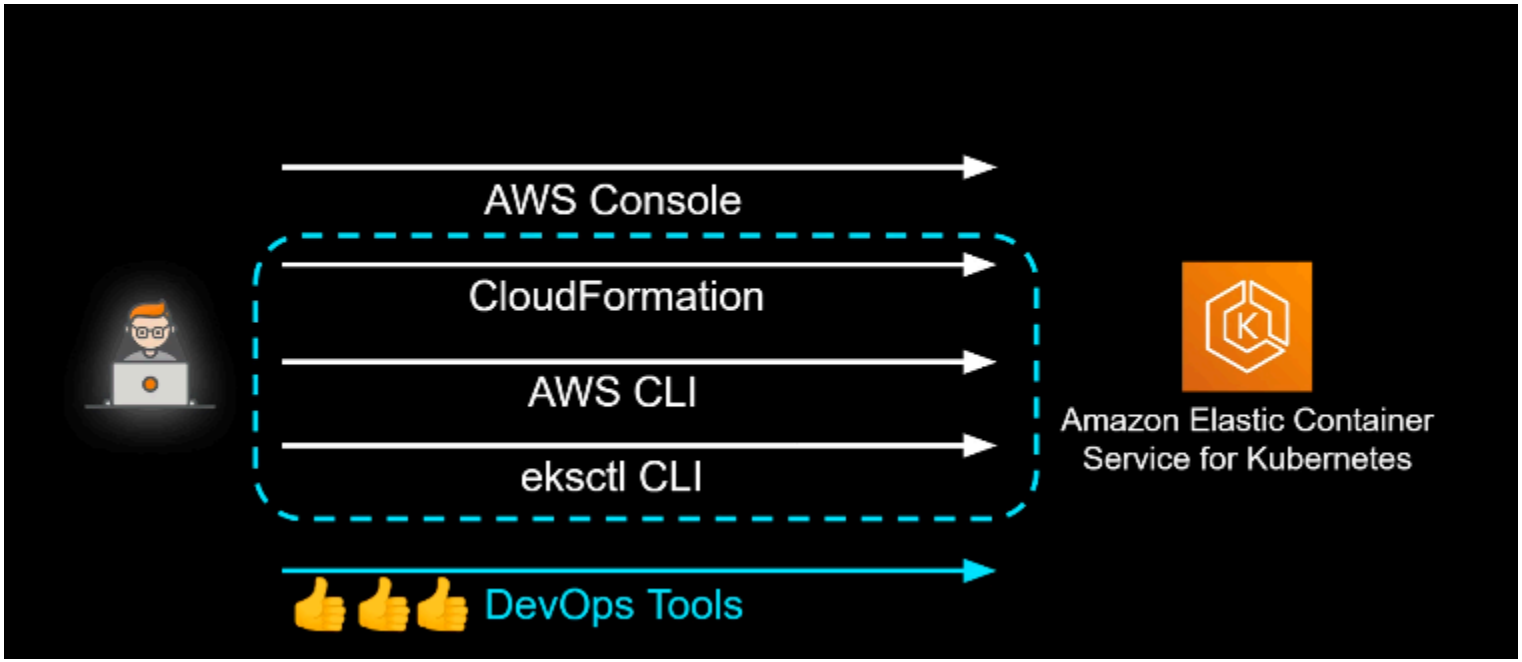
👉 **You choose how to run worker nodes based on cost, control, and automation needs**

3. AWS Services that Integrate with EKS

- EKS works seamlessly with multiple AWS services for networking, security, logging, and storage:

Category	AWS Services	Purpose
Networking	VPC, ALB, NLB, Route 53	Manages pod networking & ingress traffic
Security	IAM, AWS KMS, Secrets Manager	Identity, access control, and secret management
Observability	CloudWatch, AWS X-Ray, Prometheus, Grafana	Logging, monitoring, and tracing
Storage	EBS, EFS, S3	Persistent storage for Kubernetes workloads
Deployment & Scaling	Auto Scaling, EC2 Spot, Lambda	Manages scaling of nodes and workloads
DevOps & CI/CD	CodePipeline, CodeBuild, GitOps (ArgoCD)	Automates application deployments

2.0 : Ways to spin up EKS Cluster ?

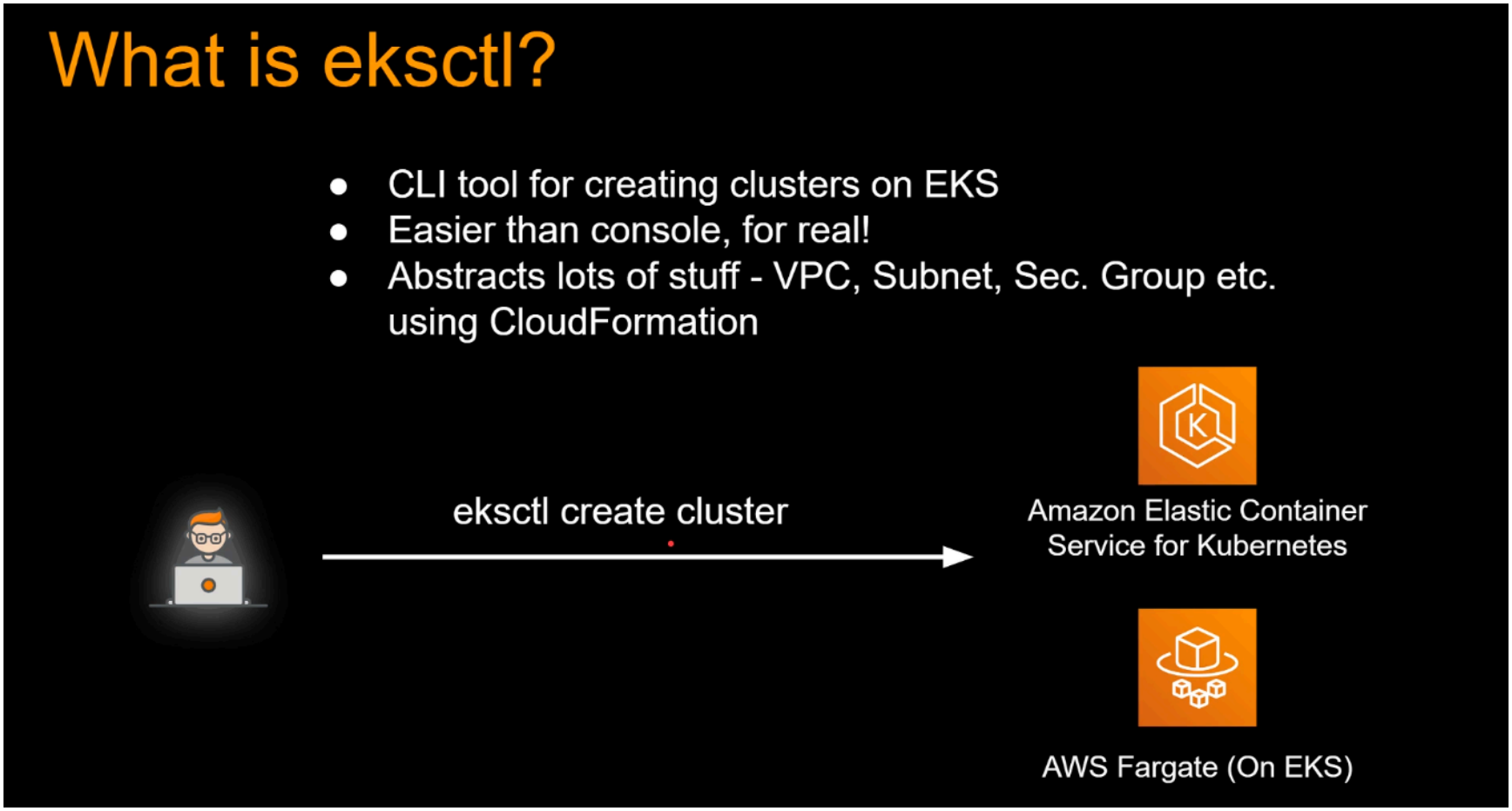


Preferred ways are :

- Eksctl
- Devops tools like, terraform

Each approach has its own advantages, with `eksctl` and `Terraform` being the most popular choices for automating and managing EKS clusters efficiently.

3.0 : What is eksctl ?



Key Features

✅ **Cluster Management:** Easily create, update, and delete EKS clusters.
✅ **Node Group Management:** Supports managed/unmanaged node groups and Fargate.
✅ **IAM Role Automation:** Automatically creates IAM roles and attaches necessary policies.
✅ **VPC and Networking:** Can set up a VPC or use an existing one.
✅ **Add-On Management:** Supports installing AWS-managed add-ons like VPC CNI, CoreDNS, and kube-proxy.
✅ **Declarative Configurations:** Allows defining clusters and node groups in YAML format.
✅ **Spot Instances & Auto-Scaling:** Supports cost-saving strategies with spot instances and auto-scaling node groups.
✅ **CloudFormation Backend**: `eksctl` **uses AWS CloudFormation** internally to create, update, and delete EKS resources.
✅ **Stack Management**: Each `eksctl` operation generates a **CloudFormation stack** that provisions the necessary AWS infrastructure.

3.1 : eksctl Commands

Command	Brief Description
eksctl create cluster	Create EKS Cluster with one nodegroup containing 2 m5.large nodes
eksctl create cluster --name <name> --version 1.15 --node-type t3.micro --nodes 2	Create EKS Cluster with K8 version 1.15 with 2 t3.micro nodes
eksctl create cluster --name <name> --version 1.15 --nodegroup-name <nodegrpname> --node-type t3.micro --nodes 2 --managed	Create EKS cluster with managed node group
eksctl create cluster --name <name> --fargate	EKS Cluster with Fargate Profile

Available eksctl features (Only on EKS)

- Create, get, list and delete clusters
- Create, drain and delete nodegroups
- Scale a nodegroup
- Update a cluster
- Use custom AMIs
- Configure VPC Networking
- Configure access to API endpoints
- Support for GPU nodegroups
- Spot instances and mixed instances
- IAM Management and Add-on Policies
- List cluster Cloudformation stacks
- Install coredns
- Write kubeconfig file for a cluster

4.0 : Configuring Tools :

Operating System I'm using : Ubuntu 22.x

4.1: Installing and configuring aws-cli

```
vishal@vishalk17:~/eks$ snap install aws-cli --classic
aws-cli (v2/stable) 2.23.15 from Amazon Web Services (aws✓) installed

vishal@vishalk17:~/eks$ aws configure
AWS Access Key ID [****************LFES]: ****************LFES
AWS Secret Access Key [****************QlHT]: ****************QlHT
Default region name [ap-southeast-2]: ap-south-1
Default output format [json]: json
```

4.2: Installing eksctl

Follow the instruction from here : https://eksctl.io/installation/

4.3: Installing kubectl

Follow the instruction from here :
https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-binary-with-curl-on-linux

5.0 : EKS Self-Managed Nodes vs EKS Managed Nodes :

EKS Self-Managed Nodes:

- You could have **hundreds of nodes**.
- Each node **requires manual patching** for security updates, Kubernetes version upgrades, and AMI updates, which can be time-consuming and error-prone.
- **Custom AMIs** need to be created and maintained by you.
- During updates:
    - Kubernetes will **terminate pods** running on the node being updated.
    - This can cause **application downtime** unless you manually configure high availability.
    - You must **provision new nodes**, then **manually spin up pods** and applications on them to maintain availability.

Eks managed node group solve the problem :

1. **Create & Manage EC2 Workers for You:**
    - EKS Managed Node Groups automatically create and manage EC2 instances (worker nodes) for your Kubernetes cluster. You don't need to manually provision or manage the underlying infrastructure.

2. **Worker Node Kubernetes Version Updated with One Click/API Call:**
   You can update the Kubernetes version of worker nodes in a managed node group with a single action (via the **AWS Console**, **CLI**, or **API**).

   Note: If upgrading to a newer Kubernetes version, you must update the **control plane** first.

3. **Amazon Releases AMIs with Bug Fixes and Security Patches:**
   AWS regularly releases updated **Amazon Machine Images (AMIs)** for EKS worker nodes, including bug fixes, security patches, and updates.
    - You can choose to:
        - Use AWS-managed AMIs, **or**
        - Bring your own **custom AMIs**.

4. **Managed Node Groups Use Auto Scaling Groups (ASGs):**

Managed Node Groups are backed by **Auto Scaling Groups (ASGs)**, which manage the lifecycle of worker nodes, including:

- **Scaling** up/down,
- **Replacing unhealthy nodes**,
- Ensuring **high availability**.

5. **AWS Orchestrates Node Replacement While Respecting Pod Disruption Budget (PDB):**

   When replacing nodes (e.g., during updates or scaling), AWS ensures that:

   - Pods are **gracefully moved** to new nodes before terminating the old ones.
   - The process respects **Pod Disruption Budgets (PDBs)** to maintain application availability.

6. **Updated AMIs with Security Patches and CVEs (Manual Deployment Required):**

   AWS provides updated AMIs with security patches and **CVE** fixes.

   - **However, these updates are not automatically deployed** to your managed node groups.
   - You must manually trigger **rolling updates** using the AWS Console, CLI, or automation tools like `eksctl` or CI/CD pipelines.

7. **Minimized Downtime (Not Guaranteed Zero Downtime):**

   Managed Node Groups aim to **minimize downtime** through graceful node replacements and PDB enforcement.

   - **However, true zero downtime is not guaranteed**, as it depends on your:
     - Application **architecture**,
     - **High availability** setup (replica configurations),
     - Correctly configured **PDBs**.

8. **No Overhead of User-Managed Orchestration:**
   AWS handles node:
   - **Provisioning**,
   - **Scaling**,
   - **Updates**,

- **Replacements**,
  reducing the operational burden on you.

9. **Auto Scaling Groups (ASGs) Are Used Behind the Scenes:**
   Managed Node Groups are built on top of **ASGs**, which handle:
   - **Scaling**,
   - **Health checks**,
   - **Lifecycle management** behind the scenes.

6.0 : EC2 instance type and Pod Limit :

1. Max. No. of allowed pods depends on ec2 instance type
2. Bigger the instance type , more pods

Ref. : https://docs.aws.amazon.com/eks/latest/userguide/choosing-instance-type.html

7.0: Spin up our first eks cluster :

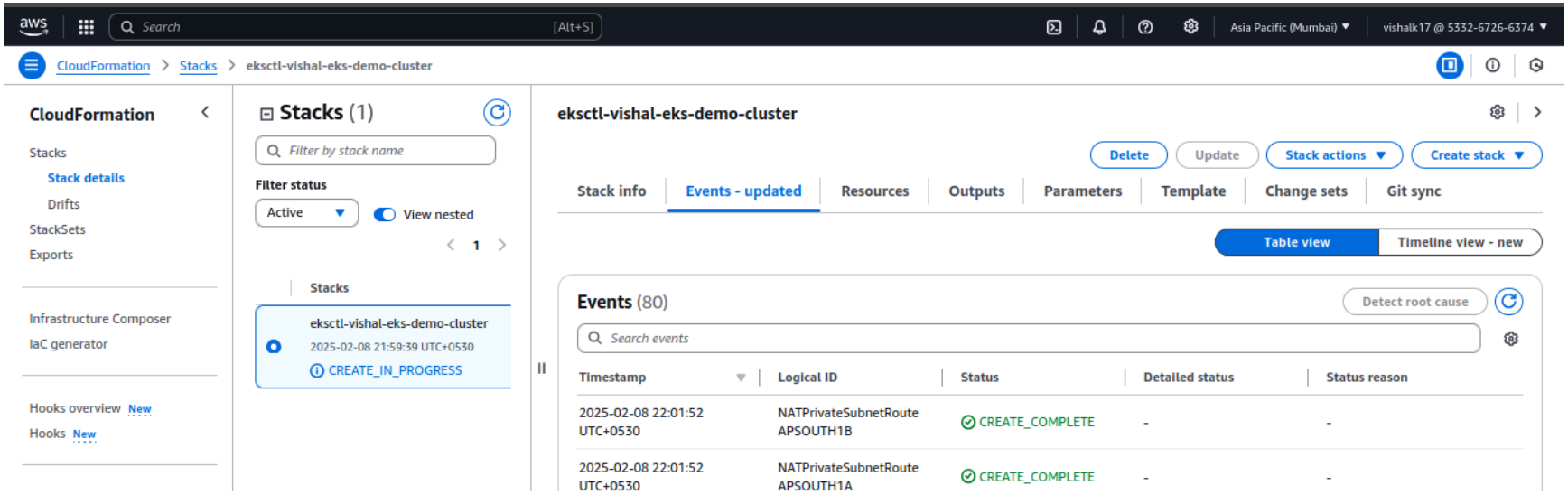
Command	Brief Description
eksctl create cluster	Create EKS Cluster with one nodegroup containing 2 m5.large nodes
eksctl create cluster --name <name> --version 1.15 --node-type t3.micro --nodes 2	Create EKS Cluster with K8 version 1.15 with 2 t3.micro nodes
eksctl create cluster --name <name> --version 1.15 --nodegroup-name <nodegrpname> --node-type t3.micro --nodes 2 --managed	Create EKS cluster with managed node group
eksctl create cluster --name <name> --fargate	EKS Cluster with Fargate Profile

7.1: Eksctl cluster create using command :

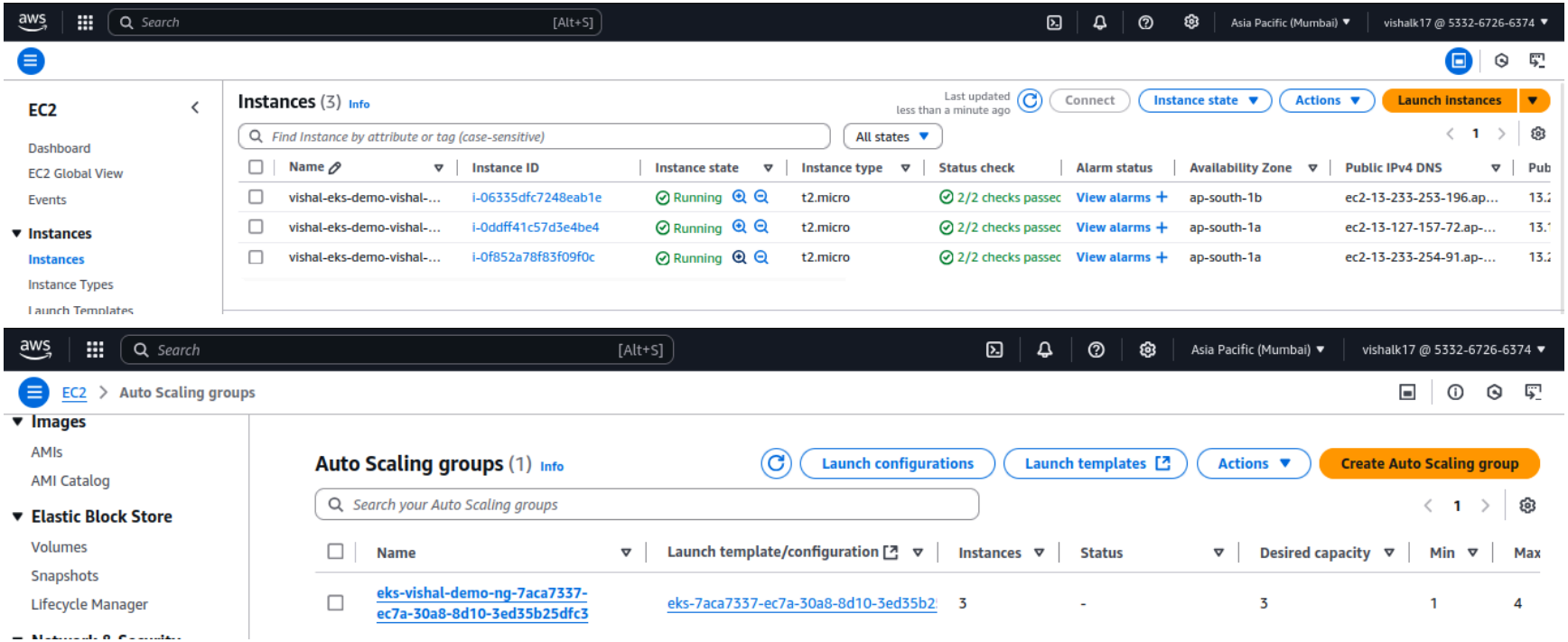
```
vishal@vishalk17:~/Downloads/vish$ eksctl create cluster --name vishal-eks-demo --region ap-south-1
--node-type t2.micro --nodes 3 --nodes-max=4 --nodes-min 1 --nodegroup-name=vishal-demo-ng --managed
2025-02-08 21:59:38 [i]  eksctl version 0.189.0
2025-02-08 21:59:38 [i]  using region ap-south-1
2025-02-08 21:59:39 [i]  skipping ap-south-1c from selection because it doesn't support the following
instance type(s): t2.micro
```

- The cluster will be created in 10-15 minutes.

Meanwhile you can see progress in cloudformation :




```
2025-02-08 22:13:55 [✔]  created 1 managed nodegroup(s) in cluster "vishal-eks-demo"
2025-02-08 22:13:55 [✖]  kubectl not found, v1.10.0 or newer is required
2025-02-08 22:13:55 [ℹ]  cluster should be functional despite missing (or misconfigured) client binaries
2025-02-08 22:13:55 [✔]  EKS cluster "vishal-eks-demo" in "ap-south-1" region is ready
vishal@vishalk17:~/Downloads/vish$ kubectl
Command 'kubectl' not found, but can be installed with:
sudo snap install kubectl
vishal@vishalk17:~/Downloads/vish$ sudo snap install kubectl
[sudo] password for vishal:
error: This revision of snap "kubectl" was published using classic confinement and thus may perform
       arbitrary system changes outside of the security sandbox that snaps are usually confined to,
       which may put your system at risk.

       If you understand and want to proceed repeat the command including --classic.
vishal@vishalk17:~/Downloads/vish$ sudo snap install kubectl --classic
kubectl 1.31.5 from Canonical✓ installed
vishal@vishalk17:~/Downloads/vish$ kubectl get pods
No resources found in default namespace.
vishal@vishalk17:~/Downloads/vish$ kubectl get nodes -o wide
NAME                                          STATUS   ROLES    AGE     VERSION               INTERNAL-IP      EXTERNAL-IP      OS-IMAGE         KERNEL-VERSION
CONTAINER-RUNTIME
ip-192-168-20-150.ap-south-1.compute.internal   Ready    <none>   6m47s   v1.30.8-eks-aeac579   192.168.20.150   13.127.157.72    Amazon Linux 2   5.10.233-223.887.amzn2.x
containerd://1.7.25
ip-192-168-21-193.ap-south-1.compute.internal   Ready    <none>   6m43s   v1.30.8-eks-aeac579   192.168.21.193   13.233.254.91    Amazon Linux 2   5.10.233-223.887.amzn2.x
containerd://1.7.25
ip-192-168-50-196.ap-south-1.compute.internal   Ready    <none>   6m53s   v1.30.8-eks-aeac579   192.168.50.196   13.233.253.196   Amazon Linux 2   5.10.233-223.887.amzn2.x
containerd://1.7.25
vishal@vishalk17:~/Downloads/vish$
```



Eksctl get cluster :

```
vishal@vishalk17:~/Downloads/vish$ eksctl get cluster
NAME            REGION          EKSCTL CREATED
vishal-eks-demo ap-south-1      True
```

Delete eks Cluster :

```
vishal@vishalk17:~/Downloads/vish$ eksctl delete cluster --name vishal-eks-demo --region ap-south-1
2025-02-08 22:24:57 [ℹ]  deleting EKS cluster "vishal-eks-demo"
2025-02-08 22:24:58 [ℹ]  will drain 0 unmanaged nodegroup(s) in cluster "vishal-eks-demo"
2025-02-08 22:24:58 [ℹ]  starting parallel draining, max in-flight of 1
2025-02-08 22:24:58 [✖]  failed to acquire semaphore while waiting for all routines to finish: context canceled
2025-02-08 22:24:58 [ℹ]  deleted 0 Fargate profile(s)
2025-02-08 22:25:00 [✔]  kubeconfig has been updated
2025-02-08 22:25:00 [ℹ]  cleaning up AWS load balancers created by Kubernetes objects of Kind Service or Ingress
2025-02-08 22:25:02 [ℹ]
2 sequential tasks: { delete nodegroup "vishal-demo-ng", delete cluster control plane "vishal-eks-demo" [async]
}
2025-02-08 22:25:02 [ℹ]  will delete stack "eksctl-vishal-eks-demo-nodegroup-vishal-demo-ng"
2025-02-08 22:25:02 [ℹ]  waiting for stack "eksctl-vishal-eks-demo-nodegroup-vishal-demo-ng" to get deleted
2025-02-08 22:25:02 [ℹ]  waiting for CloudFormation stack "eksctl-vishal-eks-demo-nodegroup-vishal-demo-ng"
```

This will **delete** your EKS cluster and **all associated resources** (such as EC2 instances in the node group, VPC, etc.). Be careful when running this command.

7.2: EKS Cluster and Node Group Creation Using eksctl :

config-file.yaml

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-cluster-demo          # Name of the EKS cluster
  region: ap-south-1             # AWS region
  version: "1.28"                # Kubernetes version

managedNodeGroups:
  - name: my-cluster-demo-ng     # Node group name
    instanceType: t3.micro       # EC2 instance type
    desiredCapacity: 2           # Number of nodes
    minSize: 1                   # Minimum nodes
```

==========

```
    maxSize: 3                    # Maximum nodes
    volumeSize: 20                # Disk size in GB
    ssh:
      allow: true                 # Enable SSH access
      publicKeyName: vish-console     # Name of the EC2 key pair
```

Create Cluster :

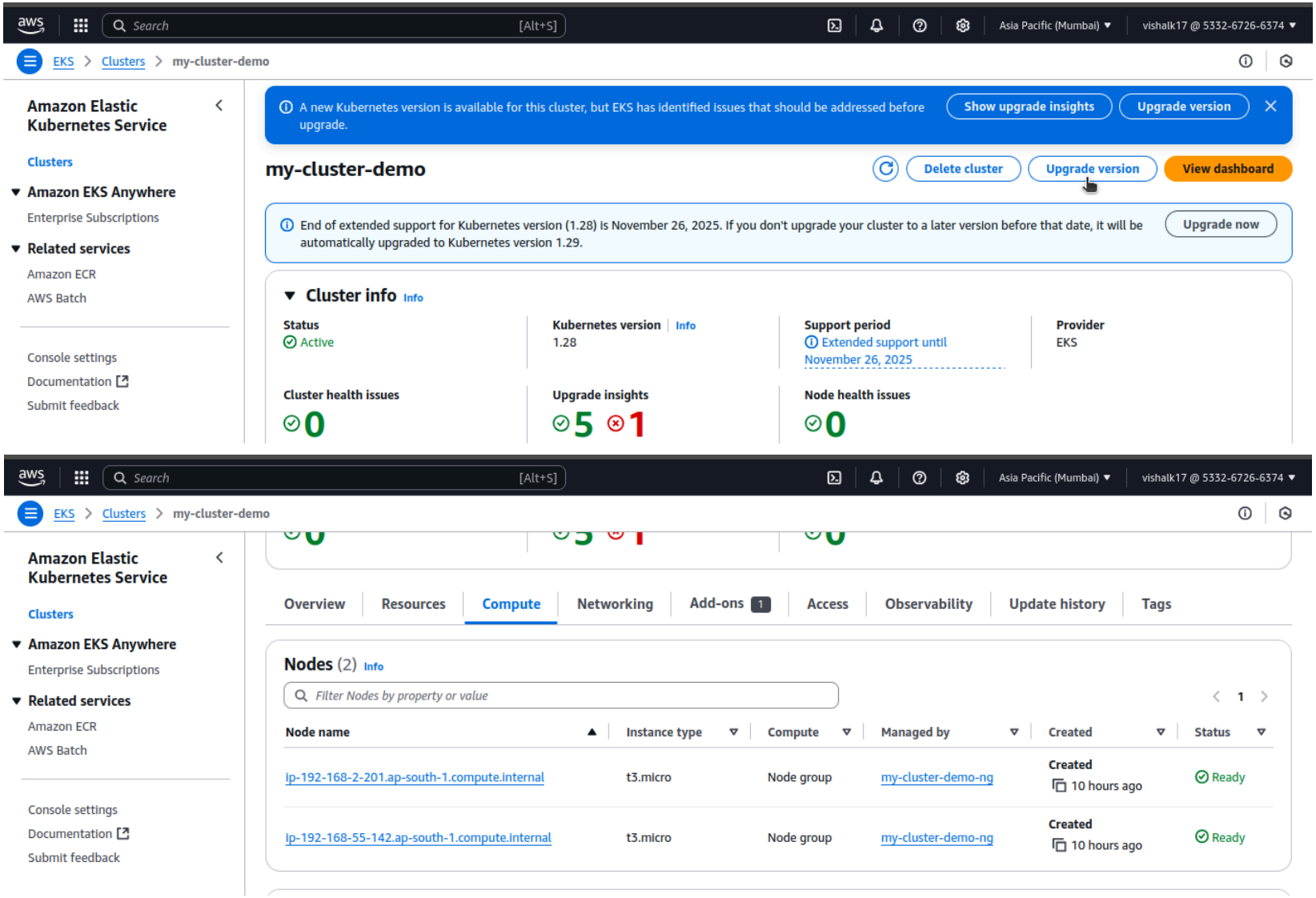
```
vishal@vishalk17:~/Downloads/vish$ ls
config-file.yaml
vishal@vishalk17:~/Downloads/vish$ eksctl create cluster -f config-file.yaml
2025-02-09 00:44:12 [ℹ]  eksctl version 0.189.0
2025-02-09 00:44:12 [ℹ]  using region ap-south-1
2025-02-09 00:44:12 [ℹ]  setting availability zones to [ap-south-1a ap-south-1b ap-south-1c]
2025-02-09 00:44:12 [ℹ]  subnets for ap-south-1a - public:192.168.0.0/19 private:192.168.96.0/19
2025-02-09 00:44:12 [ℹ]  subnets for ap-south-1b - public:192.168.32.0/19 private:192.168.128.0/19
2025-02-09 00:44:12 [ℹ]  subnets for ap-south-1c - public:192.168.64.0/19 private:192.168.160.0/19
2025-02-09 00:44:12 [ℹ]  nodegroup "my-cluster-demo-ng" will use "" [AmazonLinux2/1.28]
2025-02-09 00:44:12 [ℹ]  using EC2 key pair "vish-console"
2025-02-09 00:44:12 [ℹ]  using Kubernetes version 1.28
2025-02-09 00:44:12 [ℹ]  creating EKS cluster "my-cluster-demo" in "ap-south-1" region with managed nodes
2025-02-09 00:44:12 [ℹ]  1 nodegroup (my-cluster-demo-ng) was included (based on the include/exclude rules)
```

Delete cluster :

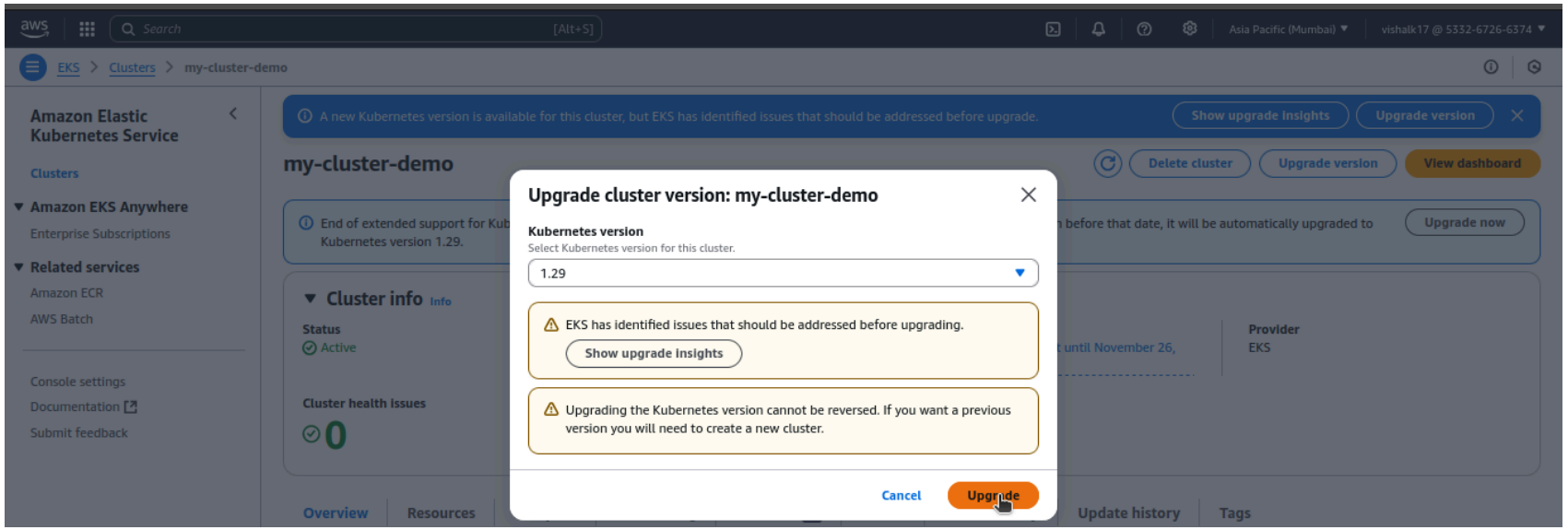
```
vishal@vishalk17:~/Downloads/vish$ eksctl delete cluster -f config-file.yaml
2025-02-09 01:00:16 [ℹ]  deleting EKS cluster "my-cluster-demo"
2025-02-09 01:00:17 [ℹ]  will drain 0 unmanaged nodegroup(s) in cluster "my-cluster-demo"
2025-02-09 01:00:17 [ℹ]  starting parallel draining, max in-flight of 1
2025-02-09 01:00:17 [✖]  failed to acquire semaphore while waiting for all routines to finish: context canceled
2025-02-09 01:00:17 [ℹ]  deleted 0 Fargate profile(s)
2025-02-09 01:00:18 [✔]  kubeconfig has been updated
2025-02-09 01:00:18 [ℹ]  cleaning up AWS load balancers created by Kubernetes objects of Kind Service or Ingress
2025-02-09 01:00:19 [ℹ]
2 sequential tasks: { delete nodegroup "my-cluster-demo-ng", delete cluster control plane "my-cluster-demo" [async]
}
2025-02-09 01:00:19 [ℹ]  will delete stack "eksctl-my-cluster-demo-nodegroup-my-cluster-demo-ng"
2025-02-09 01:00:19 [ℹ]  waiting for stack "eksctl-my-cluster-demo-nodegroup-my-cluster-demo-ng" to get deleted
2025-02-09 01:00:19 [ℹ]  waiting for CloudFormation stack "eksctl-my-cluster-demo-nodegroup-my-cluster-demo-ng"
```

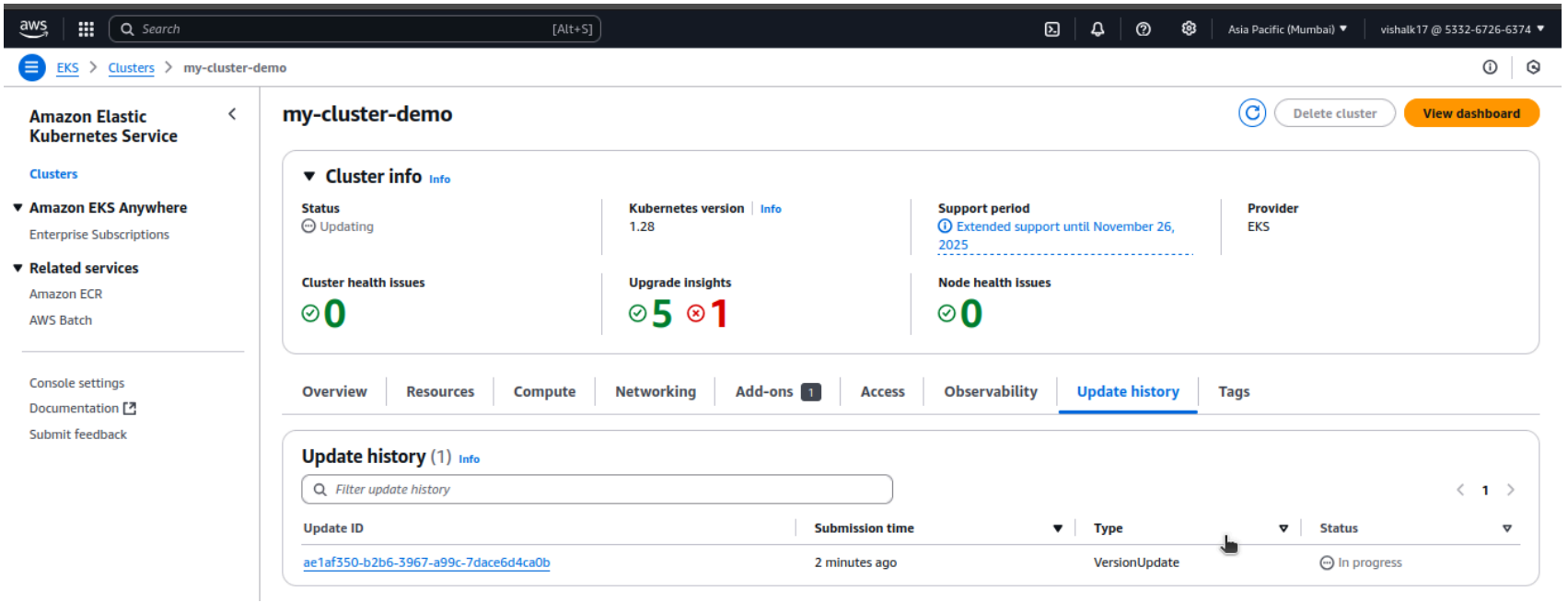
==========

7.3: EKS update :

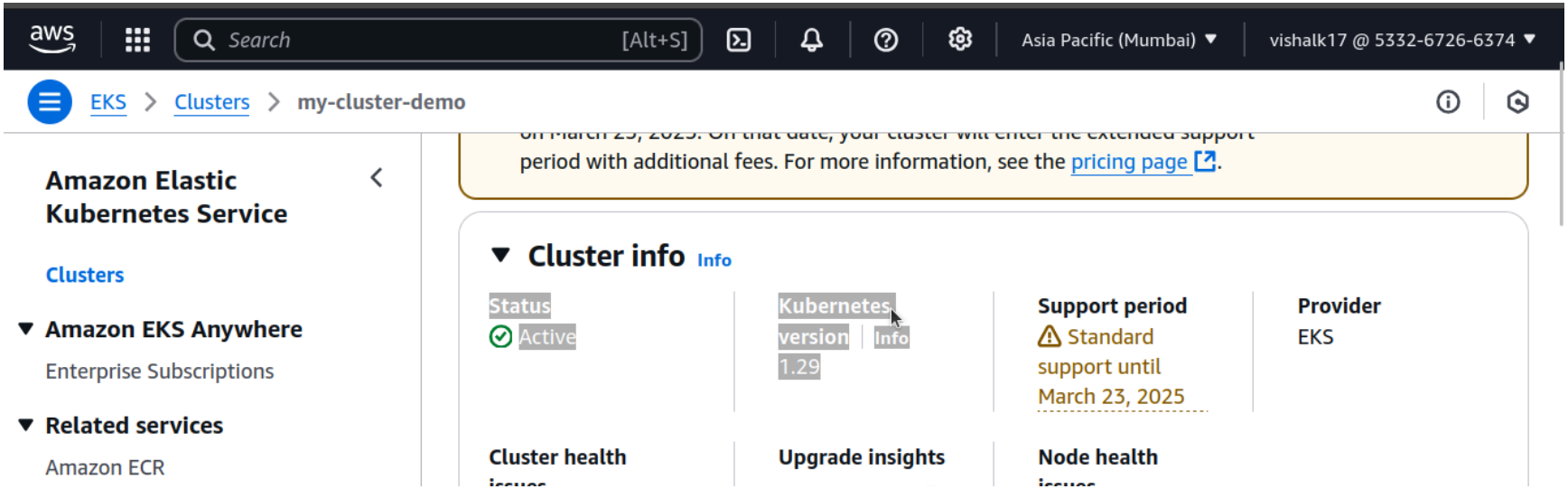


- **Update cluster to 1.29**

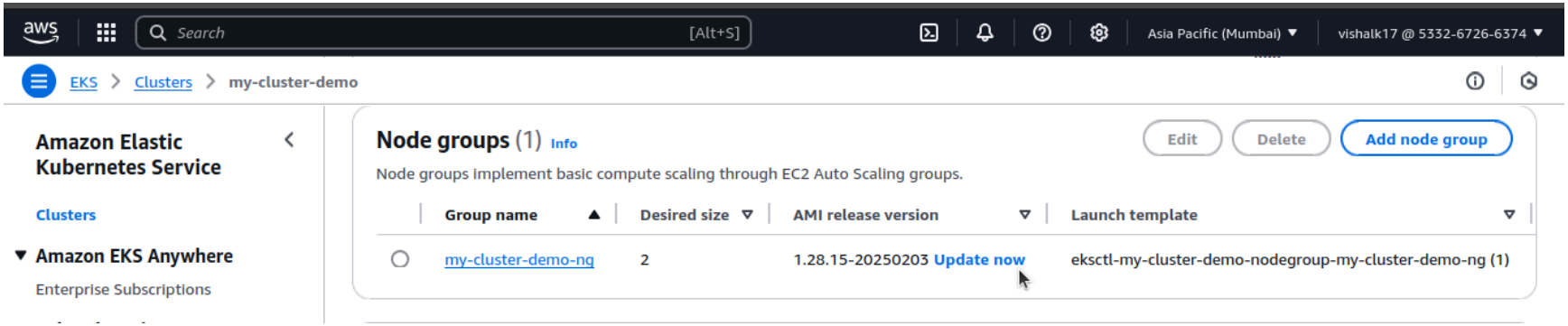

Upgrade cluster version: my-cluster-demo
Kubernetes version
Select Kubernetes version for this cluster.
1.29
EKS has identified issues that should be addressed before upgrading.
Show upgrade insights
Upgrading the Kubernetes version cannot be reversed. If you want a previous version you will need to create a new cluster.
Cancel
Upgrade
my-cluster-demo
Cluster info
Overview
Resources
Update history
Tags

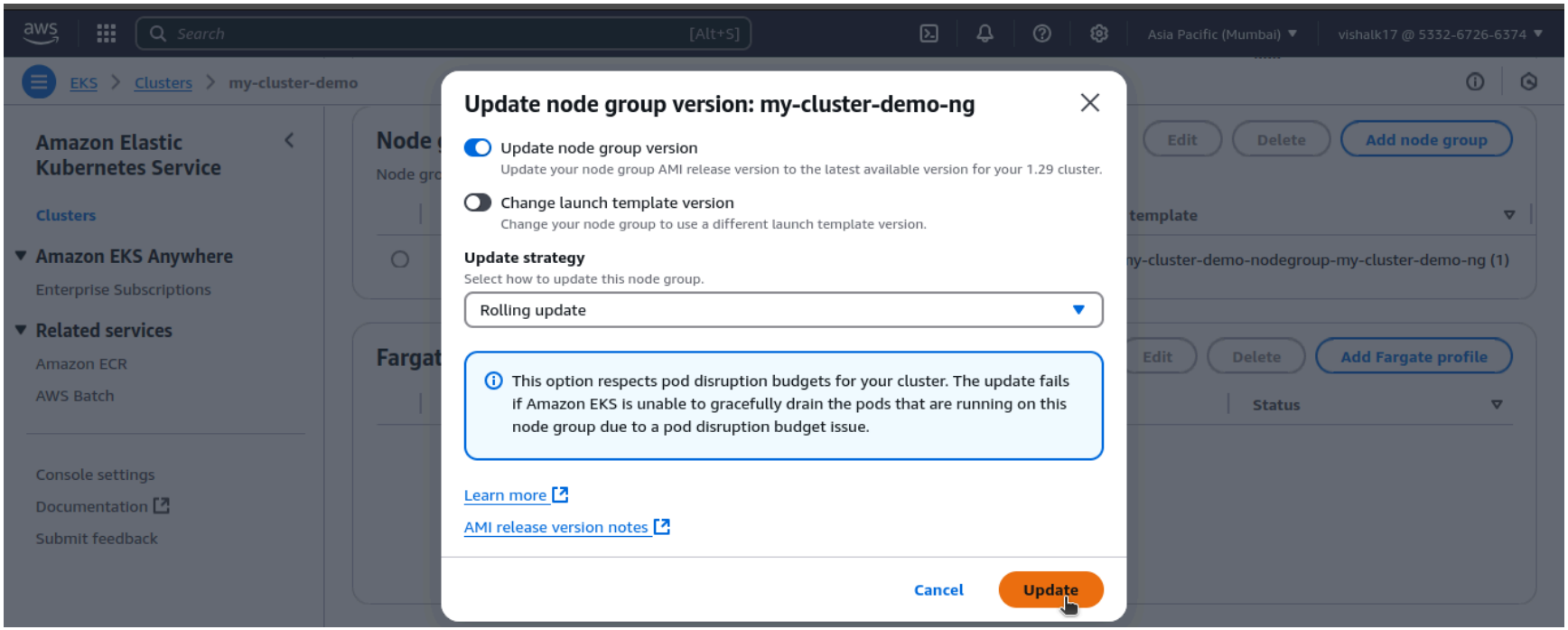

my-cluster-demo
Delete cluster
View dashboard
Cluster info
Status
Updating
Kubernetes version
1.28
Support period
Extended support until November 26, 2025
Provider
EKS
Cluster health issues
0
Upgrade insights
5 1
Node health issues
0
Overview
Resources
Compute
Networking
Add-ons 1
Access
Observability
Update history
Tags
Update history (1)
Filter update history
Update ID
Submission time
Type
Status
ae1af350-b2b6-3967-a99c-7dace6d4ca0b
2 minutes ago
VersionUpdate
In progress

Cluster Updated to 1.29



- **Update Nodegroup**






```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE   VERSION
ip-192-168-2-201.ap-south-1.compute.internal   Ready    <none>   10h   v1.28.15-eks-aeac579
ip-192-168-55-142.ap-south-1.compute.internal  Ready    <none>   10h   v1.28.15-eks-aeac579
ip-192-168-84-100.ap-south-1.compute.internal  Ready    <none>   14m   v1.29.12-eks-aeac579
ip-192-168-85-81.ap-south-1.compute.internal   Ready    <none>   14m   v1.29.12-eks-aeac579
vishal@vishalk17:~/Downloads/vish/eks$
```

8.0.0 : Kubernetes Scaling :

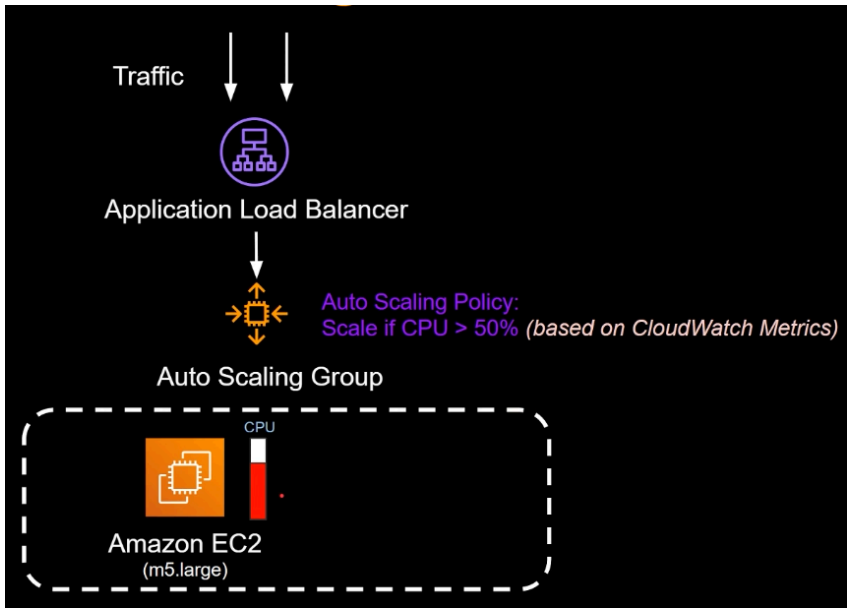
8.0.1: Kubernetes Scaling - HPA, Pod Requests, Limits :

Container Scaling :

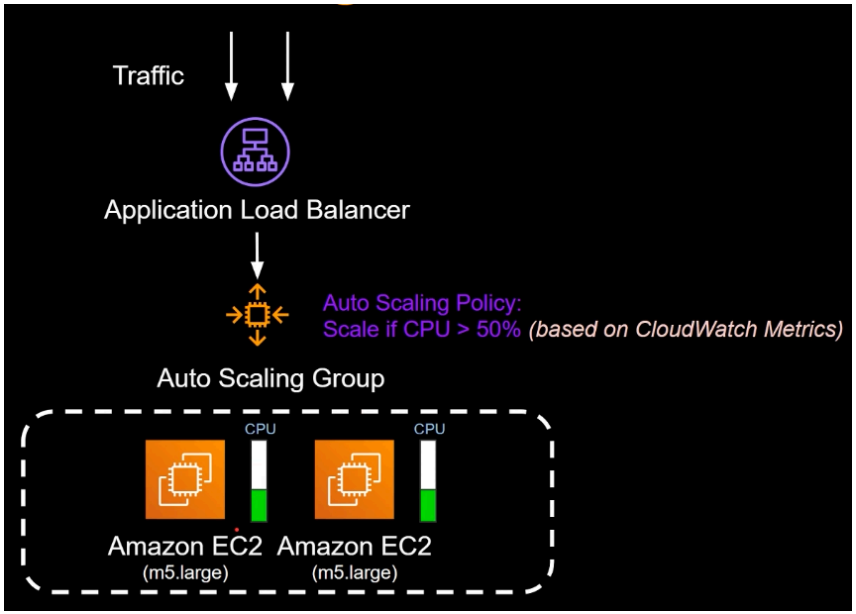
- Quick look into ec2 scaling
- Pod scaling
- Understand pod limits and Request
- Horizontal pod autoscaler
- Understand manifest file
- HPA Demo

A. Quick look into ec2 scaling :

< Traffic Increases >

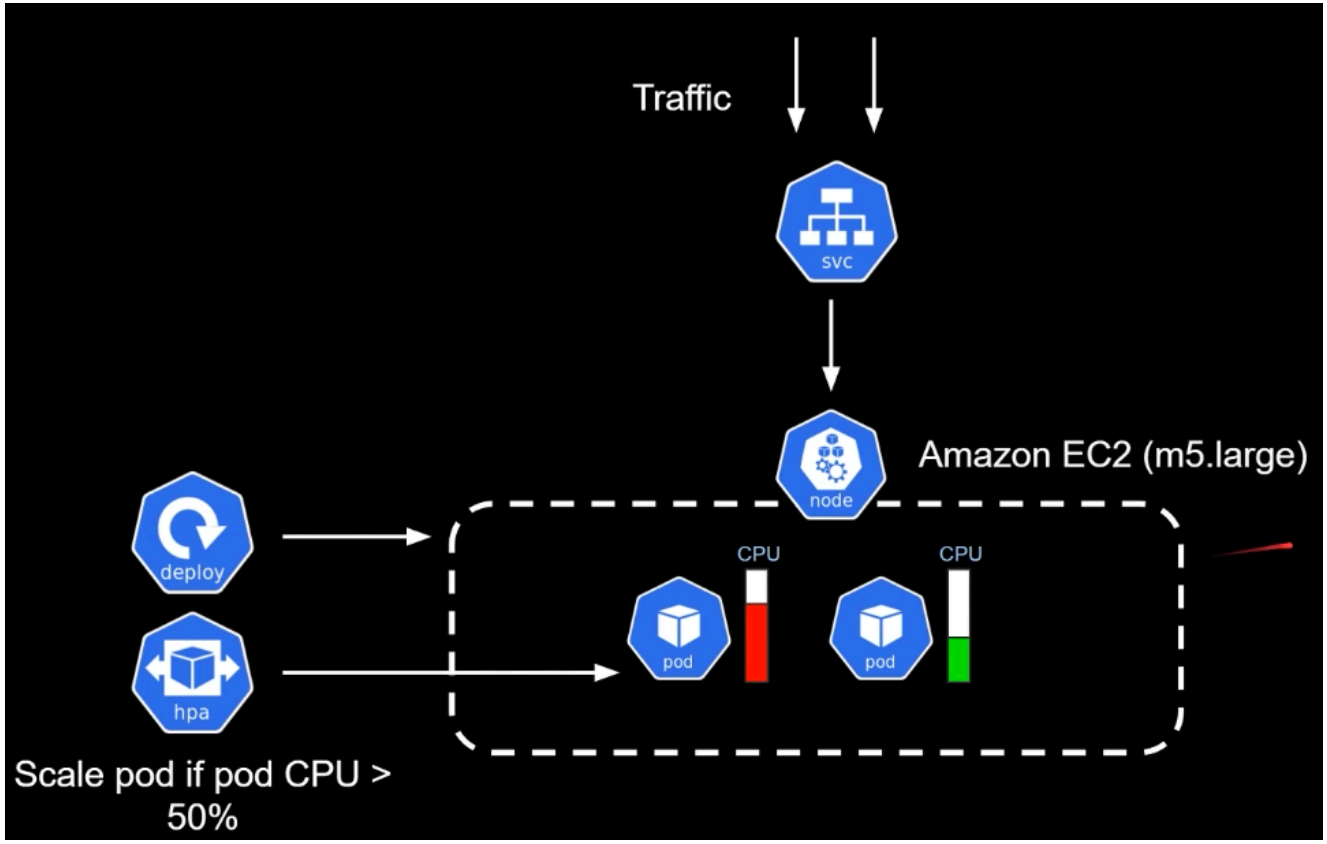


< spinup new ec2 and distribute equal traffic >



B.1 Pod Scaling :

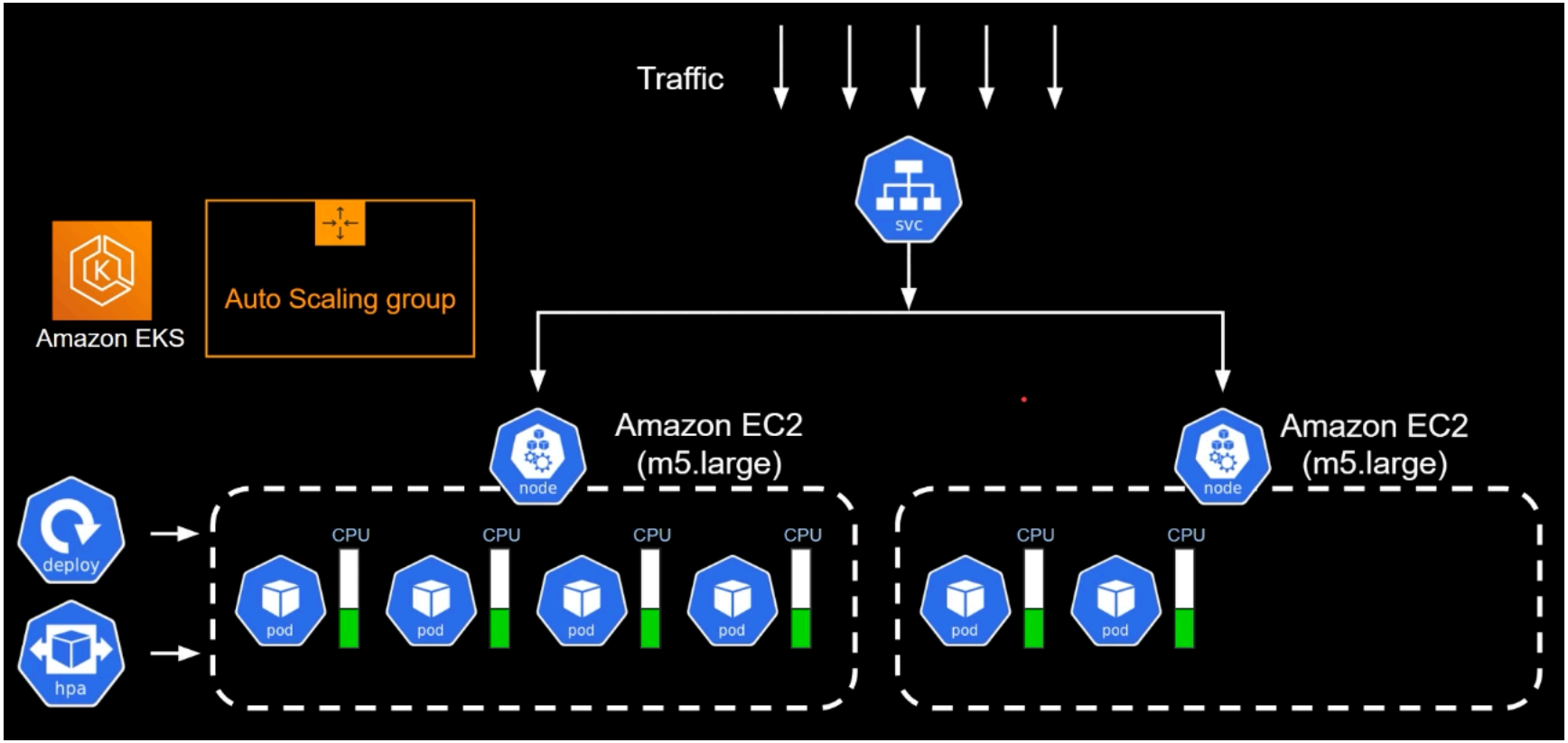
- If traffic goes beyond threshold it will create new pod because of autosale policy we define at pod level



B.2 EKS Cluster Scaling :

But what if pod scales and don't have enough enough resources to spin up new pod in case of managed nodegroups :

- Well answer is quite simple , it will gonna spin up new node at cluster level and then schedule pod as per autoscaling policy defined.



C. Understand pod limits and Request , manifest file :

In Kubernetes, resource management is crucial for ensuring that applications run efficiently and reliably. Understanding how CPU and memory limits work is essential for managing your pods effectively. Here's a breakdown of how Kubernetes handles CPU and memory limits:

```
#Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 512Mi
```

CPU Limits

1. **CPU Requests and Limits**:
    - **Request**: This is the amount of CPU that Kubernetes guarantees to the container. If the node has enough resources, the container will always get at least this amount of CPU.
    - **Limit**: This is the maximum amount of CPU that the container can use. If the container tries to use more than this limit, Kubernetes will throttle the CPU usage.
2. **Throttling**:
    - When a container exceeds its CPU limit, Kubernetes does not terminate the container. Instead, it throttles the CPU usage, meaning that the container will be restricted to the specified limit. This can lead to performance degradation, as the application may not be able to process requests as quickly as it would like.

Memory Limits

1. **Memory Requests and Limits**:
    - **Request**: Similar to CPU, this is the amount of memory that Kubernetes guarantees to the container.
    - **Limit**: This is the maximum amount of memory that the container can use.
2. **Out of Memory (OOM) Handling**:
    - If a container exceeds its memory limit, Kubernetes will take action to protect the node and other containers running on it. Specifically, it will kill the container that exceeds its memory limit.
    - When a container is killed due to exceeding its memory limit, Kubernetes will typically restart the container, depending on the restart policy defined for the pod.

Summary

- **CPU**: If a pod exceeds its CPU limit, Kubernetes throttles the application, which can lead to slower performance but does not terminate the container.

- **Memory**: If a pod exceeds its memory limit, Kubernetes kills the container and may restart it, depending on the pod's restart policy.

D. HPA Demo :

- Install metric server
- Deploy deployment , service , HPA
- Increase Load
- Pod Scale

Install Metric Server :

https://github.com/kubernetes-sigs/metrics-server

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

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created

vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods -n kube-system
NAME                              READY   STATUS    RESTARTS   AGE
aws-node-92784                    2/2     Running   0          31m
```

```
aws-node-tzb55                  2/2     Running   0          31m
coredns-56b8d964f7-9m449        1/1     Running   0          34m
coredns-56b8d964f7-bnv4p        1/1     Running   0          34m
kube-proxy-flvxc                1/1     Running   0          31m
kube-proxy-x9qgd                1/1     Running   0          31m
metrics-server-75bf97fcc9-zgggd 1/1     Running   0          89s
```

Deploy :

hpa-demo.yaml :

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

```
#Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
```

```
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 512Mi

---
# Horizontal Pod Autoscaler (HPA)
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/php-apache-598b474864-v4nw8     1/1     Running   0          98s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.100.0.1     <none>        443/TCP   23m
service/php-apache   ClusterIP   10.100.76.93   <none>        80/TCP    98s

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/php-apache   1/1     1            1           98s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/php-apache-598b474864   1         1         1       98s

NAME                                                  REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/php-apache   Deployment/php-apache   0%/50%    1         10        1          98s
```

Increase the load :

Next, see how the autoscaler reacts to increased load. To do this, you'll start a different Pod to act as a client. The container within the client Pod runs in an infinite loop, sending queries to the php-apache service.

```
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0
.01; do wget -q -O- http://php-apache; done"
If you don't see a command prompt, try pressing enter.
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
```

```
vishal@vishalk17:~/docker$ kubectl get hpa
NAME         REFERENCE               TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%     1         10        1          8m13s
vishal@vishalk17:~/docker$ kubectl get hpa
NAME         REFERENCE               TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   250%/50%   1         10        4          8m48s
vishal@vishalk17:~/docker$
```

No. of replica scaled :

```
vishal@vishalk17:~/docker$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
php-apache-598b474864-v4nw8   1/1     Running   0          7m28s
vishal@vishalk17:~/docker$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
load-generator                1/1     Running   0          69s
php-apache-598b474864-7n4v6   1/1     Running   0          12s
php-apache-598b474864-ds8wt   1/1     Running   0          27s
php-apache-598b474864-fwtpt   1/1     Running   0          27s
php-apache-598b474864-sgl86   1/1     Running   0          27s
php-apache-598b474864-v4nw8   1/1     Running   0          8m58s
vishal@vishalk17:~/docker$
```

E. EKS Cluster AutoScaling using Autoscaler :

It has 2 components :

- Open source autoscaler
- Eks implementation [Autoscaling Group (ASG) , I'am]

In EKS, we define scaling parameters using `minSize`, `maxSize`, and `desiredCapacity` for the node group. However, **EKS won't automatically scale the nodes unless the Cluster Autoscaler is deployed and properly configured.**

🚀 Why Does the Cluster Autoscaler Need to Be Deployed in EKS?

1. **`eksctl` Config Defines Capacity Limits, Not Autoscaling Logic**
   - The parameters `minSize: 1`, `maxSize: 4`, and `desiredCapacity: 2` only set the boundaries for scaling.
   - This configuration defines the *limits* but doesn't trigger scaling based on workload demands.
   - **Cluster Autoscaler** is required to monitor the cluster's resource utilization and pending pods to initiate scaling actions dynamically.
2. **Missing Cluster Autoscaler Means No Scaling Trigger**
   - Even though the node group allows scaling, without the Cluster Autoscaler, **EKS has no mechanism to detect when to scale up or down**.
   - The autoscaler continuously checks for pods that cannot be scheduled due to insufficient resources and then adjusts the number of nodes accordingly.

Create I am policy :

`cluster-autoscaler-policy.json`

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

==========

```
      "Effect": "Allow",
      "Action": [
        "autoscaling:DescribeAutoScalingGroups",
        "autoscaling:DescribeAutoScalingInstances",
        "autoscaling:DescribeLaunchConfigurations",
        "autoscaling:DescribeScalingActivities",
        "ec2:DescribeImages",
        "ec2:DescribeInstanceTypes",
        "ec2:DescribeLaunchTemplateVersions",
        "ec2:GetInstanceTypesFromInstanceRequirements",
        "eks:DescribeNodegroup"
      ],
      "Resource": ["*"]
    },
    {
      "Effect": "Allow",
      "Action": [
        "autoscaling:SetDesiredCapacity",
        "autoscaling:TerminateInstanceInAutoScalingGroup"
      ],
      "Resource": ["*"]
    }
  ]
}
```

```
aws iam create-policy \
  --policy-name CustomEKSClusterAutoscalerPolicy \
  --policy-document file://cluster-autoscaler-policy.json
```

==========

```
vishal@vishalk17:~/Downloads/vish/eks$ aws iam create-policy \
  --policy-name CustomEKSClusterAutoscalerPolicy \
  --policy-document file://cluster-autoscaler-policy.json
{
    "Policy": {
        "PolicyName": "CustomEKSClusterAutoscalerPolicy",
        "PolicyId": "ANPAXYKJUXNDKYJEO56VQ",
        "Arn": "arn:aws:iam::533267266374:policy/CustomEKSClusterAutoscalerPolicy",
        "Path": "/",
        "DefaultVersionId": "v1",
        "AttachmentCount": 0,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "CreateDate": "2025-02-12T18:54:44+00:00",
        "UpdateDate": "2025-02-12T18:54:44+00:00"
    }
}
```

Note down : ARN of policy

=========

Get nodeGroup name we are using :

```
vishal@vishalk17:~/Downloads/vish/eks$ eksctl get nodegroup --cluster my-cluster-demo
CLUSTER         NODEGROUP               STATUS  CREATED
my-cluster-demo my-cluster-demo-ng      ACTIVE  2025-02-12T16:57:18Z
```

==========

Get nodeGroup ARN we are using :

```
vishal@vishalk17:~/Downloads/vish/eks$ aws eks describe-nodegroup --cluster-name my-cluster-demo
--nodegroup-name my-cluster-demo-ng --query "nodegroup.nodeRole" --output text
arn:aws:iam::533267266374:role/eksctl-my-cluster-demo-nodegroup-m-NodeInstanceRole-pmyOEz976TZh
```

Note down : role name , here it is : `eksctl-my-cluster-demo-nodegroup-m-NodeInstanceRole-pmyOEz976TZh`

==========

Add `CustomEKSClusterAutoscalerPolicy` to the role `<node-instance-role>`

```
aws iam attach-role-policy \
  --role-name <node-instance-role> \
  --policy-arn arn:aws:iam::<account-id>:policy/CustomEKSClusterAutoscalerPolicy
```

Replace place holder and apply this command

```
aws iam attach-role-policy \
  --role-name eksctl-my-cluster-demo-nodegroup-m-NodeInstanceRole-pmyOEz976TZh \
  --policy-arn arn:aws:iam::533267266374:policy/CustomEKSClusterAutoscalerPolicy
```

Deploy the Cluster Autoscaler to your cluster with the following command :

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/autoscaler/master/cluster-autoscaler/cloudprovider/aws/examp
les/cluster-autoscaler-autodiscover.yaml
serviceaccount/cluster-autoscaler created
clusterrole.rbac.authorization.k8s.io/cluster-autoscaler created
role.rbac.authorization.k8s.io/cluster-autoscaler created
```

==========

```
clusterrolebinding.rbac.authorization.k8s.io/cluster-autoscaler created
rolebinding.rbac.authorization.k8s.io/cluster-autoscaler created
deployment.apps/cluster-autoscaler created
```

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

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl -n kube-system annotate deployment.apps/cluster-autoscaler
cluster-autoscaler.kubernetes.io/safe-to-evict="false"
deployment.apps/cluster-autoscaler annotated
```

Edit the Cluster Autoscaler deployment with the following command :

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

```
    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/<YOUR
        CLUSTER NAME>
      image: registry.k8s.io/autoscaling/cluster-autoscaler:v1.26.2
      imagePullPolicy: Always
      name: cluster-autoscaler
      resources:
```

Edit the command to replace *<YOUR CLUSTER NAME>* with your cluster's name, and add the following options.

--balance-similar-node-groups

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

==========

Update the autoscaler image tag to the latest Kubernetes version you are using.

I am currently using Kubernetes version 1.28, so search for the latest image tag available for that particular Kubernetes version.

From https://github.com/kubernetes/autoscaler/releases

I found that the latest tag available is `registry.k8s.io/autoscaling/cluster-autoscaler:v1.28.7`.

replace the current tag with this one.

```
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/my-cluster-demo
  - --balance-similar-node-groups
  - --skip-nodes-with-system-pods=false
  image: registry.k8s.io/autoscaling/cluster-autoscaler:v1.28.7
  imagePullPolicy: Always
  name: cluster-autoscaler
```

- Save and exit

Deploy nginx with over 30 replicas to see cluster scaling :

nginx-sample.yaml :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
```

==========

```
  labels:
    app: nginx
spec:
  replicas: 30
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
          requests:
            memory: "64Mi"
            cpu: "250m"
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /
```

```
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 10
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl apply -f nginx-sample.yaml
deployment.apps/nginx-deployment created
```

Currently we have 2 nodes :

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                            STATUS   ROLES    AGE    VERSION
ip-192-168-10-103.ap-south-1.compute.internal   Ready    <none>   147m   v1.28.15-eks-aeac579
ip-192-168-34-32.ap-south-1.compute.internal    Ready    <none>   148m   v1.28.15-eks-aeac579
```

Most of the pods will remain in pending state in a while as there is no room for them on 2 nodes. Then autoscaler will come in action and provision additional nodes , as we have defined in the autoscaling group.

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods | grep nginx
nginx-deployment-76df669566-2jzhl   0/1     Pending   0          51s
nginx-deployment-76df669566-4hg6x   1/1     Running   0          51s
nginx-deployment-76df669566-4ttfs   0/1     Pending   0          50s
nginx-deployment-76df669566-68bjk   1/1     Running   0          51s
nginx-deployment-76df669566-8lnqw   0/1     Pending   0          50s
nginx-deployment-76df669566-b7lxh   0/1     Pending   0          50s
xxxxxxxxxxxxxxxxxxxx more xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Additional Nodes are provisioned as per ASG defined.

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE    VERSION
ip-192-168-10-103.ap-south-1.compute.internal   Ready    <none>   148m   v1.28.15-eks-aeac579
ip-192-168-34-32.ap-south-1.compute.internal    Ready    <none>   148m   v1.28.15-eks-aeac579
ip-192-168-71-23.ap-south-1.compute.internal    Ready    <none>   47s    v1.28.15-eks-aeac579
ip-192-168-91-64.ap-south-1.compute.internal    Ready    <none>   45s    v1.28.15-eks-aeac579
```

After deployment delete , autoscaler will automatically scale down nodes :

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE    VERSION
ip-192-168-10-103.ap-south-1.compute.internal   Ready    <none>   160m   v1.28.15-eks-aeac579
ip-192-168-34-32.ap-south-1.compute.internal    Ready    <none>   160m   v1.28.15-eks-aeac579
ip-192-168-71-23.ap-south-1.compute.internal    Ready    <none>   12m    v1.28.15-eks-aeac579
ip-192-168-91-64.ap-south-1.compute.internal    Ready    <none>   12m    v1.28.15-eks-aeac579
```

It will take a while like 10min +. To scale down nodes to avoid any false positive,

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE    VERSION
ip-192-168-10-103.ap-south-1.compute.internal   Ready    <none>   161m   v1.28.15-eks-aeac579
ip-192-168-34-32.ap-south-1.compute.internal    Ready    <none>   161m   v1.28.15-eks-aeac579
ip-192-168-71-23.ap-south-1.compute.internal    Ready    <none>   13m    v1.28.15-eks-aeac579
```

8.0.2: VPA (Vertical Pod Autoscaler) :

1. **VPA is not recommended in production** because it restarts pods to apply new CPU and memory requests, leading to potential downtime. This behavior can lead to disruptions in production environments, especially for stateful applications.

2. **Pod restart in VPA means:**
    - The existing pod is **terminated** (deleted).
    - A **new pod** is created with updated resource requests/limits.
    - This is required because **Kubernetes does not allow changing CPU/memory requests on a running pod**.

3. **VPA and HPA should not be used together for CPU scaling**, as they can conflict.

    - HPA scales pod replicas based on CPU/memory usage.
    - VPA adjusts individual pod requests, which can cause instability when combined with HPA.

4. **Vertical vs. Horizontal Scaling:**
    - **Vertical Scaling**: Increasing CPU/memory of existing pods (**scaling up/down**).
    - **Horizontal Scaling**: Increasing/decreasing the number of pods (**scaling out/in**).
5. **Used in dev to determine optimal CPU and Memory for the Pod :**
    - VPA recommends pod request and limits
    - Use the numbers for request, limits, HPA
    - VPA should not be used with HPA

6. **Pod will go up and down in size (after restart) :**
    - The pod gets restarted with new resource requests and limits, effectively **changing its size** (CPU/memory).

A. Installation of VPA :

- Make sure you have already installed the Metric server [https://github.com/kubernetes-sigs/metrics-server].

Clone src code :

```
git clone https://github.com/kubernetes/autoscaler
cd vertical-pod-autoscaler
./hack/vpa-down.sh
```

B. Installation of Goldilocks

Ref. : https://goldilocks.docs.fairwinds.com/installation/#requirements

Goldilocks is a utility that can help you identify a starting point for resource requests and limits.

```
git clone https://github.com/FairwindsOps/goldilocks.git
cd goldilocks
kubectl create namespace goldilocks
kubectl -n goldilocks apply -f hack/manifests/controller
kubectl -n goldilocks apply -f hack/manifests/dashboard
```

Enable Namespace

Pick an application namespace and label it like so in order to see some data:

```
kubectl label ns goldilocks goldilocks.fairwinds.com/enabled=true
```

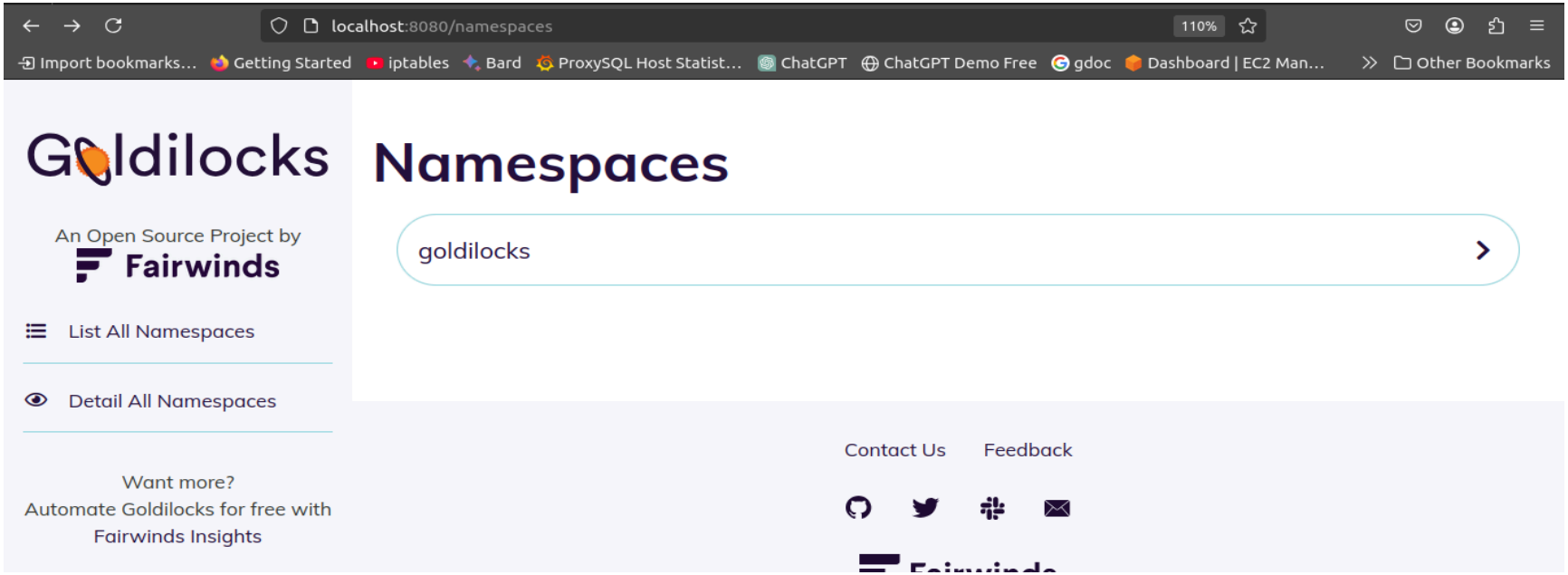
Viewing the Dashboard :

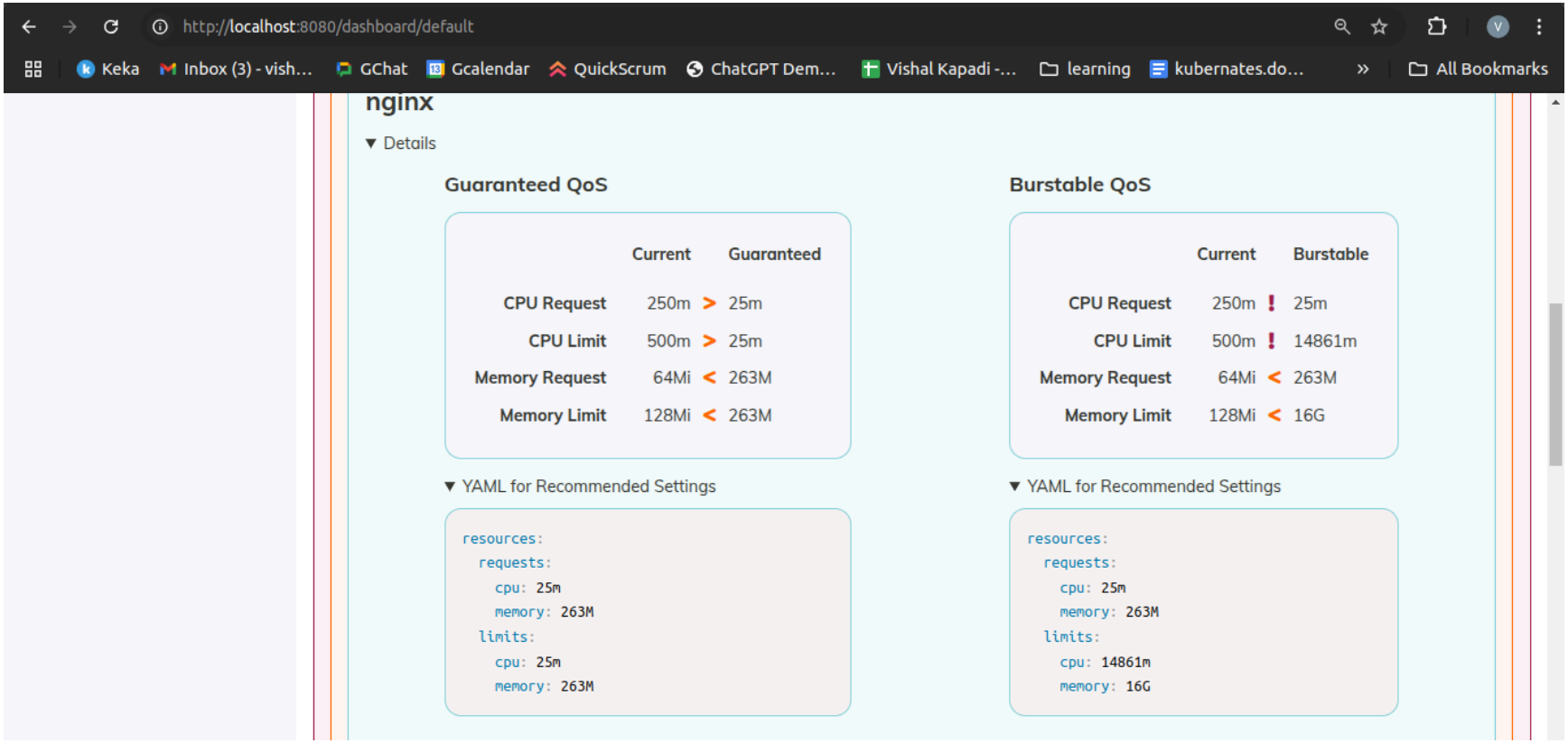
The default installation creates a ClusterIP service for the dashboard. You can access via port forward:

```
kubectl -n goldilocks port-forward svc/goldilocks-dashboard 8080:80
```

Then open your browser to http://localhost:8080/

```
vishal@vishalk17:~/Downloads/vish/goldilocks$ kubectl -n goldilocks apply -f hack/manifests/dashboard
clusterrole.rbac.authorization.k8s.io/goldilocks-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/goldilocks-dashboard created
deployment.apps/goldilocks-dashboard created
service/goldilocks-dashboard created
serviceaccount/goldilocks-dashboard created
vishal@vishalk17:~/Downloads/vish/goldilocks$ kubectl label ns goldilocks goldilocks.fairwinds.com/enabled=true
namespace/goldilocks labeled
vishal@vishalk17:~/Downloads/vish/goldilocks$ kubectl label ns default goldilocks.fairwinds.com/enabled=true
namespace/default labeled
vishal@vishalk17:~/Downloads/vish/goldilocks$ kubectl -n goldilocks port-forward svc/goldilocks-dashboard 8080:80
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```





Here, you will receive recommended settings, which you can later use in pod deployment to set limits and requests.

9.0.0 : EKS Auto Mode :

Amazon EKS **Auto Mode** is a fully managed deployment option for running Kubernetes clusters with minimal operational overhead. It simplifies the cluster setup by managing the control plane and worker nodes automatically

How Does EKS Auto Mode Select Instance Types?

1. **Workload Analysis:**
   - EKS Auto Mode evaluates your pod resource requests (CPU, memory, GPU).
   - It determines the best instance type to match the pod requirements.
2. **Instance Type Selection:**
   - Uses a mix of **Amazon EC2 Spot Instances** and **On-Demand Instances** to optimize cost and performance.
   - Selects instances from various families like **t3, m5, c5, r5, g4dn, p4d** based on pod requirements.
   - If you deploy GPU workloads, it picks **G4 or P4 instances** automatically.
3. **Automatic Scaling:**
   - EKS Auto Mode adjusts the number of instances dynamically based on pod scheduling demands.
   - Unused nodes are **terminated** automatically to reduce costs.

Key Features of EKS Auto Mode

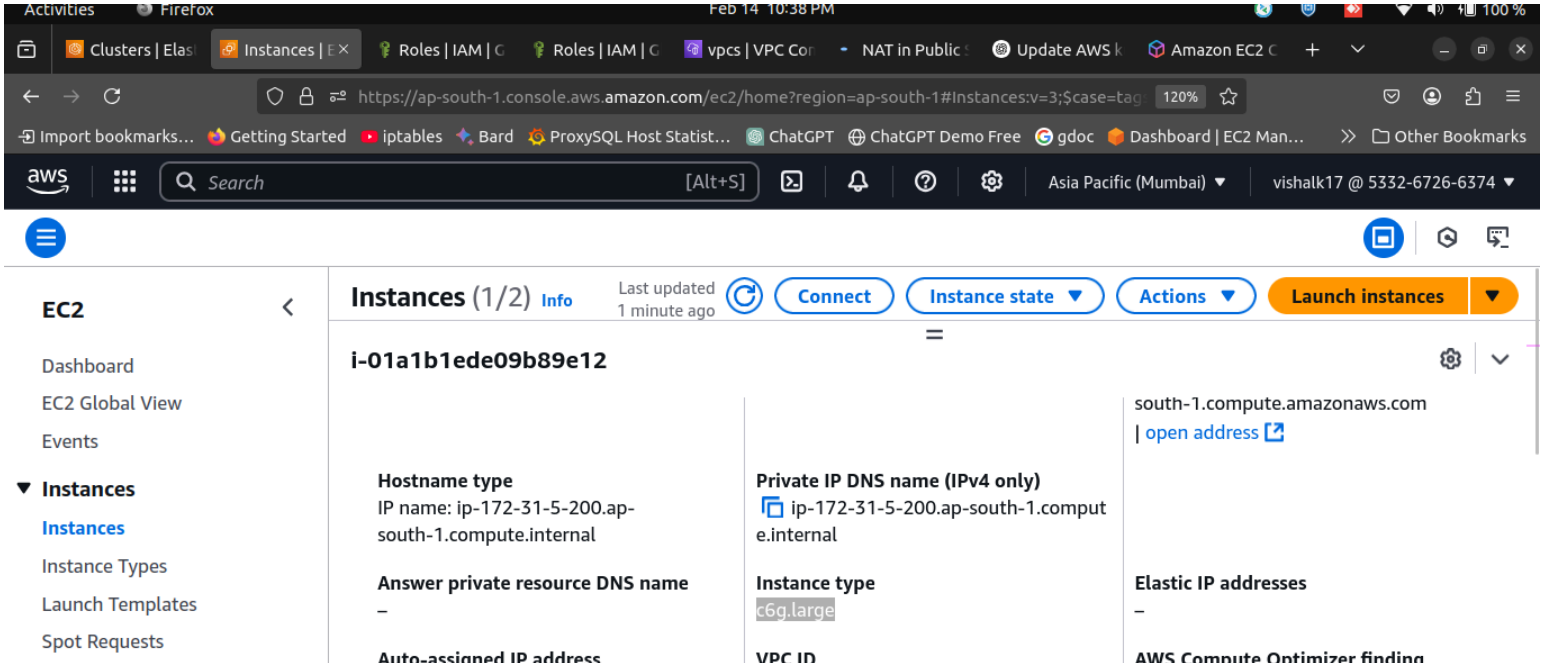
✅ **Fully Managed Control Plane & Nodes** – AWS automatically provisions and manages both the Kubernetes control plane and select compute capacity needs by application..
✅ **Simplified Node Management** – No need to manually configure or maintain worker nodes.
✅ **Optimized for Cost & Performance** – Automatically adjusts node types based on workload demands.
✅ **Integrated AWS Networking & Security** – Comes preconfigured with VPC, IAM roles, and security best practices.
✅ **Automatic Upgrades & Patching** – AWS handles Kubernetes version upgrades and security patches.
✅ **Effortless Scaling** – Dynamically scales worker nodes as needed, similar to Fargate but with more flexibility.

I have created an EKS cluster in auto mode using the console. You can do that too.

Now, let's check how many nodes we have once the EKS cluster is launched successfully,

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                  STATUS   ROLES    AGE   VERSION
i-012609e6a0e276ebb   Ready    <none>   21m   v1.31.4-eks-0f56d01
i-01a1b1ede09b89e12   Ready    <none>   21m   v1.31.4-eks-0f56d01
```

Both instance specs : c6g.large 2vpu and 4gb ram



```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods -o wide -A
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE   IP              NODE
NOMINATED NODE   READINESS GATES
kube-system   metrics-server-675475fb68-7fcnh   1/1     Running   0          25m   172.31.2.48     i-01a1b1ede09b89e12
kube-system   metrics-server-675475fb68-lrlj8   1/1     Running   0          25m   172.31.25.208   i-012609e6a0e276ebb
```

Nginx-eks-automode-test.yaml :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: "2Gi"
            cpu: "3000m"
```

It should automatically launch a new instance, as **c6g.large** supports only 2 vCPUs.

Let's check,

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl apply -f nginx-eks-automode-test.yaml
deployment.apps/nginx-deployment created

vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                  STATUS   ROLES    AGE   VERSION
i-012609e6a0e276ebb   Ready    <none>   29m   v1.31.4-eks-0f56d01
i-01a1b1ede09b89e12   Ready    <none>   29m   v1.31.4-eks-0f56d01

vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
nginx-deployment-6b7b6f4868-9dglk   0/1     ContainerCreating   0          24s

vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                  STATUS   ROLES    AGE   VERSION
i-012609e6a0e276ebb   Ready    <none>   30m   v1.31.4-eks-0f56d01
i-01a1b1ede09b89e12   Ready    <none>   30m   v1.31.4-eks-0f56d01
i-0aa25e931decd8262   Ready    <none>   33s   v1.31.4-eks-0f56d01
```

`i-0aa25e931decd8262` instance launched with specs c6.xlarge (4vcpu , 8gb Ram)

10.0 : Logging & Monitoring EKS :

10.1 : Two Different Kinds of Logging for EKS

1. EKS Control Plane Logging

Logs related to the Kubernetes control plane components:

- **Kubernetes API Server Logs** – Requests and responses to the Kubernetes API server.
- **Audit Logs** – Records of requests made to the API server, useful for security and compliance.
- **Authenticator Logs** – Logs related to authentication requests to the API server.
- **Controller Manager Logs** – Logs from the Kubernetes controller manager, which regulates cluster state.
- **Scheduler Logs** – Logs from the Kubernetes scheduler, responsible for scheduling pods onto nodes.

2. EKS Worker Node Logging

Logs collected from worker nodes running workloads:

- **System Logs** – Logs from essential node components:
  - **Kubelet Logs** – Logs from the Kubernetes node agent.
  - **Kube-proxy Logs** – Logs from the Kubernetes network proxy.
  - **Container Runtime Logs** – Logs from the container runtime, such as **dockerd** or **containerd**.
- **Application Logs** – Logs generated by applications running inside containers.

10.2 : In order to extract logs from nodes, First we need to know where they are stored :

1. Log Locations

- **Containerized Application Logs** → Written to **stdout/stderr** (retrievable via kubectl log).
- **System Logs** → Stored and managed by **systemd** (accessible via journalctl).
- **Container Logs** → Redirected to **/var/log/containers/*.log**, which symlink to /var/log/pods/ and are managed by the container runtime (Docker, containerd, etc.).

2.Various Observability Tools Available for Log Collection & Monitoring

- **Prometheus / Grafana / Loki** → Open-source stack for collecting metrics, logs, and visualizing them.
- **CloudWatch Container Insights** → AWS-native monitoring and logging solution for containerized workloads.
- **Fluentd / Kibana / ElasticSearch**→ A log forwarder that collects, filters, and routes logs to various destinations like CloudWatch, Elasticsearch, or Loki.
- **And more.**

11.0 : Ingress :

Ingress is an API object that manages external access to services within a cluster, typically via HTTP and HTTPS. It allows you to define rules for routing external traffic to internal services.

11.1 : Why Use Ingress?

- Provides **centralized routing** for services.
- Supports **path-based and host-based routing**.
- Can **terminate SSL/TLS** connections.
- Eliminates the need for multiple LoadBalancers or NodePorts.

11.2 : Types of Routing

1. **Host-Based Routing**
   - Routes requests based on domain name (`example.com` → `app1-service`).
2. **Path-Based Routing**
   - Routes requests based on the URL path (`example.com/app1` → `app1-service`).

11.3 : Various Types of Ingress Controller available :

- NGINX
- AWS ALB (For AWS env.)
- Traefik
- Istio
- & more

11.4 : Key Components :

1️⃣ Ingress Controller

- Monitors **Ingress Resources** and applies the defined rules.

==========

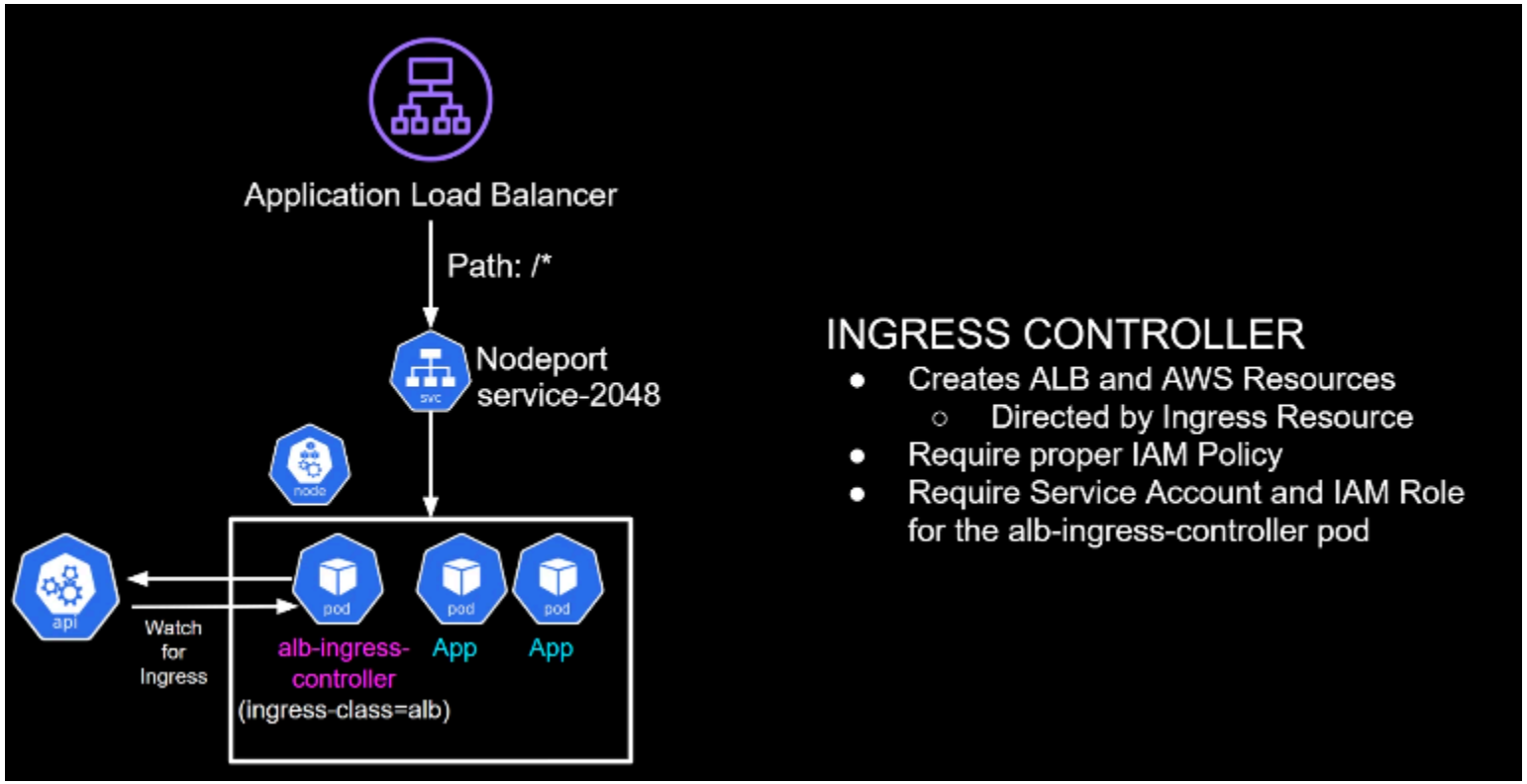
- Creates necessary cloud resources (e.g., **ALB** for AWS ALB Ingress Controller).
- One cluster can have **multiple Ingress Controllers** (e.g., NGINX and Traefik together).
- Each Ingress Resource can specify **which Ingress Controller to use** via `ingressClassName`.

2️⃣ Ingress Resource

- Defines **routing rules** for HTTP/S traffic.
- Specifies **which Ingress Controller** should handle the requests.
- Maps **URL paths** and **hostnames** to corresponding **backend services**.
- Can include additional features like **TLS termination, rewrite rules, and authentication**.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-xyz-ingress
  annotations:
    kubernetes.io/ingress.class: alb  # Tells Kubernetes to use AWS ALB Ingress Controller
    alb.ingress.kubernetes.io/scheme: internet-facing  # Creates a public ALB
spec:
  rules:
  - host: myapp.example.com  # Domain name
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app-service  # Maps to Kubernetes Service
            port:
              number: 80  # Target port
```

11.5: AWS Ingress :



The AWS ALB Ingress Controller supports two traffic modes:

The AWS ALB Ingress Controller (now part of the AWS Load Balancer Controller) supports two primary traffic modes: **Instance Mode** and **IP Mode**.

1. Instance Mode (Default Mode)

- **Description:**
  In **Instance Mode**, the ALB registers the **worker nodes** (EC2 instances) in the Kubernetes cluster as targets.
- **Traffic Flow:**
  - Traffic from the ALB is routed to the **NodePort** of the service on the worker nodes.
  - The traffic is then **proxied** to the pods within the Kubernetes cluster by the nodes.
- **Requirements:**
  - Your Kubernetes service must be of type **NodePort** or **LoadBalancer**.
  - The service must expose a NodePort that allows ALB to forward traffic to the worker nodes.

2. IP Mode

- **Description:**
  In **IP Mode**, the ALB registers the **individual pods** of the Kubernetes service as targets. Traffic is routed directly from the ALB to the pods.
- **Traffic Flow:**
  - ALB routes traffic **directly to the pods**, bypassing the worker nodes.
- **Requirements:**
  - You must use a **CNI plugin** that supports assigning **secondary IP addresses to pods**. The **AWS VPC CNI plugin** is commonly used in this mode.
- **Use Cases:**
  - This mode can **reduce latency** by routing traffic directly to the pods and is useful when you need **fine-grained control** over pod networking.

Traffic Mode Configuration

You can specify the traffic mode for your ALB Ingress by setting the annotation `alb.ingress.kubernetes.io/target-type` in your **Ingress** or **Service** definition:

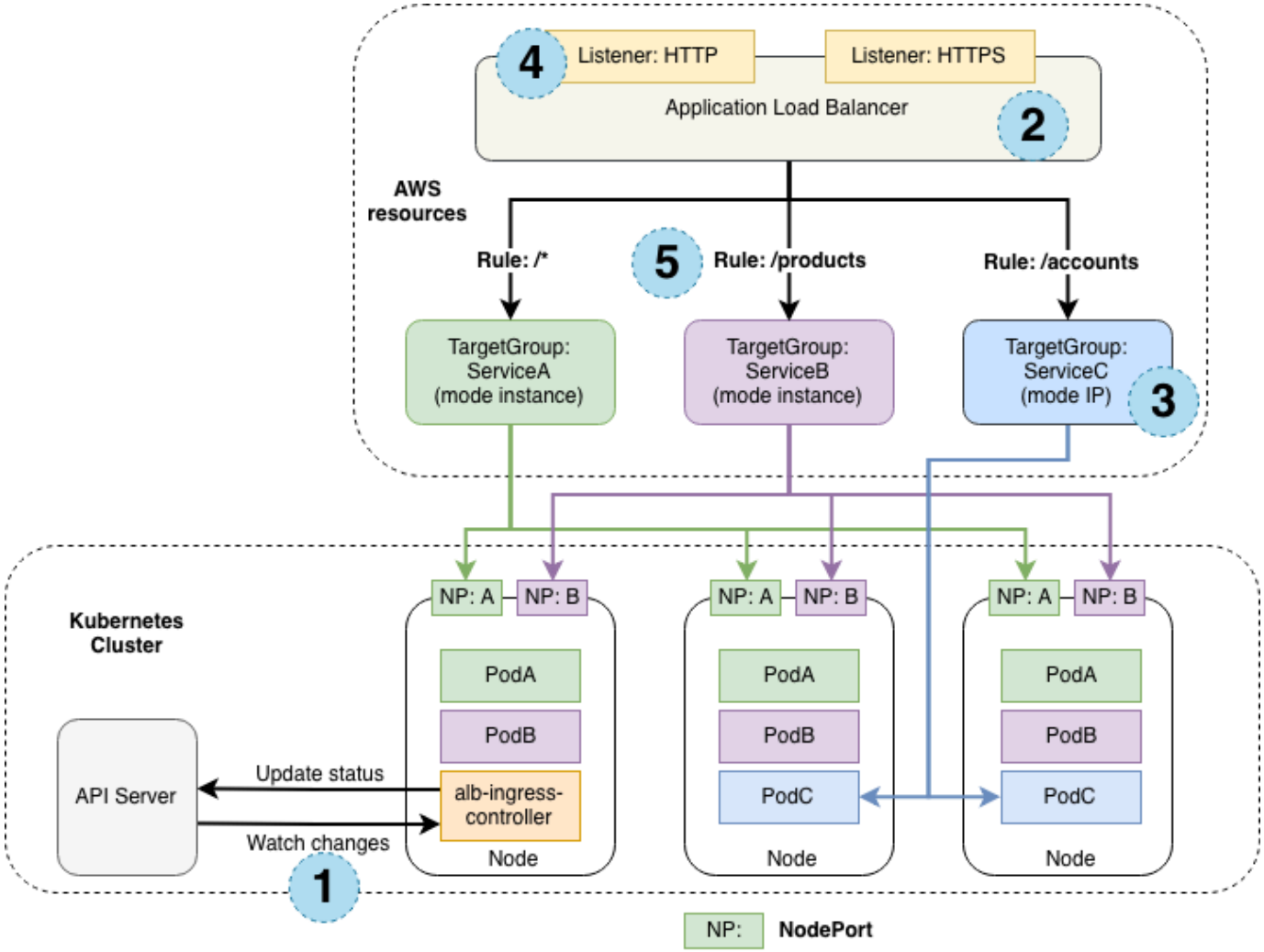
Instance Mode (default):

```
metadata:
  annotations:
    alb.ingress.kubernetes.io/target-type: instance
```

==========

IP Mode:

```
metadata:
  annotations:
    alb.ingress.kubernetes.io/target-type: ip
```



12.0 : EKS Advanced Concepts :

Setup EKS Environment and Setup AWS Load Balancer Ingress Controller :

A. Create EKS Cluster :

```
vishal@vishalk17:~/Downloads/vish/eks$ ls
config-file.yaml  hpa-demo.yaml  nginx-eks-automode-test.yaml  nginx-sample.yaml

vishal@vishalk17:~/Downloads/vish/eks$ eksctl create cluster -f config-file.yaml
```

B. Now create an IAM OIDC provider and associate it with your cluster:

An OIDC provider is necessary to deploy in Amazon EKS because it enables secure authentication between your Kubernetes cluster and AWS services. It allows pods to assume IAM roles, granting them temporary credentials to access AWS resources without hardcoding secrets.

```
eksctl utils associate-iam-oidc-provider --cluster=<cluser-name> --approve
```

```
vishal@vishalk17:~/Downloads/vish/eks$ cat config-file.yaml | grep name
  name: my-cluster-demo         # Name of the EKS cluster
  - name: my-cluster-demo-ng    # Node group name

vishal@vishalk17:~/Downloads/vish/eks$ eksctl utils associate-iam-oidc-provider --cluster=my-cluster-demo
--approve
2025-02-27 13:56:53 [i]  will create IAM Open ID Connect provider for cluster "my-cluster-demo" in
"ap-south-1"
2025-02-27 13:56:54 [✔]  created IAM Open ID Connect provider for cluster "my-cluster-demo" in
"ap-south-1"
```

C. Deploy AWS ALB Ingress controller Using helm :

Ref.: https://docs.aws.amazon.com/eks/latest/userguide/lbc-helm.html

Download an IAM policy for the AWS Load Balancer Controller that allows it to make calls to AWS APIs on your behalf.

```
curl -O
https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.11.0/docs/install/iam_policy.json
```

Create an IAM policy using the policy downloaded in the previous step.

```
aws iam create-policy \
    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://iam_policy.json
```

Note Down **ARN**.

Replace *ARN* and then run the command.

```
eksctl create iamserviceaccount \
  --cluster=<your-cluster-name> \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --role-name AmazonEKSLoadBalancerControllerRole \
  --attach-policy-arn=<ARN-FROM-PREVIOUS-STEP> \
  --approve
```

Add the eks-charts Helm chart repository. AWS maintains this repository on GitHub.

```
helm repo add eks https://aws.github.io/eks-charts
```

==========

Update your local repo to make sure that you have the most recent charts.

```
helm repo update eks
```

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=my-cluster \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

The `helm install` command automatically installs the custom resource definitions (CRDs) for the controller. The `helm upgrade` command does not. If you use `helm upgrade`, you must manually install the CRDs. Run the following command to install the CRDs:

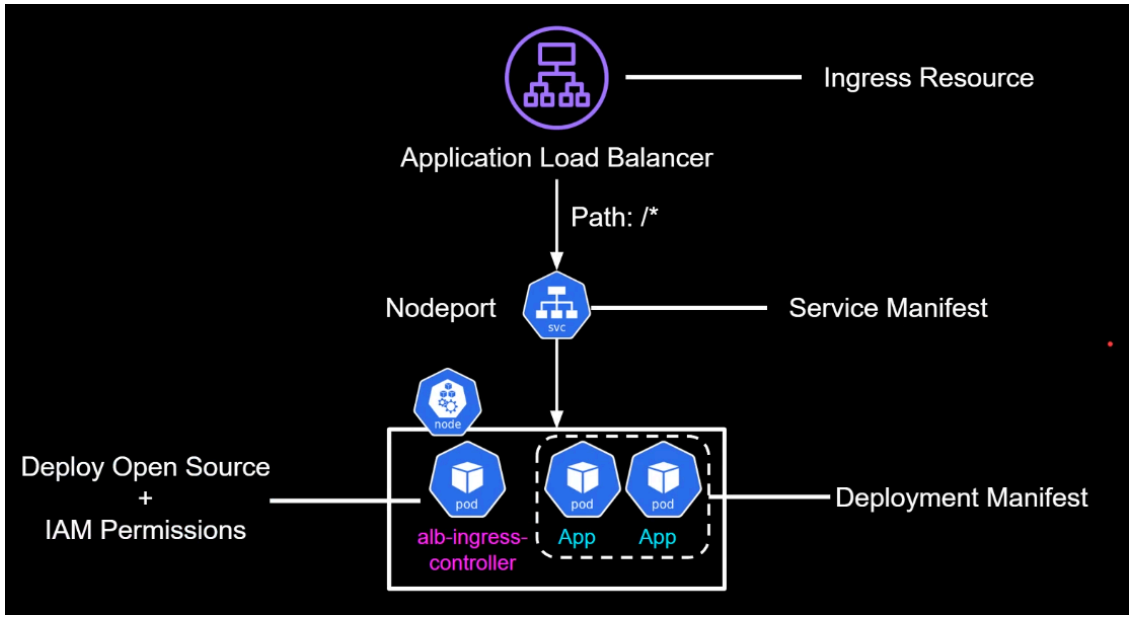
```
wget https://raw.githubusercontent.com/aws/eks-charts/master/stable/aws-load-balancer-controller/crds/crds.yaml
kubectl apply -f crds.yaml
```

Verify that the controller is installed.

```
kubectl get deployment -n kube-system aws-load-balancer-controller
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get deployment -n kube-system aws-load-balancer-controller
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
aws-load-balancer-controller   2/2     2            2           33s
```

12.1 : Demo 1 : AWS ingress-path :



Deploy Sample Application Game :

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.1.4/docs/examples/2048/2048-namespace.yaml

kubectl apply -f
https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.1.4/docs/examples/2048/2048-deployment.yaml

kubectl apply -f
https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.1.4/docs/examples/2048/2048-service.yaml
```

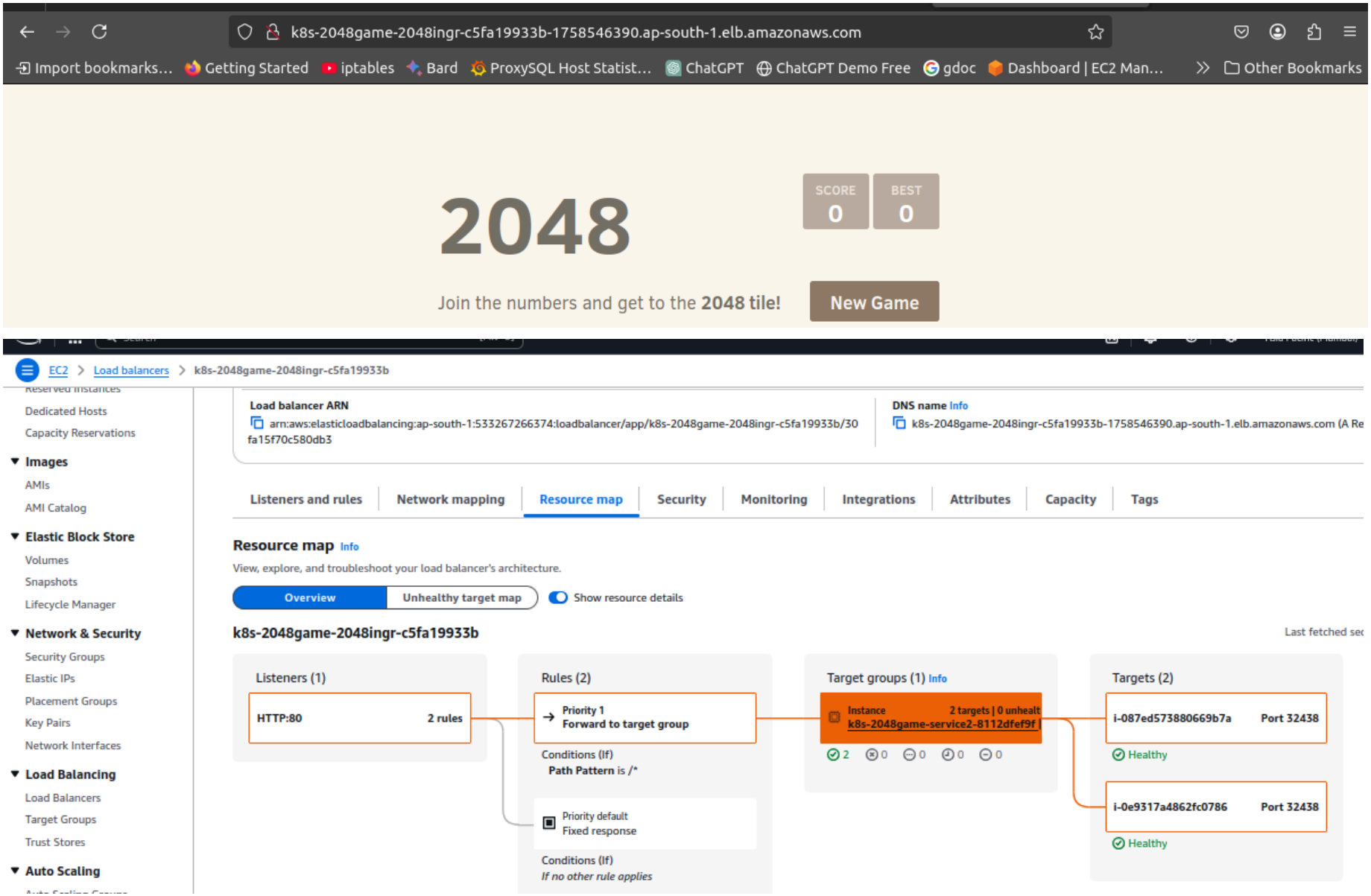
==========

Deploy ingress :

ingress-path.yaml

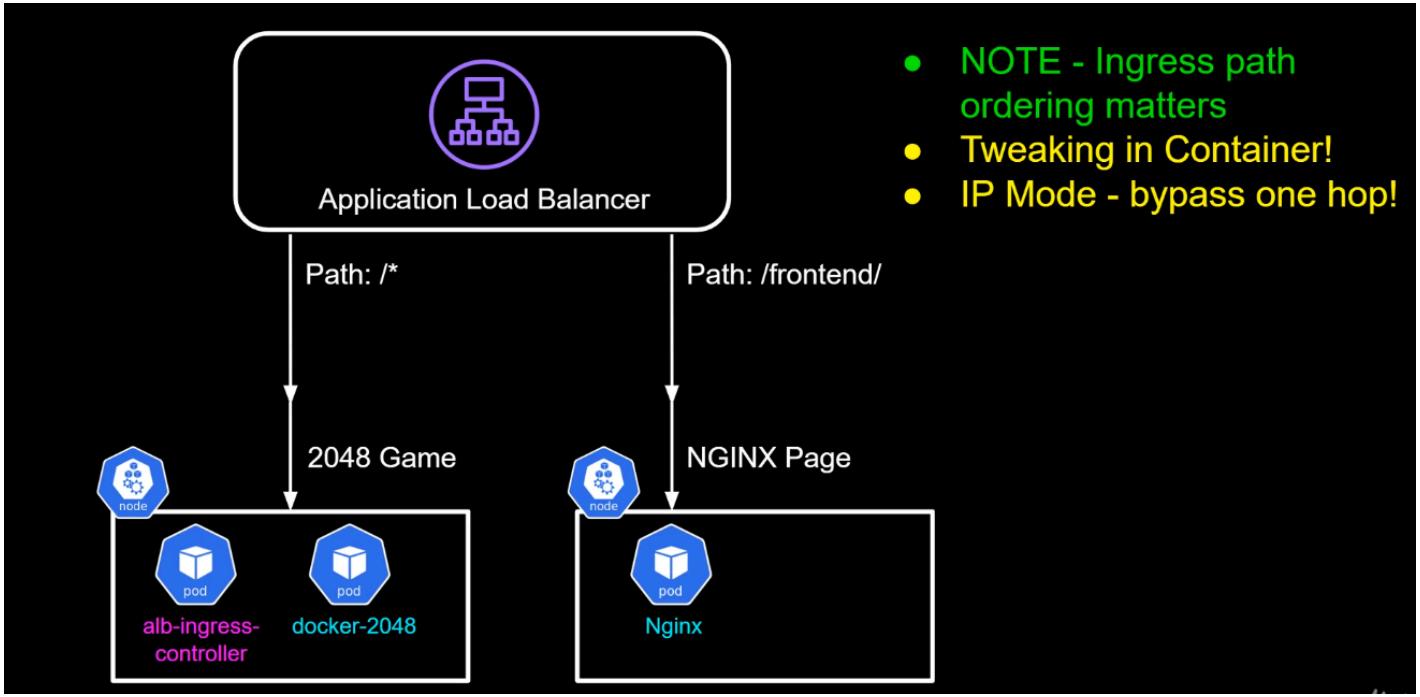
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: 2048-ingress
  namespace: 2048-game
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
  labels:
    app: 2048-ingress
spec:
  ingressClassName: alb    # Added this instead of the deprecated annotation
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service-2048
                port:
                  number: 80
```

```
vishal@vishalk17:~/Downloads/vish/eks/2048-game$ kubectl get ingress -n 2048-game
NAME           CLASS   HOSTS   ADDRESS                                                   PORTS   AGE
2048-ingress   alb     *       k8s-2048game-2048ingr-c5fa19933b-1758546390.ap-south-1.elb.amazonaws.com   80      19s
```


k8s-2048game-2048ingr-c5fa19933b-1758546390.ap-south-1.elb.amazonaws.com
Import bookmarks... Getting Started iptables Bard ProxySQL Host Statist... ChatGPT ChatGPT Demo Free gdoc Dashboard | EC2 Man... Other Bookmarks
2048
SCORE 0
BEST 0
Join the numbers and get to the 2048 tile!
New Game
EC2 > Load balancers > k8s-2048game-2048ingr-c5fa19933b
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Load balancer ARN
arn:aws:elasticloadbalancing:ap-south-1:533267266374:loadbalancer/app/k8s-2048game-2048ingr-c5fa19933b/30fa15f70c580db3
DNS name Info
k8s-2048game-2048ingr-c5fa19933b-1758546390.ap-south-1.elb.amazonaws.com (A Re
Listeners and rules
Network mapping
Resource map
Security
Monitoring
Integrations
Attributes
Capacity
Tags
Resource map Info
View, explore, and troubleshoot your load balancer's architecture.
Overview
Unhealthy target map
Show resource details
k8s-2048game-2048ingr-c5fa19933b
Last fetched sec
Listeners (1)
HTTP:80 2 rules
Rules (2)
Priority 1
Forward to target group
Conditions (If)
Path Pattern is /*
Priority default
Fixed response
Conditions (If)
If no other rule applies
Target groups (1) Info
Instance 2 targets | 0 unhealt
k8s-2048game-service2-8112dfef9f
2 0 0 0 0
Targets (2)
i-087ed573880669b7a Port 32438
Healthy
i-0e9317a4862fc0786 Port 32438
Healthy

12.2 : Demo 2 : path with IP model :

- In **IP Mode**, the ALB registers the **individual pods** of the Kubernetes service as targets. Traffic is routed directly from the ALB to the pods.

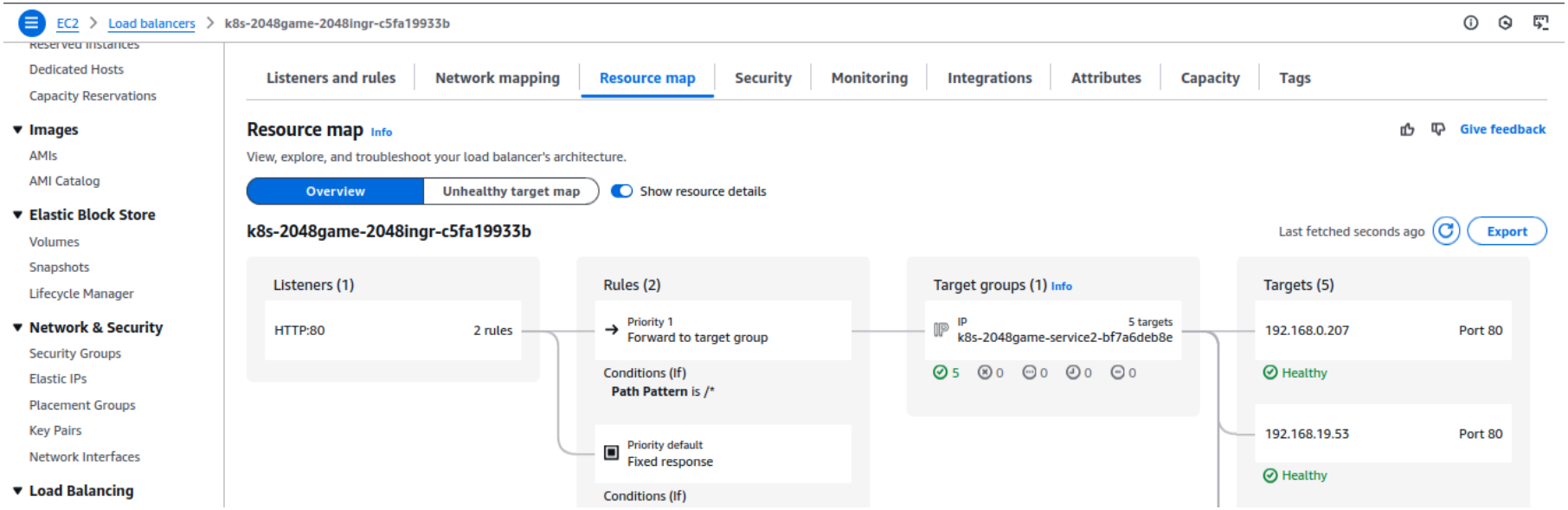


Ingress-ip.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: 2048-ingress
  namespace: 2048-game
```

==========

```
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip  # For IP-based targeting
  labels:
    app: 2048-ingress
spec:
  ingressClassName: alb    # Added this instead of the deprecated annotation
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service-2048
                port:
                  number: 80
```



```
vishal@vishalk17:~/Downloads/vish/eks/2048-game$ kubectl apply -f ingress-ip.yaml
ingress.networking.k8s.io/2048-ingress created
vishal@vishalk17:~/Downloads/vish/eks/2048-game$ kubectl get pods -n 2048-game
NAME                                READY   STATUS    RESTARTS   AGE
2048-deployment-9ccbf594b-499db   1/1     Running   0          35m
2048-deployment-9ccbf594b-7k9ws   1/1     Running   0          35m
2048-deployment-9ccbf594b-j5stk   1/1     Running   0          35m
2048-deployment-9ccbf594b-n45m9   1/1     Running   0          35m
2048-deployment-9ccbf594b-s5lfj   1/1     Running   0          35m
vishal@vishalk17:~/Downloads/vish/eks/2048-game$ kubectl get pods -o wide -n 2048-game
NAME                                READY   STATUS    RESTARTS   AGE   IP               NODE
ODE   READINESS GATES
2048-deployment-9ccbf594b-499db   1/1     Running   0          35m   192.168.63.168   ip-192-168-42
      <none>
2048-deployment-9ccbf594b-7k9ws   1/1     Running   0          35m   192.168.19.53    ip-192-168-21
      <none>
2048-deployment-9ccbf594b-j5stk   1/1     Running   0          35m   192.168.36.154   ip-192-168-42
      <none>
2048-deployment-9ccbf594b-n45m9   1/1     Running   0          35m   192.168.5.88     ip-192-168-21
      <none>
2048-deployment-9ccbf594b-s5lfj   1/1     Running   0          35m   192.168.0.207    ip-192-168-21
      <none>
vishal@vishalk17:~/Downloads/vish/eks/2048-game$
```

- Pod ips registered as a target

12.3.0 : Service Mesh :

A **service mesh** is a dedicated infrastructure layer that manages **communication between services** in a microservices architecture. It's like a traffic maintainer for your app's services, handling all the networking stuff—routing, security, retries, monitoring—without you having to code it into each service yourself

Core Idea: It uses **proxies** to intercept and manage traffic between services, controlled by a **central system** that sets the rules.

12.3.1: Key Components of a Service Mesh

1. **Data Plane**:
   - Made up of **proxies** (usually Envoy) that sit next to each service.
   - In Kubernetes, these are typically **sidecar proxies**—extra containers injected into your pods.
   - They handle the actual traffic: routing, encryption, retries, collecting metrics, etc.
2. **Control Plane**:
   - The brains of the operation. A central component (e.g., Istio's istiod) that configures the proxies.
   - You tell the control plane what you want (e.g., "encrypt traffic between App1 and App2"), and it pushes those rules to the proxies.

12.3.2: What Does a Service Mesh Do?

Here's the practical stuff it handles:

1. **Traffic Management**:
   - **Routing**: Directs requests to the right service (e.g., /api to App1, /web to App2).
   - **Load Balancing**: Spreads traffic across multiple instances of a service (e.g., App1's pods).
   - **Advanced Features**: Traffic splitting (e.g., 90% to v1, 10% to v2 for canary), retries, timeouts, circuit breaking.
2. **Security**:
   - **mTLS (Mutual TLS)**: Automatically encrypts traffic between services and verifies their identities.
   - No need to code TLS into your app—the mesh does it for you.
3. **Observability**:
   - Collects **metrics** (e.g., request latency, error rates), **logs**, and **traces** for every request.
   - Feeds this into tools like Prometheus or Jaeger so you can see what's happening.

4. **Resilience**:
    - Handles failures gracefully with retries (e.g., try again if a request fails) or circuit breakers (stop sending traffic to a failing service).

12.3.3: Why Use a Service Mesh?

- **Scalability**: When you have 10s or 100s of services, managing communication manually is a nightmare. A service mesh automates it.
- **Consistency**: Applies the same security, monitoring, and routing rules across all services.
- **No Code Changes**: Add features like encryption or retries without touching your app's code.

12.3.4: Popular Service Meshes

- **Istio**: Most popular, uses Envoy, integrates with Kubernetes via sidecars and an Ingress Gateway.
- **Linkerd**: Lightweight, simpler than Istio, also uses sidecars.
- **AWS App Mesh**: AWS-managed, uses Envoy, integrates with ECS/EKS (but being discontinued in 2026).
- **Consul**: Service mesh + service discovery, works across clouds.

12.3.5: How It Works in Kubernetes (e.g., Istio)

In a Kubernetes cluster, a service mesh like Istio looks like this:

- **Sidecar Proxies**:
    - Each pod gets an Envoy proxy injected as a sidecar (if Istio is enabled for that pod/namespace).
    - Example: App1's pod has its app container + an Envoy sidecar. All traffic to/from App1 goes through the sidecar.

Traffic Flow:

```
[App1 Pod] --> [Sidecar Proxy A] ------------> [Sidecar Proxy B] --> [App2 Pod]
   |               |                               |                   |
Simple HTTP     Adds mTLS, retries           Decrypts mTLS         Simple HTTP
                   Routes to App2            Applies policies
```

==========

Control Plane:

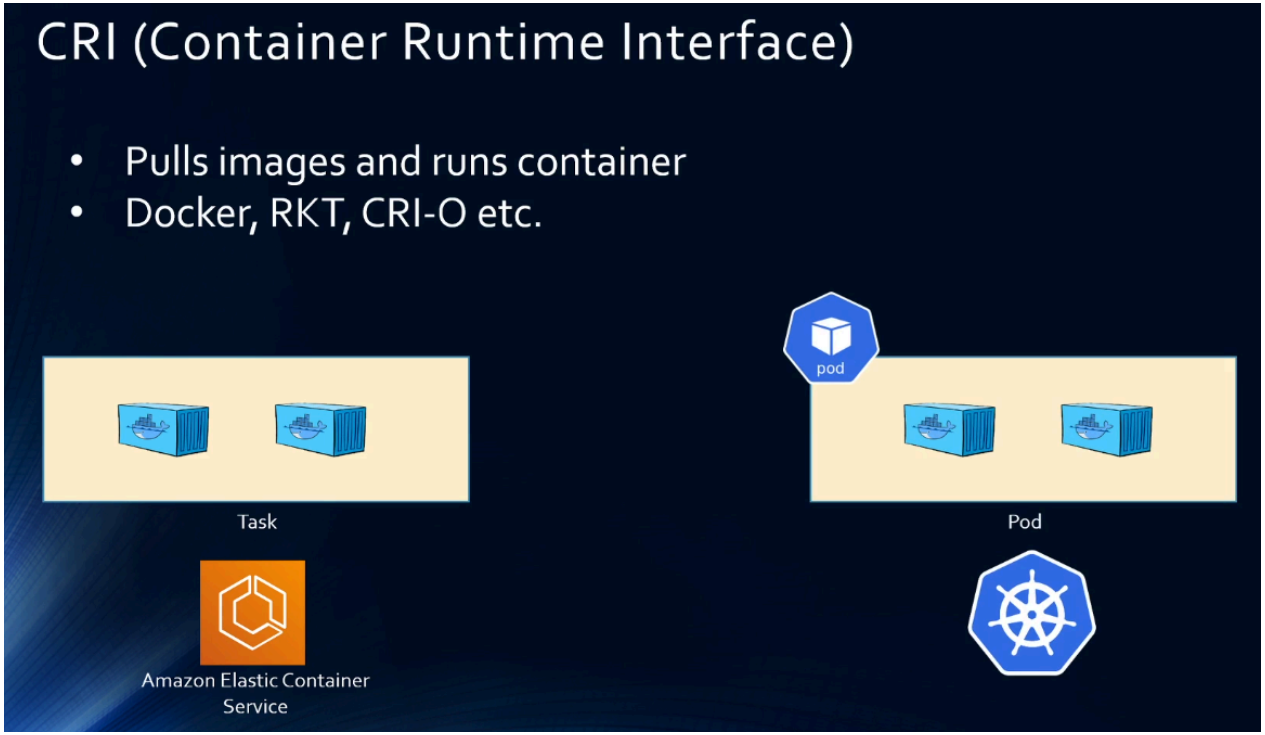
- Istio's istiod configures the sidecars with rules (e.g., "encrypt traffic to App2 with mTLS").

Ingress:

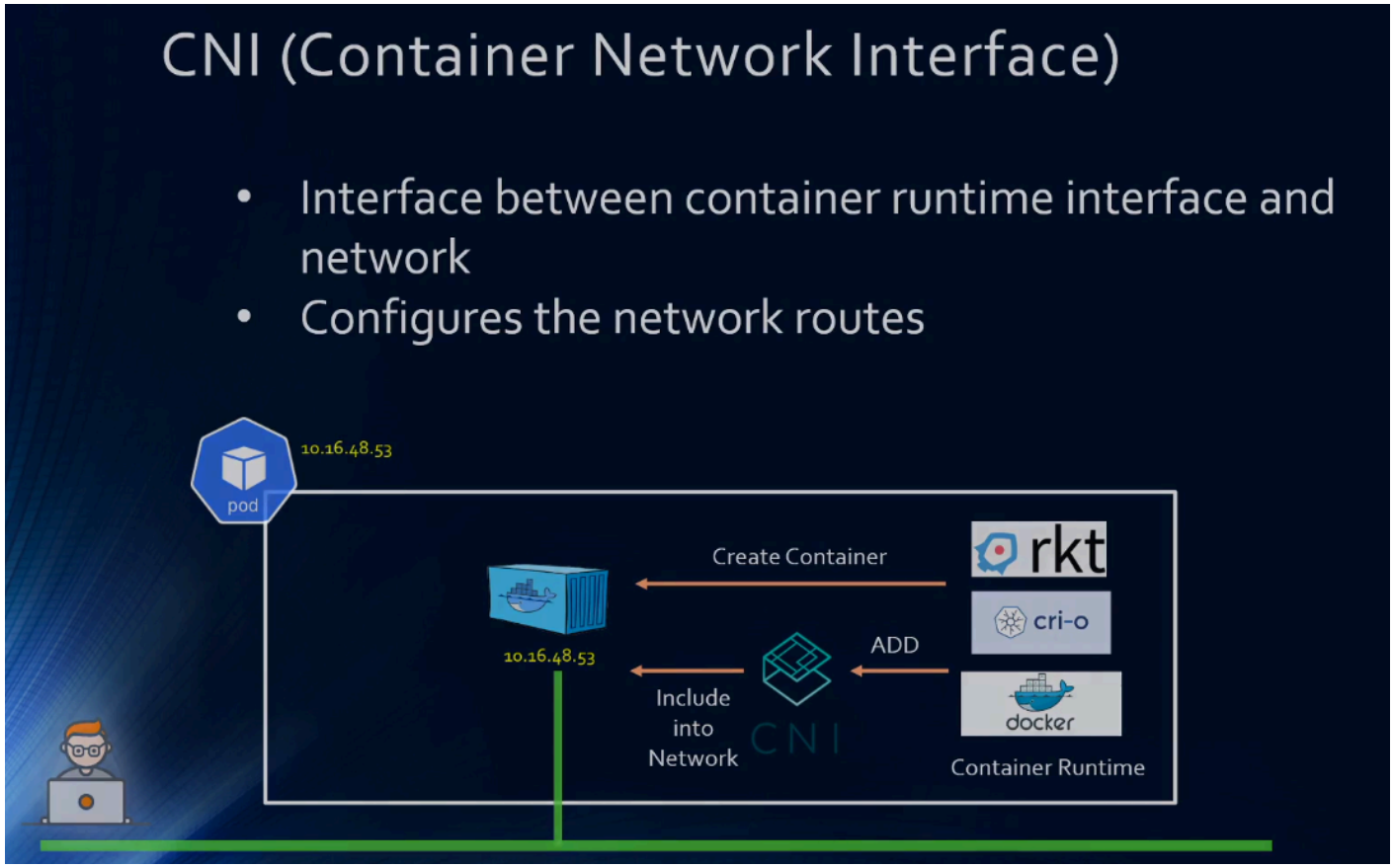
- External traffic comes in via the **Istio Ingress Gateway** (or something like AWS ALB), then hits the sidecars.

12.4.0 : CNI / CRI and Kubernetes Networking :

12.4.1 : CRI (Container Runtime Interface) :



12.4.2 : CNI (Container Network Interface) :



When a pod is created in Kubernetes:

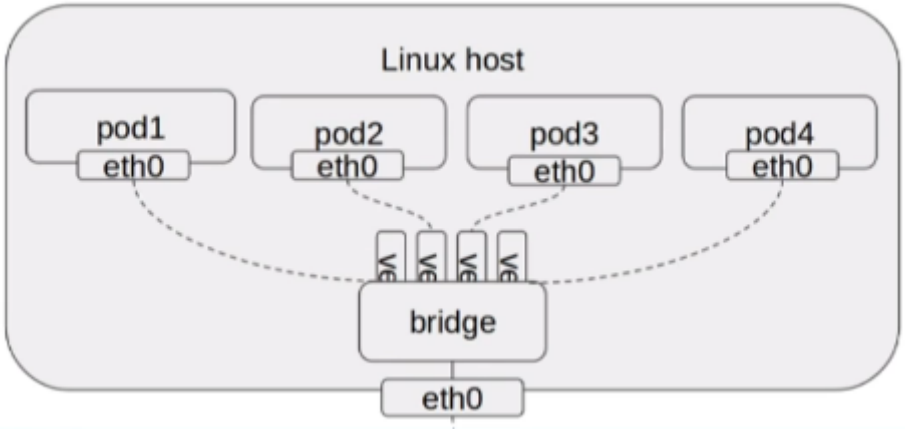
1. **CNI Assigns an IP Address**
   - Each pod gets a unique IP from the configured IP pool.
   - This IP is used for pod-to-pod communication.
2. **CNI Configures Network Rules**
   - It sets up routing, bridges, Network Address Translation (NAT), or overlays.

==========

- These rules enable communication between pods across nodes.

3. **CNI Connects the Pod to the Network**
   - The pod's network namespace is attached to a virtual switch, bridge, or overlay.
   - This ensures that the pod can communicate with other pods, services, and external networks.

Kubernetes Network :



A. Where is CNI Used?

CNI is **not limited to Kubernetes**. It is also used in **other container runtimes** and platforms:

Use Case	Example Platforms
Container Orchestration	Kubernetes, OpenShift, Nomad
Standalone Container Runtimes	Docker, CRI-O, containerd, Podman
Serverless Platforms	Knative, AWS Fargate
Virtual Machines	Firecracker, Kata Containers

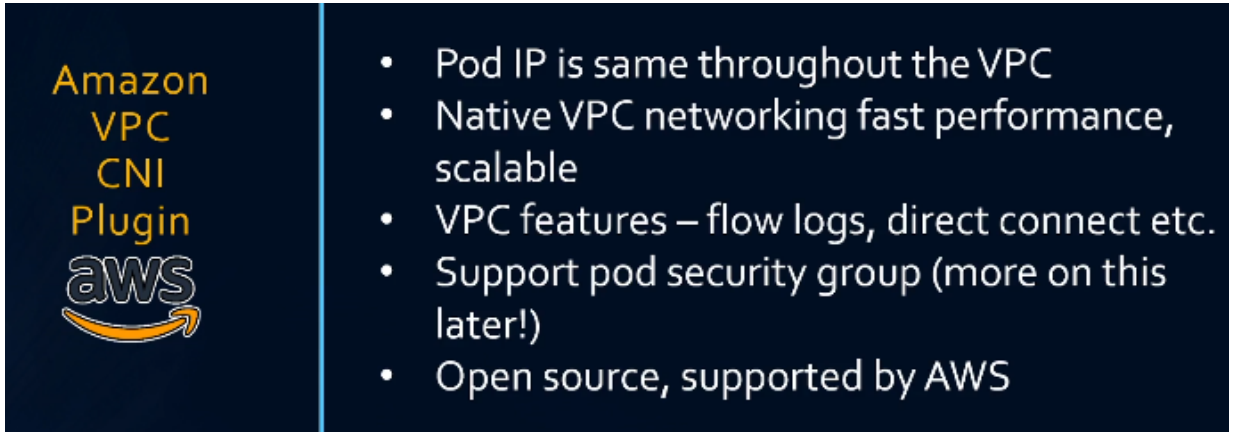
B. Kubernetes Networking Requirements :

- Each pod gets its own IP
  - Containers within a pod share network namespaces
- All pods can communicate with all other pods without NAT (Network Address Translation)
- All nodes can communicate with all pods without NAT
- The IP of the pod is same throughout the cluster

C. Various CNI Plugins Available for k8s :

- Calico
- Flannel
- Amazon VPC CNI
- & More.

Amazon VPC CNI Plugin (Available in AWS EKS) :



D. Amazon VPC CNI Pod Networking: IP Allocation and Scalability Across Subnets in EKS:

In this setup, an **AWS Virtual Private Cloud (VPC)** contains two subnets:

- **Subnet-1:** Hosts an **EC2 instance (EKS node)** and has **only 3 free IP addresses** available.
- **Subnet-2:** Has **100 free IP addresses**, providing additional capacity for pod IP allocation.

==========

IP Allocation for Pods in Subnet-1

The **Amazon VPC CNI plugin** assigns **IP addresses to pods** directly from the **available IP pool of Subnet-1**.

- With **only 3 free IPs** in Subnet-1, the **EC2 instance can initially support 3 pods**.
- Each pod receives a **unique IP address from the VPC subnet**, ensuring **native networking without NAT**.

Scaling Pods Using Subnet-2 IPs

When **Subnet-1 runs out of available IPs**, the **VPC CNI plugin** can allocate additional IPs from **Subnet-2**.

- This is achieved using **Elastic Network Interfaces (ENIs)** attached to the EC2 instance.
- These ENIs allow the instance to pull **secondary IP addresses** from **Subnet-2**, enabling seamless pod scaling **without requiring manual reconfiguration**.

Expanding IP Capacity with New CIDR Blocks

If both **Subnet-1 and Subnet-2 (or the entire VPC) exhaust their IP addresses**, the EKS cluster's capacity can be expanded by:

1. **Adding a new CIDR block** to the VPC.
2. **Creating additional subnets** with new IP ranges.
3. Allowing the **VPC CNI plugin** to utilize these new subnets for IP allocation.

This ensures the cluster can **scale efficiently** without running into IP shortages.

12.5.0 : Kubernetes Network Policy & Security Group for Pods :

12.5.1: Kubernetes Network Policy :

What is Kubernetes Network Policy?

Kubernetes Network Policy is a resource that allows you to control network traffic to and from pods in a Kubernetes cluster. By default, Kubernetes allows all pods to communicate with each other freely (unless restricted by the underlying network layer). Network Policies provide a way to define rules for ingress (incoming) and egress (outgoing) traffic at the pod level, giving you fine-grained control over pod-to-pod communication.

Think of it like a firewall for your pods: you specify *who* can talk to *whom* and *how*. Network Policies are especially useful in multi-tenant clusters or when you need to enforce security boundaries between workloads.

Note : All above are pods

Key Concepts :

1. **Pods and Labels**: Network Policies apply to pods, which are selected using labels (e.g., app: web).
2. **Ingress and Egress**:
    - *Ingress*: Rules for incoming traffic to a pod.

- *Egress*: Rules for outgoing traffic from a pod.

3. **Namespaces**: Policies can apply within a namespace and can reference pods in other namespaces.
4. **CIDR Blocks**: You can allow or deny traffic based on IP ranges (e.g., 192.168.1.0/24).
5. **Ports and Protocols**: Policies can specify which ports (e.g., 80) and protocols (e.g., TCP, UDP) are allowed.

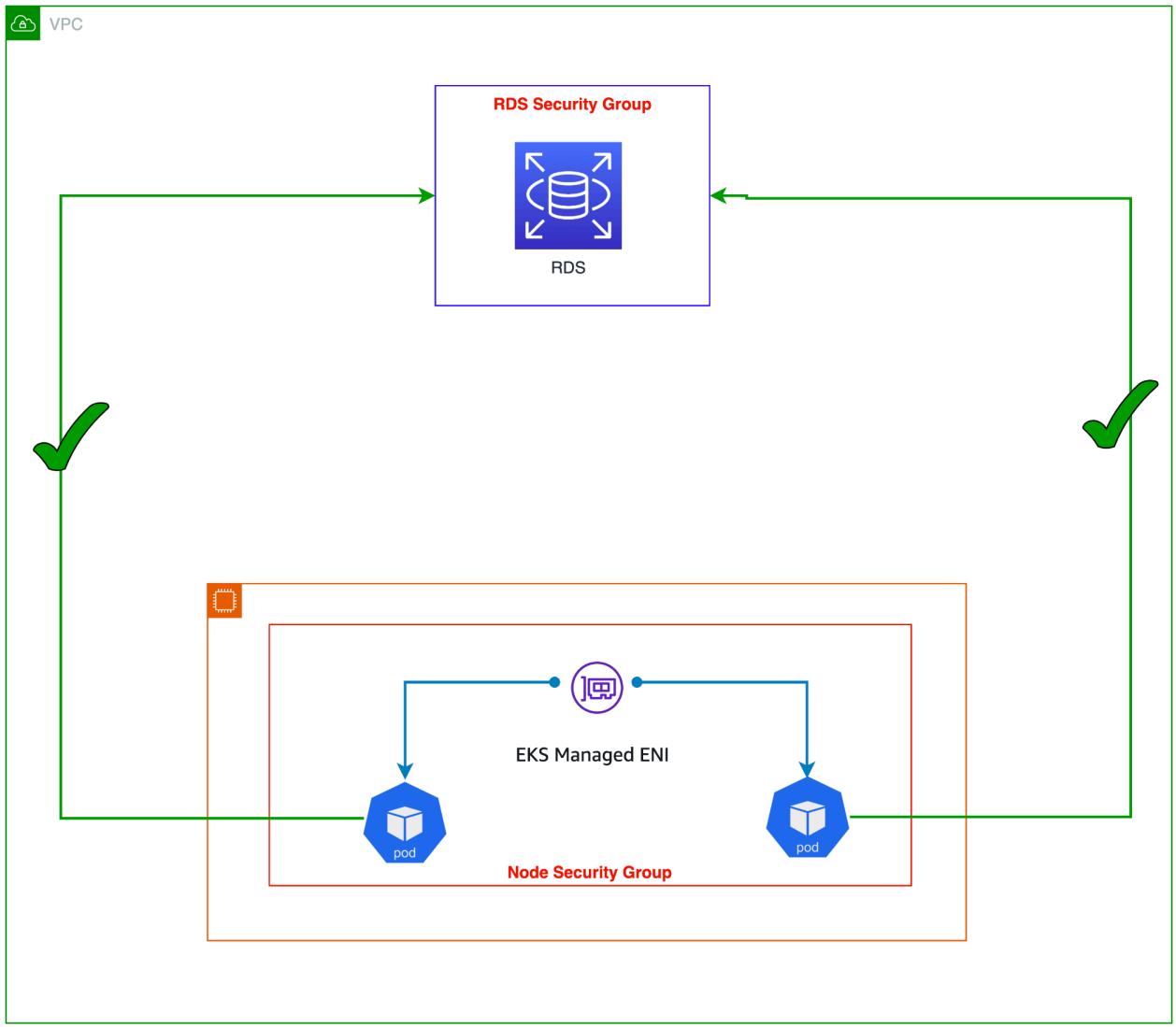
How Network Policies Work :

1. **Default Behavior**: Without a Network Policy, all pods can communicate freely within the cluster.
2. **Policy Enforcement**: Once a Network Policy is applied to a pod (via label selectors), it becomes isolated unless explicitly allowed by the policy.
3. **Allow Rules**: You define what traffic is permitted. Anything not explicitly allowed is denied (implicit deny).

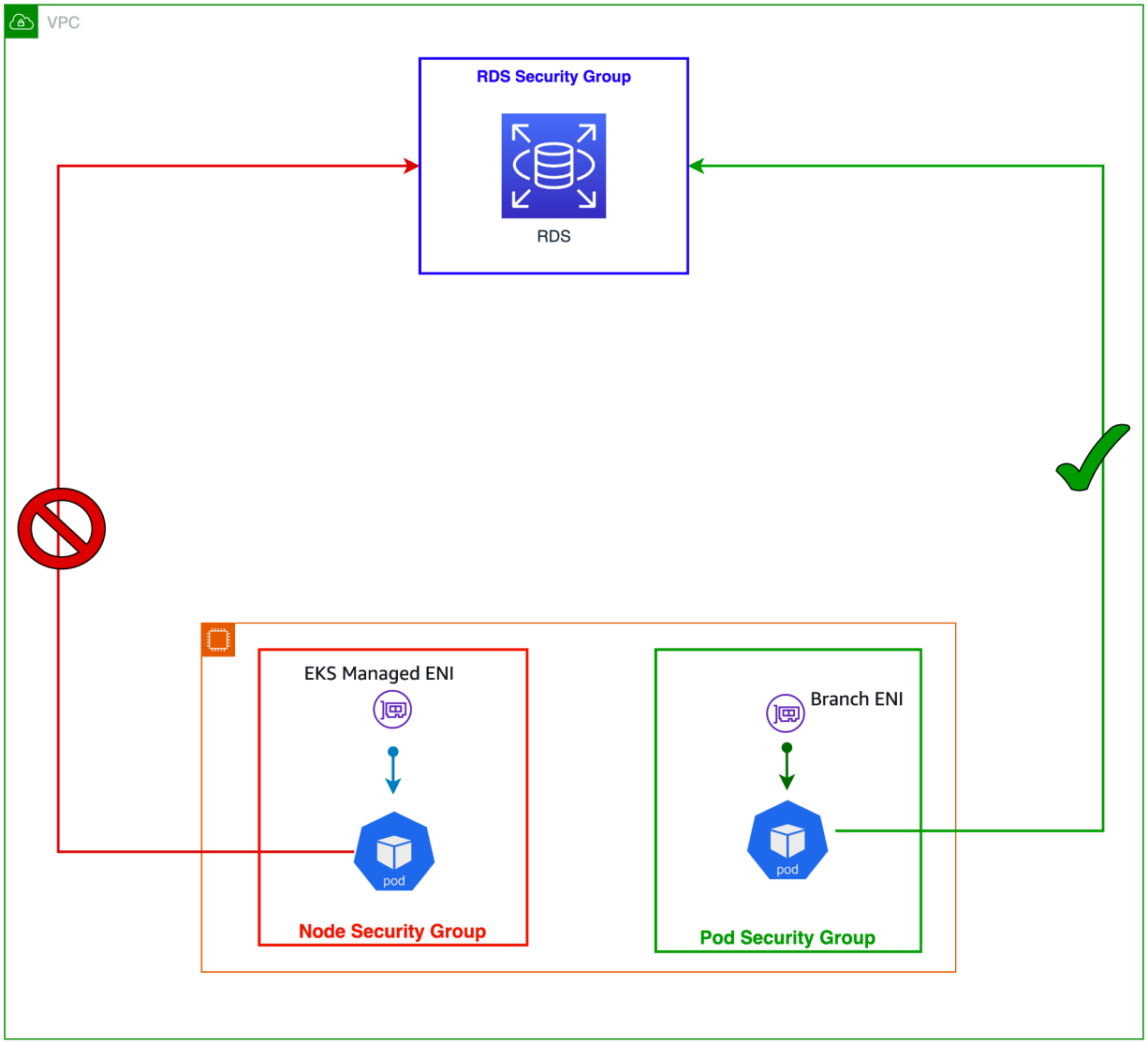
For example:

- Allow only pods labeled app: frontend to talk to pods labeled app: backend on port 80.
- Block all other traffic to app: backend.

12.5.2: Security Groups for Pods :



Before


VPC
RDS Security Group
RDS
EKS Managed ENI
pod
Node Security Group
Branch ENI
pod
Pod Security Group

After

Security Groups for Pods:

What: Assigns EC2 security groups to individual Kubernetes pods in EKS for fine-grained network control.

Purpose: Enhances pod-level network security, integrates with AWS services (e.g., RDS, ElastiCache).

How It Works :

1. **Components**:
    - **Trunk Interface**: Primary ENI on worker node.
    - **Branch Interface**: Secondary ENI per pod, tied to a security group.
    - **VPC CNI Plugin**: Manages IP allocation and ENI attachment.
    - **SecurityGroupPolicy**: Kubernetes CRD to link pods to security groups.
    - **VPC Resource Controller**: Allocates branch ENIs.
2. **Process**:
    - Pod scheduled → VPC CNI assigns IP → SecurityGroupPolicy applies security group via branch ENI.
    - Traffic follows pod-specific security group rules.
3. **Traffic Modes** (VPC CNI 1.11+):
    - strict: Only pod's security group rules apply.
    - standard: Fallback to node's security group for external traffic.

Key Benefits

- Granular traffic control (pod-to-pod, pod-to-AWS services).
- Network segmentation for compliance/security.
- Reuse existing security group policies.
- Efficient resource sharing on nodes.

Requirements

- **EKS Cluster**: Version 1.17+ (platform eks.3+ recommended).
- **VPC CNI**: Version 1.7.7+ (1.11+ for full features).

- **Instance Types**: Nitro-based (e.g., m5, c5, r5, m6g); no t-family support.
- **Fargate**: Supported with 1.7.7+ (IPv6 only initially, check for IPv4 updates).

Limitations

- **Pod Limits**: Maximum pods with custom security groups capped by branch ENIs per instance (e.g., **m5.large = 9 branch ENIs**; total pods may drop to 27 vs. default 29 due to IP constraints).
- **Supported Instances**: Restricted to **Nitro-based types** (e.g., m5, c5, r5, m6g); no t-family (e.g., t3) support.
- **IP Constraints**: Limited by total secondary IPs (e.g., **27 for m5.large**); branch ENIs reduce capacity for standard pods.
- **No Windows Node Support**: Unavailable on Windows worker nodes or Fargate Windows.
- **Fargate Restrictions**: Supported only with VPC CNI **1.7.7+** (initially IPv6; check for IPv4 updates); no Windows Fargate.
- **CNI Dependency**: Requires compatible VPC CNI (e.g., **1.7.7+** for basic, **1.11+** for strict/standard modes).
- **Cluster Version**: Needs EKS **1.17+** (platform **eks.3+** or later).
- **Resource Overhead**: Branch ENI creation may delay pod startup; consumes VPC IP space faster.
- **Configuration Complexity**: Managing SecurityGroupPolicy adds overhead; misconfiguration can block traffic.

12.6.0 : EKS Addons

Installation mode : Cli and GUI

Overview

- **Definition**: EKS add-ons are software providing supporting capabilities to Kubernetes applications, such as networking, observability, storage, and security, without being application-specific.
- **Purpose**: They simplify the installation, configuration, and management of critical tools for EKS clusters.
- **Managed by**: Add-ons can be Amazon EKS-managed (lifecycle handled by AWS) or self-managed (user-managed).

Default Add-ons

- **Automatically Installed**: Every EKS cluster comes with self-managed versions of:
    - **Amazon VPC CNI Plugin**: Handles pod networking by assigning VPC IP addresses. Not compatible with EKS Hybrid Nodes.
    - **CoreDNS**: A DNS server for cluster name resolution, deploying two replicas by default.
    - **Kube-proxy**: Maintains network rules on nodes for pod communication.

Types of Add-ons

1. **AWS Add-ons**:
    - Built and fully supported by AWS (e.g., Amazon EFS CSI driver, AWS Load Balancer Controller).
    - Integrate with other AWS services.
2. **AWS Marketplace Add-ons**:

- Provided by independent vendors, scanned by AWS (e.g., Splunk for monitoring).

3. **Community Add-ons**:
   - Open-source, scanned by AWS but community-supported (e.g., Kubernetes Metrics Server).

Key Features

- **Configuration**:
  - Customizable fields not managed by EKS can be adjusted (e.g., via configurationValues in JSON format).
  - Use tools like aws eks describe-addon-configuration to view customizable schema.
- **IAM Permissions**:
  - Requires IAM roles for service accounts (IRSA) or EKS Pod Identity for add-ons needing AWS resource access.
  - Example: VPC CNI requires AmazonEKS_CNI_Policy for IPv4 or a custom policy for IPv6.
- **Installation**:
  - Added via AWS CLI, eksctl, or AWS Management Console.
  - Example: eksctl create addon --cluster my-cluster --name vpc-cni --version latest.
- **Updates**:
  - Not automatic; users must initiate updates to newer versions.
  - Example: Update from 1.28.x to 1.30.x requires stepping through 1.29.x.

12.6.1 : EKS addon - - -> EKS EBS driver

Install this addon driver if it is not installed , We will need this driver to create persistent storage.

So let's start the installation procedure.

Create Iam Service Account

Syntax : [Replace Your Values]

```
eksctl create iamserviceaccount \
  --name ebs-csi-controller-sa \
  --namespace kube-system \
  --cluster < your-cluster-name > \
  --attach-policy-arn arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDriverPolicy \
  --approve \
  --role-only \
  --role-name AmazonEKS_EBS_CSI_Driver_Role \
  --region < region-in-which-eks-cluster-is >
```

Install Addon :

Syntax : [Replace Your Values]

```
eksctl create addon --name aws-ebs-csi-driver \
  --cluster < your-cluster-name > \
  --service-account-role-arn arn:aws:iam::<yours-aws-account-id>:role/AmazonEKS_EBS_CSI_Driver_Role \
  --region < region-in-which-eks-cluster-is > \
```

==========

```
  --force
```

Create Storage Class :

ebs-storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: ebs-sc
 annotations:
   storageclass.kubernetes.io/is-default-class: "true"
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
parameters:
 type: gp3
 encrypted: "true"
```

```
kubectl apply -f ebs-storageclass.yaml
```

12.6.2 : EKS addon - - -> Kubecost

Ref. : https://docs.aws.amazon.com/eks/latest/userguide/cost-monitoring-kubecost.html

Amazon EKS supports Kubecost, which you can use to monitor your costs broken down by Kubernetes resources including Pods, nodes, namespaces, and labels.

As a Kubernetes platform administrator and finance leader, you can use Kubecost to visualize a breakdown of Amazon EKS charges, allocate costs, and charge back organizational units such as application teams. You can provide your internal teams and business

units with transparent and accurate cost data based on their actual AWS bill. Moreover, you can also get customized recommendations for cost optimization based on their infrastructure environment and usage patterns within their clusters.

Installation :

Syntax :

```
aws eks create-addon \
  --cluster-name <cluster-name> \
  --addon-name xyz_addon \
  --region <your-region>
```

Search Addons :

```
vishal@vishalk17:~/Downloads/vish/eks$ aws eks describe-addon-versions --region ap-south-1 | grep -i kubecost
            "addonName": "kubecost_kubecost",
            "publisher": "kubecost",
```

Install kubecost :

```
aws eks create-addon \
  --cluster-name <cluster-name> \
  --addon-name kubecost_kubecost \
  --region <your-region>
```

Install svc to access dashboards externally :

kubecost-svc.yaml :

```
apiVersion: v1
kind: Service
```

==========

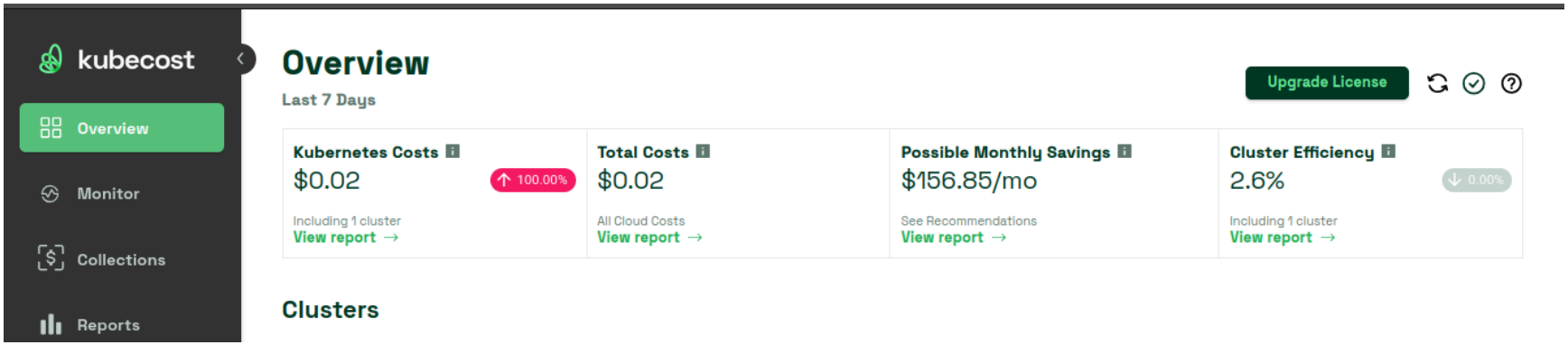
```
metadata:
 name: cost-analyzer-nlb
 namespace: kubecost
 annotations:
   # Specify NLB as the load balancer type
   service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
   # Make it internet-facing
   service.beta.kubernetes.io/aws-load-balancer-scheme: "internet-facing"
spec:
 type: LoadBalancer
 selector:
   # Targets the cost-analyzer pods
   app: cost-analyzer
 ports:
   - port: 9090
     targetPort: 9090
     protocol: TCP
     name: http

---
apiVersion: v1
kind: Service
metadata:
 name: cost-analyzer-grafana-nlb
 namespace: kubecost
 annotations:
   # Specify NLB as the load balancer type
   service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
   # Make it internet-facing
   service.beta.kubernetes.io/aws-load-balancer-scheme: "internet-facing"
spec:
 type: LoadBalancer
```

```
selector:
  # Targets the cost-analyzer-grafana pods
  app: cost-analyzer-grafana
ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
```

```
kubectl apply -f kubecost-svc.yaml
```

```
service/cost-analyzer-grafana-nlb created
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get svc -n kubecost
NAME                            TYPE           CLUSTER-IP       EXTERNAL-IP                                                                                                  PORT(S)             AGE
cost-analyzer                   ClusterIP      10.100.42.170    <none>                                                                                                       9003/TCP,9090/TCP   10m
cost-analyzer-aggregator        ClusterIP      10.100.234.210   <none>                                                                                                       9004/TCP            10m
cost-analyzer-cloud-cost        ClusterIP      10.100.95.199    <none>                                                                                                       9005/TCP            10m
cost-analyzer-forecasting       ClusterIP      10.100.232.159   <none>                                                                                                       5000/TCP            10m
cost-analyzer-grafana           ClusterIP      10.100.203.29    <none>                                                                                                       80/TCP              10m
cost-analyzer-grafana-nlb       LoadBalancer   10.100.33.214    a4edd48332e00497eab6d2c1d634808e-a0b2c30b31a4665c.elb.ap-south-1.amazonaws.com                               80:30293/TCP        14s
cost-analyzer-nlb               LoadBalancer   10.100.198.167   a1dbdb329f2f04cf9a501b4b5c431014-681246cd4f5d6b58.elb.ap-south-1.amazonaws.com                               9090:31492/TCP      15s
cost-analyzer-prometheus-server ClusterIP      10.100.152.20    <none>                                                                                                       80/TCP              10m
vishal@vishalk17:~/Downloads/vish/eks$
```

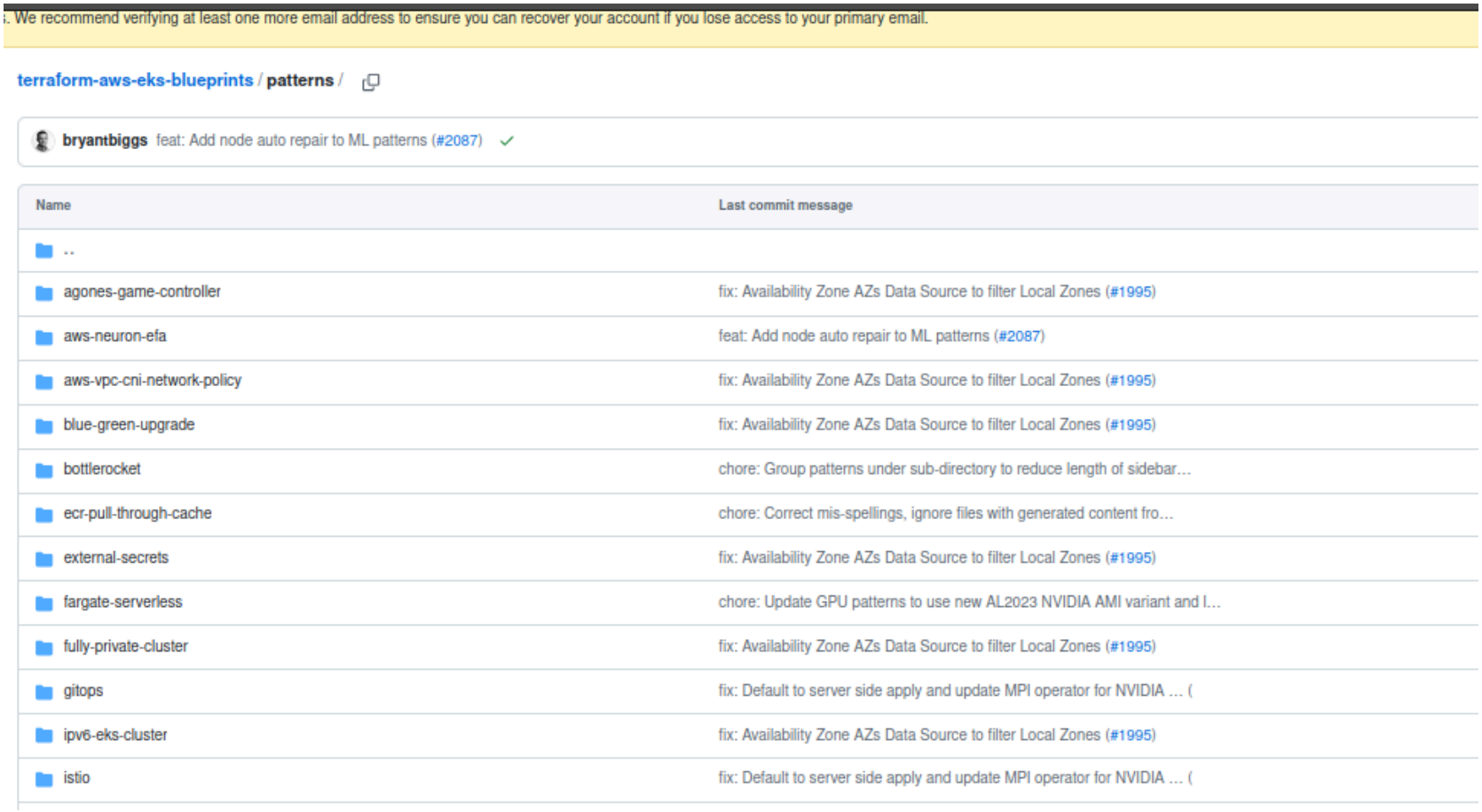
Now you can access them via `Loadbalancer_domain:port`



12.7.0 : EKS Blueprints :

This project contains a collection of Amazon EKS cluster patterns implemented in Terraform that demonstrate how fast and easy it is for customers to adopt Amazon EKS. The patterns can be used by AWS customers, partners, and internal AWS teams to configure and manage complete EKS clusters that are fully bootstrapped with the operational software that is needed to deploy and operate workloads.

Ref. : https://github.com/aws-ia/terraform-aws-eks-blueprints



12.8.0 : Kubernetes Admission Controller / Webhooks -> OPA/ Kyverno :

1. What is an Admission Controller in Kubernetes?

Definition:

Admission Controllers are plugins in Kubernetes that intercept requests to the Kubernetes API server (e.g., creating a Pod, updating a Deployment) before they are persisted or executed.

Purpose:

They enforce rules, validate configurations, or modify objects to ensure security, compliance, or operational best practices.

When do they run?

Admission Controllers execute **after authentication and authorization** but **before the object is stored in etcd** (Kubernetes' database).

2. What are Webhooks in Admission Controllers?

Definition:

Webhooks extend Kubernetes' built-in admission control by calling an external service (via HTTP) to validate or mutate objects.

How It Works:

1. The Kubernetes API server receives a request (e.g., `kubectl apply -f pod.yaml`).
2. Instead of relying only on built-in logic, it sends the request details (as JSON) to an external webhook server.
3. The webhook server processes it and responds with:
    - **"Allow" or "Deny"** → For validation.
    - **A modified object** → For mutation.

Types of Admission Webhooks

1. MutatingAdmissionWebhook:

- Modifies the resource.
- Examples:
    - Adds labels.
    - Injects sidecars.
    - Sets default values.

2. ValidatingAdmissionWebhook:

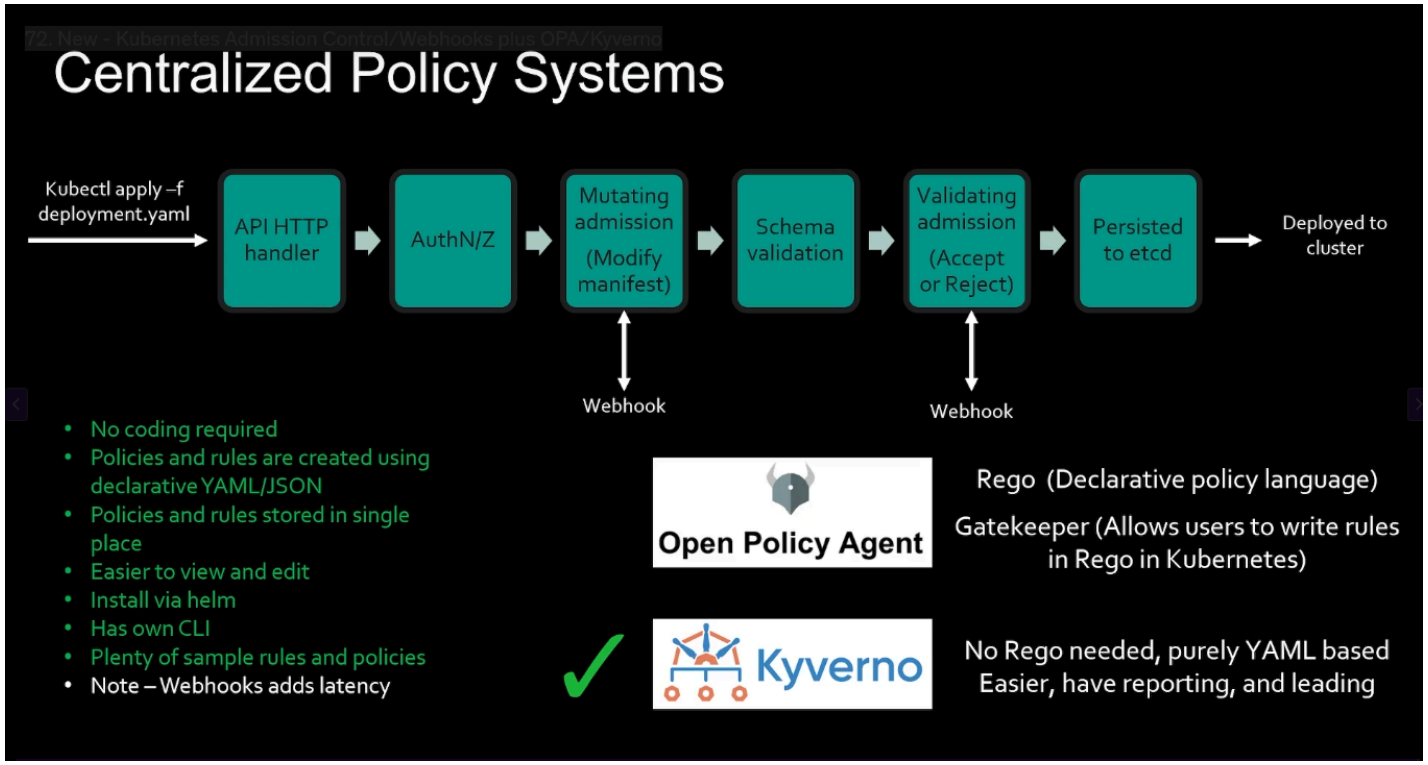
- Validates the resource.
- Examples:
    - Rejects Pods without certain annotations.
    - Enforces security policies.

Use Cases of Admission Webhooks

✅ Enforce security policies (e.g., reject Pods that don't use a specific security context).
✅ Inject sidecars (e.g., for service mesh like Istio).
✅ Enforce naming conventions or metadata standards.

OPA vs. Kyverno :

Feature	OPA (Open Policy Agent)	Kyverno
Policy Language	Rego (custom language)	YAML (Kubernetes-style)
Scope	General-purpose (K8s + beyond)	Kubernetes-specific
Ease of Use	Harder (learning curve)	Easier (familiar to K8s users)
Flexibility	Very flexible (supports external APIs, services)	Less flexible (works only in Kubernetes)
Mutation Support	❌ No (only validation)	✅ Yes (can modify resources)
Validation Support	✅ Yes	✅ Yes
Best Use Case	Complex, cross-platform policies	Simple Kubernetes policies


72. New - Kubernetes Admission Control/Webhooks plus OPA/Kyverno
Centralized Policy Systems
Kubectl apply –f deployment.yaml
API HTTP handler
AuthN/Z
Mutating admission (Modify manifest)
Schema validation
Validating admission (Accept or Reject)
Persisted to etcd
Deployed to cluster
Webhook
Webhook
No coding required
Policies and rules are created using declarative YAML/JSON
Policies and rules stored in single place
Easier to view and edit
Install via helm
Has own CLI
Plenty of sample rules and policies
Note – Webhooks adds latency
Open Policy Agent
Rego (Declarative policy language)
Gatekeeper (Allows users to write rules in Rego in Kubernetes)
Kyverno
No Rego needed, purely YAML based
Easier, have reporting, and leading

12.9.0 : EKS Upgrades :

Why Upgrade?

Kubernetes releases new minor versions approximately every four months, and Amazon EKS follows soon after. Keeping your EKS cluster up to date ensures:

- **Enhanced Security** – Addresses vulnerabilities and strengthens protection.
- **New Features** – Access to the latest Kubernetes capabilities.
- **Bug Fixes** – Improved stability and performance.
- **Avoid Forced Disruption:** If you don't upgrade, AWS eventually steps in, which can disrupt your cluster if you're unprepared.
-

12.9.1: EKS Support Lifecycle

- AWS provides **14 months** of standard support for each Kubernetes version.
- An **additional 12 months** of extended support is available at **$0.50 per cluster hour**.
- After **26 months**, clusters are **automatically upgraded** to the next supported version, which may disrupt unprepared workloads.

12.9.2 : Key Considerations Before Upgrading

1. **No Downgrades** – Once upgraded, you **cannot revert** to a previous version. Always test upgrades in a non-production environment first.
2. **Version Progression** – Upgrades must be performed **one minor version at a time** (e.g., 1.28 → 1.29, not 1.28 → 1.30 directly).
3. **Compatibility** – Ensure your applications, add-ons (e.g., VPC CNI, CoreDNS, kube-proxy), and dependencies are compatible with the new version.
4. **Node Updates** – The control plane and worker nodes should ideally run the **same minor version**, though nodes can temporarily lag up to two versions behind.

12.9.3 : Upgrade Process

1. Upgrade the Control Plane

- Initiate via the **AWS Console**, **AWS CLI**, or **eksctl**.
- AWS replaces API server nodes with the new version while maintaining **high availability**.
- This process takes **minutes to hours**, depending on the cluster size, but minimal disruption occurs if Kubernetes clients handle reconnections properly.

Command Example:

```
aws eks update-cluster-version --region <your-region> --name <cluster-name> --kubernetes-version <new-version>
```

Or using `eksctl`:

```
eksctl upgrade cluster --name <cluster-name> --version <new-version>
```

2. Upgrade Worker Nodes

Managed Node Groups

- AWS EKS **automates rolling updates**, respecting **Pod Disruption Budgets (PDBs)** to minimize downtime.
- Upgrade using AWS CLI:

```
aws eks update-nodegroup-version --cluster-name <cluster-name> --nodegroup-name <nodegroup-name>
```

Or with `eksctl`:

```
eksctl upgrade nodegroup --name <nodegroup-name> --cluster <cluster-name>
```

==========

Self-Managed Nodes

1. Create **new nodes** with the latest EKS-optimized AMI.
2. **Cordon** old nodes to prevent scheduling:

```
kubectl cordon <node-name>
```

3. **Drain workloads** before terminating old nodes:

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

4. **Detach old nodes** after verifying workloads are running on new nodes.

3. Upgrade Kubernetes Add-Ons

Upgrade essential components like:

- VPC CNI Plugin
- CoreDNS
- Kube-proxy

Command Example:

```
eksctl utils update-kube-proxy --cluster <cluster-name>
eksctl utils update-coredns --cluster <cluster-name>
eksctl utils update-aws-node --cluster <cluster-name>
```

Best Practices :

1. **Test First** – Use a **staging cluster** to validate application behavior before production upgrades.
2. **Backup** – Tools like **Velero** can help snapshot cluster state before upgrading.
3. **Automate** – Use **eksctl**, Managed Node Groups, or **Karpenter** to streamline upgrades.
4. **Monitor Version Lifecycle** – Track Kubernetes support timelines

Tools & Insights :

- **EKS Upgrade Insights** (available in the AWS Console) detects compatibility issues before upgrading.
- **Extended Support** provides additional time (up to **26 months**), but plan upgrades proactively to avoid unexpected auto-upgrades.

By following these steps, you can ensure a **smooth, secure, and efficient** EKS upgrade process while minimizing disruptions. 🚀

13.0.0 : Securing EKS Cluster :

Securing an Amazon Elastic Kubernetes Service (EKS) cluster is crucial for protecting your applications and data.

Ref.: https://github.com/vishalk17/devops/tree/main/kubernetes/2-after-k8s-learn/rbac [RBAC >> notes, examples yamls]

13.1.0 : RBAC :

1. User

- **What It Is**: A user represents a human or external entity (e.g., a developer or admin) interacting with the Kubernetes cluster, typically via kubectl or API calls. Users are not managed as Kubernetes resources—they're defined externally (e.g., via certificates, tokens, or an identity provider like AWS IAM in EKS).
- **Scope**: Cluster-wide (but permissions are controlled by RBAC).
- **How It Works**: Users authenticate using credentials (e.g., client certificates or AWS IAM credentials in EKS). Their actions are authorized based on RBAC rules tied to their identity.

2. Group

- **What It Is**: A group is a collection of users, defined externally (e.g., via an identity provider). Groups are used to manage permissions for multiple users at once in Kubernetes RBAC.
- **Scope**: Cluster-wide (permissions are applied via RBAC).
- **How It Works**: Groups are specified in the aws-auth ConfigMap (in EKS) or an external system. A RoleBinding or ClusterRoleBinding assigns permissions to the group, and all users in that group inherit those permissions.

3. Role

- **What It Is**: A Role is a Kubernetes resource that defines a set of permissions (e.g., what actions can be performed on which resources) within a single namespace.
- **Scope**: Namespace-specific.
- **How It Works**: A Role specifies apiGroups, resources (e.g., pods, services), and verbs (e.g., get, list, create). It's bound to a user, group, or Service Account via a RoleBinding.

- **Example**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Key Point: Roles are used for fine-grained access control within a namespace.

4. RoleBinding

- **What It Is**: A RoleBinding is a Kubernetes resource that binds a Role to a subject (user, group, or Service Account) within the same namespace, granting the subject the Role's permissions.
- **Scope**: Namespace-specific (matches the Role's namespace).
- **How It Works**: It lists the subjects (e.g., vishal, dev-team, or app-sa) and references the Role. The subject inherits the Role's permissions in that namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: default
  name: pod-reader-binding
subjects:
- kind: User
  name: vishal
```

```
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Key Point: RoleBindings apply namespace-scoped permissions to specific subjects.

5. ClusterRole

- **What It Is**: A ClusterRole is a Kubernetes resource that defines permissions across the entire cluster, not limited to a single namespace. It can apply to namespace-scoped resources (e.g., pods) or cluster-scoped resources (e.g., nodes).
- **Scope**: Cluster-wide.
- **How It Works**: Like a Role, it specifies apiGroups, resources, and verbs, but its permissions extend to all namespaces or cluster-level resources. It's bound via a ClusterRoleBinding.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Key Point: ClusterRoles are used for broad access, such as cluster administration or cross-namespace operations.

6. ClusterRoleBinding

- **What It Is**: A ClusterRoleBinding is a Kubernetes resource that binds a ClusterRole to a subject (user, group, or Service Account) across the entire cluster, granting cluster-wide permissions.
- **Scope**: Cluster-wide (matches the ClusterRole's scope).
- **How It Works**: It lists the subjects and references the ClusterRole. The subject inherits the ClusterRole's permissions across all namespaces and cluster resources.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding
subjects:
- kind: ServiceAccount
  name: app-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Key Point: ClusterRoleBindings enable cluster-wide access for subjects.

7. Service Account

- **What It Is**: A Service Account is a Kubernetes resource that provides an identity for pods (not humans) to authenticate with the Kubernetes API. It's created within a namespace and generates a token for API access.
- **Scope**: Namespace-scoped (exists in a specific namespace, e.g., default).
- **How It Works**: A Service Account is assigned to a pod via serviceAccountName. Its token is mounted into the pod (at /var/run/secrets/kubernetes.io/serviceaccount/token), allowing the pod to call the API. Permissions are granted via RoleBindings or ClusterRoleBindings.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-sa
  namespace: default
---
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: default
spec:
  serviceAccountName: app-sa
  containers:
  - name: my-container
    image: nginx
```

Key Point: Service Accounts are for pod authentication, and their permissions can be namespace-specific (Role) or cluster-wide (ClusterRole).

8. IRSA (IAM Roles for Service Accounts) :

A. What is an IAM Role and How Does It Work?

An **IAM Role** in AWS is like a temporary permission set that an entity (e.g., a user, an EC2 instance, or a pod in EKS) can assume to perform actions on AWS resources (like S3). Unlike IAM users (which have static credentials), IAM roles provide temporary security credentials via the AWS Security Token Service (STS).

B. What It Is IRSA

- **IAM Roles for Service Accounts (IRSA)** is an EKS-specific feature that allows pods to assume AWS IAM roles to access AWS services (e.g., S3, DynamoDB) securely, without hardcoding AWS credentials.
- **Scope**: Namespace-scoped (the Service Account is tied to a namespace), but the IAM role's permissions are AWS-wide, not Kubernetes-specific.

C. Key Concepts:

- **Trust Policy**: Defines who can assume the role (e.g., a specific Service Account in EKS via OIDC).
- **Permission Policy**: Specifies what the role can do (e.g., read from an S3 bucket).
- **Assume Role**: The entity (e.g., a pod) uses a token to call sts:AssumeRoleWithWebIdentity and gets temporary credentials (access key, secret key, session token).
- **IRSA Integration**: In EKS, IRSA uses OpenID Connect (OIDC) to let pods assume IAM roles through a Service Account, avoiding hardcoded credentials.

D. How It Works in EKS with IRSA:

1. **OIDC Setup**: Your EKS cluster has an OIDC provider (e.g., oidc.eks.us-west-2.amazonaws.com/id/1234).
2. **IAM Role Creation**: You create an IAM role with a trust policy allowing a specific Service Account to assume it.
3. **Service Account Annotation**: The Service Account is annotated with the IAM role ARN.
4. **Pod Execution**: The pod uses the Service Account, and the EKS admission controller injects the role ARN and OIDC token. The AWS SDK/CLI in the pod assumes the role and gets temporary credentials to access AWS services.

Demo :

EKS Cluster: Assume your cluster is named `vishalk17-cluster-demo`
New Namespace: We'll use test-ns instead of default.
S3 Bucket: We'll create a new bucket named my-test-bucket
AWS Account ID: Replace <account-id> with your AWS account ID (e.g., 123456789012).
OIDC Provider: Verify it with
aws eks describe-cluster --name my-eks-cluster --query "cluster.identity.oidc.issuer" --output text

==========

Create the S3 Bucket s3://vishalk17-dir-123 in ap-south-1:

Create the bucket & upload some files in it :

```
aws s3 mb s3://vishalk17-dir-123 --region ap-south-1
```

Verify it:

```
aws s3 ls --region ap-south-1
```

Verify the OIDC issuer URL:

```
aws eks describe-cluster --name <cluste-name> --region <your-region>--query
"cluster.identity.oidc.issuer" --output text
```

Terminal output :

```
vishal@vishalk17:~/Downloads/vish/eks$ aws eks describe-cluster --name vishalk17-cluster-demo --region
ap-south-1 --query "cluster.identity.oidc.issuer" --output text

https://oidc.eks.ap-south-1.amazonaws.com/id/511EC565B0B2DC9EB51BBAE77B2964F8
```

Create the New Namespace:

```
kubectl create namespace test-ns
```

Create an IAM Policy for S3 Access:

Policy to allow listing and getting objects from `vishalk17-dir-123`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:ListBucket", "s3:GetObject"],
      "Resource": ["arn:aws:s3:::vishalk17-dir-123", "arn:aws:s3:::vishalk17-dir-123/*"]
    }
  ]
}
```

Save as s3-policy.json and create the policy:

```
aws iam create-policy --policy-name S3TestAccessPolicy --policy-document file://s3-policy.json
```

```
vishal@vishalk17:~/Downloads/vish/eks/irsa$ vim s3-policy.json
vishal@vishalk17:~/Downloads/vish/eks/irsa$ aws iam create-policy --policy-name S3TestAccessPolicy
--policy-document file://s3-policy.json

{
    "Policy": {
        "PolicyName": "S3TestAccessPolicy",
        "PolicyId": "ANPAXYKJUXNDJAZXXMMKQ",
        "Arn": "arn:aws:iam::533267266374:policy/S3TestAccessPolicy",
        "Path": "/",
        "DefaultVersionId": "v1",
```

==========

```
        "AttachmentCount": 0,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "CreateDate": "2025-03-18T13:58:03+00:00",
        "UpdateDate": "2025-03-18T13:58:03+00:00"
    }
}
```

Note the ARN (e.g., arn:aws:iam::<account-id>:policy/S3TestAccessPolicy).

Create the Service Account and IAM Role with eksctl:

- Use eksctl create iamserviceaccount to automate the process:

```
eksctl create iamserviceaccount \
  --name app-sa \
  --namespace test-ns \
  --cluster <your-cluster> \
  --region <your-region> \
  --attach-policy-arn arn:aws:iam::<account-id>:policy/S3TestAccessPolicy \
  --approve
```

What This Does:

- Creates a Service Account named app-sa in test-ns.
- Creates an IAM role with a trust policy for system:serviceaccount:test-ns:app-sa.
- Attaches the `S3TestAccessPolicy` to the role.
- Annotates the Service Account with the IAM role ARN.

```
vishal@vishalk17:~/Downloads/vish/eks/irsa$ eksctl create iamserviceaccount \
  --name app-sa \
  --namespace test-ns \
  --cluster vishalk17-cluster-demo \
  --region ap-south-1 \
  --attach-policy-arn arn:aws:iam::533267266374:policy/S3TestAccessPolicy \
  --approve
2025-03-18 20:35:37 [ℹ]  1 iamserviceaccount (test-ns/app-sa) was included (based on the include/exclude rules)
2025-03-18 20:35:37 [!]  serviceaccounts that exist in Kubernetes will be excluded, use --override-existing-serviceaccounts to override
2025-03-18 20:35:37 [ℹ]  1 task: {
    2 sequential sub-tasks: {
        create IAM role for serviceaccount "test-ns/app-sa",
        create serviceaccount "test-ns/app-sa",
    } }2025-03-18 20:35:37 [ℹ]  building iamserviceaccount stack "eksctl-vishalk17-cluster-demo-addon-iamserviceaccount-test-ns-app-sa"
2025-03-18 20:35:37 [ℹ]  deploying stack "eksctl-vishalk17-cluster-demo-addon-iamserviceaccount-test-ns-app-sa"
2025-03-18 20:35:37 [ℹ]  waiting for CloudFormation stack "eksctl-vishalk17-cluster-demo-addon-iamserviceaccount-test-ns-app-sa"
2025-03-18 20:36:07 [ℹ]  waiting for CloudFormation stack "eksctl-vishalk17-cluster-demo-addon-iamserviceaccount-test-ns-app-sa"
2025-03-18 20:36:07 [ℹ]  created serviceaccount "test-ns/app-sa"
vishal@vishalk17:~/Downloads/vish/eks/irsa$
```

Create a Pod with AWS CLI :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: aws-cli-pod
  namespace: test-ns
spec:
  serviceAccountName: app-sa
  containers:
  - name: aws-cli
    image: amazon/aws-cli
    command: ["sleep", "3600"]
```

Save as aws-cli-pod.yaml and apply:

```
kubectl apply -f aws-cli-pod.yaml
```

Test IAM Role Assumption:

==========

Exec into the pod:

```
kubectl exec -it aws-cli-pod -n test-ns -- bash
```

Check environment variables:

```
env | grep AWS
```

Test S3 access:

```
aws s3 ls s3://vishalk17-dir-123 --region ap-south-1
```

Terminal Output :

```
vishal@vishalk17:~/Downloads/vish/eks/irsa$ kubectl apply -f aws-cli-pod.yaml

pod/aws-cli-pod created
vishal@vishalk17:~/Downloads/vish/eks/irsa$ kubectl exec -it aws-cli-pod -n test-ns -- bash
bash-4.2#
bash-4.2# env | grep AWS
AWS_ROLE_ARN=arn:aws:iam::533267266374:role/eksctl-vishalk17-cluster-demo-addon-iamservic-Role1-4tpHbyWSqHrL
AWS_WEB_IDENTITY_TOKEN_FILE=/var/run/secrets/eks.amazonaws.com/serviceaccount/token
AWS_DEFAULT_REGION=ap-south-1
AWS_REGION=ap-south-1
AWS_STS_REGIONAL_ENDPOINTS=regional
bash-4.2#
bash-4.2# aws s3 ls s3://vishalk17-dir-123 --region ap-south-1
2025-03-18 14:24:20        466 minikube.txt
bash-4.2#
```

Text Diagram: IAM Role with IRSA in test-ns (ap-south-1)

```
[Pod in test-ns]  --->  [Service Account: app-sa or default]  --->  [IAM Role]  --->  [AWS S3: vishalk17-dir-123]
   |                       |                                  |                  |
   |                       |                                  |                Grants access
Uses SA                 Configured via eksctl             Assumed via OIDC
   |                       |                                  | (sts:AssumeRoleWithWebIdentity)
   |                       V                                  V
[Kubernetes API]        [Token Injected]                [Temporary Credentials]
```

Why eksctl Makes It Easier

- **Automation**: eksctl create iamserviceaccount handles the IAM role creation, trust policy setup, and Service Account annotation in one command.
- **Consistency**: Reduces the chance of manual errors in trust policies or annotations.
- **Efficiency**: Saves time compared to manual AWS IAM and Kubernetes steps.

13.2.0 : KubeConfig:

Definition: A file that kubectl uses to connect to and authenticate with a Kubernetes cluster (like your EKS cluster).
Components:

- **Clusters**: Specifies the cluster's API server URL and certificate data.
- **Users**: Defines authentication details (e.g., AWS IAM token for EKS).
- **Contexts**: Ties a cluster and user together, optionally with a namespace.

EKS Usage: Generated with:

```
aws eks update-kubeconfig --region <region> --name <cluster-name>
```

- Saves to ~/.kube/config by default.

Example:

```
apiVersion: v1
clusters:
- cluster:
    server: https://<cluster-endpoint>
    certificate-authority-data: <base64-cert>
  name: my-eks-cluster
contexts:
- context:
    cluster: my-eks-cluster
    user: my-user
  name: my-context
current-context: my-context
users:
- name: my-user
```

==========

```
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      command: aws
      args:
        - eks
        - get-token
        - --cluster-name
        - <cluster-name>
```

- **Purpose**: Allows kubectl to authenticate to EKS using AWS IAM credentials via the aws command.

13.3.0 : What is aws-auth ? :

A ConfigMap in the kube-system namespace that maps AWS IAM users/roles to Kubernetes identities in EKS.
Purpose: Handles authentication by linking IAM entities to Kubernetes usernames and groups; RBAC then controls authorization.
Structure:

- **mapRoles**: Maps IAM roles (e.g., for worker nodes) to Kubernetes groups like system:nodes.
- **mapUsers**: Maps IAM users to Kubernetes usernames and groups (e.g., system:masters for admins).

Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
```

```
    - rolearn: arn:aws:iam::<account-id>:role/EKS-Node-Role
      username: system:node:{{EC2PrivateDNSName}}
      groups:
        - system:bootstrappers
        - system:nodes
  mapUsers: |
    - userarn: arn:aws:iam::<account-id>:user/<username>
      username: <username>
      groups:
        - <group-name>
```

Editing: Use

```
kubectl edit configmap aws-auth -n kube-system
```

Key Notes:

- IAM users need eks:DescribeCluster permission to generate tokens.
- Groups like system:masters grant full access; custom groups require RBAC setup.

13.4.0 : Demo - give access to users :

Prerequisites :

1. **Two Users Access Keys and Secret Keys**:
   - You need IAM credentials for vishalk17 and chinu:
     - **vishalk17**: AWS access key and secret key (for admin access).
     - **chinu**: AWS access key and secret key (for limited access).
2. **EKS Cluster Running**: Assume an EKS cluster named <cluster-name> in region <region>

==========

1. Configure AWS for Both Users

Since both users are on the same machine, configure their credentials using AWS CLI profiles.

- **Edit** ~/.aws/credentials: Replace access key and secret access key

```
[vishalk17]
aws_access_key_id = <vishalk17-access-key>
aws_secret_access_key = <vishalk17-secret-key>

[chinu]
aws_access_key_id = <chinu-access-key>
aws_secret_access_key = <chinu-secret-key>
```

Edit ~/.aws/config (optional, for region defaults): [Replace values]

```
[profile vishalk17]
region = <region>
output = json

[profile chinu]
region = <region>
output = json
```

Verify Configuration:

```
aws configure list --profile vishalk17
aws configure list --profile chinu
```

- Ensures both profiles are set up correctly.

==========

```
vishal@vishalk17:~/Downloads/vish/eks$ aws configure list --profile vishalk17
      Name                    Value             Type    Location
      ----                    -----             ----    --------
   profile                vishalk17           manual    --profile
access_key     ****************AZ4W shared-credentials-file
secret_key     ****************hz9w shared-credentials-file
    region                ap-south-1      config-file    ~/.aws/config
vishal@vishalk17:~/Downloads/vish/eks$
vishal@vishalk17:~/Downloads/vish/eks$ aws configure list --profile chinu
      Name                    Value             Type    Location
      ----                    -----             ----    --------
   profile                   chinu           manual    --profile
access_key     ****************VQPR shared-credentials-file
secret_key     ****************WE/M shared-credentials-file
    region                ap-south-1      config-file    ~/.aws/config
vishal@vishalk17:~/Downloads/vish/eks$
```

2. Generate kubeconfig for Both Users

Set up a single kubeconfig file with contexts for vishalk17 and chinu.

- **For vishalk17** (admin):

```
aws eks update-kubeconfig --region <region> --name <cluster-name> --profile vishalk17 --alias vishalk17-context
```

- **For chinu** (limited access):

```
aws eks update-kubeconfig --region <region> --name <cluster-name> --profile chinu --alias chinu-context
```

Check kubeconfig:

```
kubectl config view
```

```
vishal@vishalk17:~/Downloads/vish/eks$ aws eks update-kubeconfig --region ap-south-1 --name vishalk17-cluster-demo --profile vishalk17 --alias vishalk17-context
Updated context vishalk17-context in /home/vishal/.kube/config
vishal@vishalk17:~/Downloads/vish/eks$ aws eks update-kubeconfig --region ap-south-1 --name vishalk17-cluster-demo --profile vishalk17 --alias chinu-context
Updated context chinu-context in /home/vishal/.kube/config
vishal@vishalk17:~/Downloads/vish/eks$
```

List Contexts:

```
kubectl config get-contexts
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl config get-contexts
CURRENT   NAME                                                   CLUSTER
          arn:aws:eks:ap-south-1:533267266374:cluster/extravagant-rock-gopher   arn:aws:eks:ap-south-1:533267266374:cluster/extravagant-rock-gopher
*         chinu-context                                        arn:aws:eks:ap-south-1:533267266374:cluster/vishalk17-cluster-demo
          vishalk17-context                                    arn:aws:eks:ap-south-1:533267266374:cluster/vishalk17-cluster-demo
          vishalk17@vishalk17-cluster-demo.ap-south-1.eksctl.io   vishalk17-cluster-demo.ap-south-1.eksctl.io
vishal@vishalk17:~/Downloads/vish/eks$
```

Now Let me switch to, admin access context which is currently holding full admin permissions of cluster ,

```
kubectl config use-context <context-name>
```

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl config use-context vishalk17@vishalk17-cluster-demo.ap-south-1.eksctl.io
Switched to context "vishalk17@vishalk17-cluster-demo.ap-south-1.eksctl.io".
vishal@vishalk17:~/Downloads/vish/eks$
vishal@vishalk17:~/Downloads/vish/eks$ kubectl config get-contexts
CURRENT   NAME                                                   CLUSTER
          arn:aws:eks:ap-south-1:533267266374:cluster/extravagant-rock-gopher   arn:aws:eks:ap-south-1:533267266374:cluster/extravagant-rock-gopher
          chinu-context                                        arn:aws:eks:ap-south-1:533267266374:cluster/vishalk17-cluster-demo
          vishalk17-context                                    arn:aws:eks:ap-south-1:533267266374:cluster/vishalk17-cluster-demo
*         vishalk17@vishalk17-cluster-demo.ap-south-1.eksctl.io   vishalk17-cluster-demo.ap-south-1.eksctl.io
vishal@vishalk17:~/Downloads/vish/eks$
```

3. Grant Admin Access to vishalk17

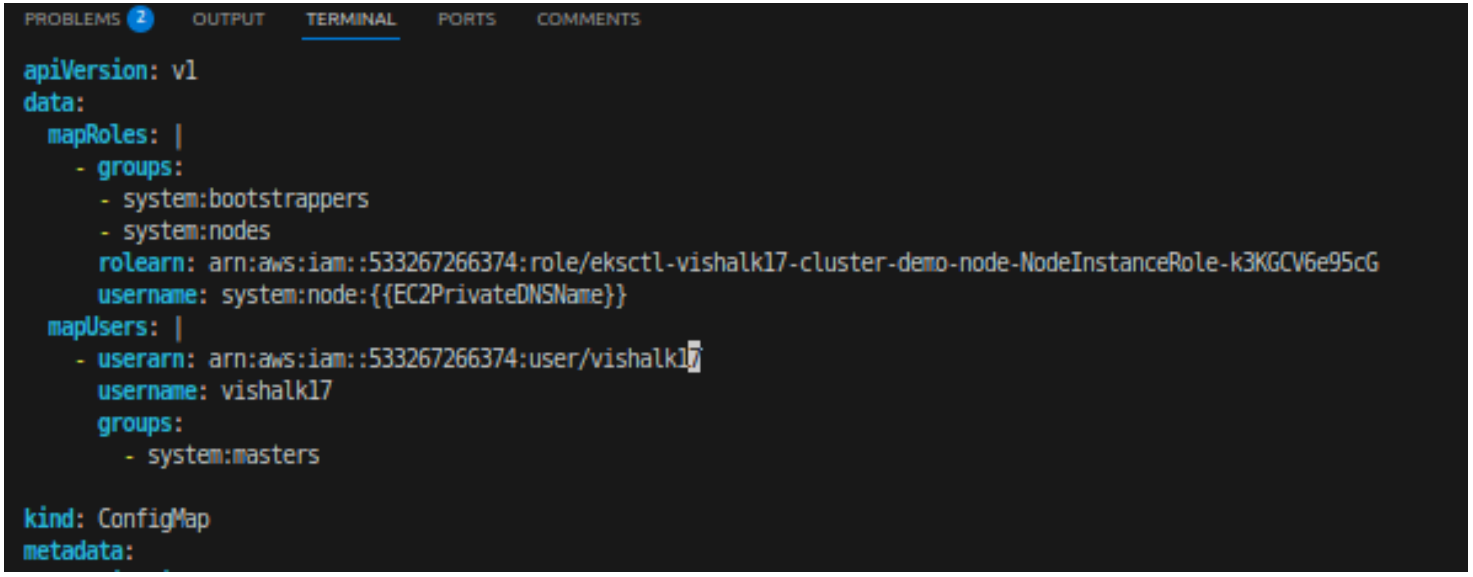
- Update aws-auth ConfigMap:

```
kubectl edit configmap aws-auth -n kube-system
```

- Add under mapUsers (replace <account-id>):

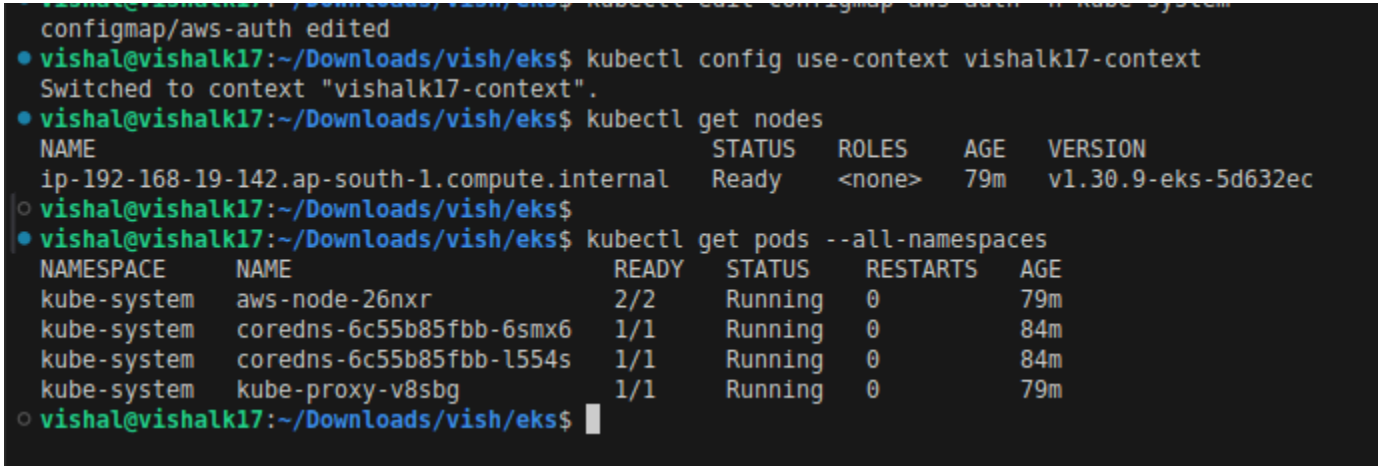
```
mapUsers: |
  - userarn: arn:aws:iam::<account-id>:user/vishalk17
    username: vishalk17
    groups:
      - system:masters
```

- Save and exit. system:masters gives full admin access



Test Access:

```
kubectl config use-context vishalk17-context
kubectl get nodes
kubectl get pods --all-namespaces
```



- Expected: Lists nodes and all pods (admin privileges confirmed).

4. Grant Limited Access to chinu (Frontend Namespace)

- Create frontend Namespace:

```
kubectl create namespace frontend
```

- Deploy Nginx (Example):

```
# nginx-deployment.yaml
```

==========

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

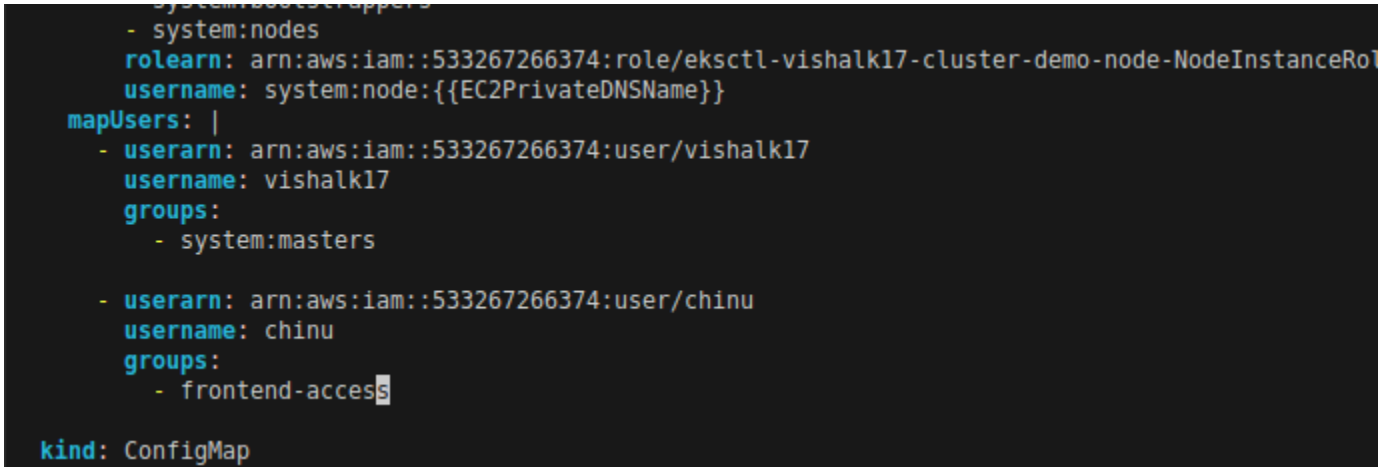
```
kubectl apply -f nginx-deployment.yaml
```

- Update aws-auth ConfigMap:

```
kubectl edit configmap aws-auth -n kube-system
```

- Add under mapUsers:

```
mapUsers: |
  - userarn: arn:aws:iam::<account-id>:user/vishalk17
    username: vishalk17
    groups:
      - system:masters
  - userarn: arn:aws:iam::<account-id>:user/chinu
    username: chinu
    groups:
      - frontend-access
```



Create Role for frontend:

```
# frontend-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

==========

```
metadata:
  namespace: frontend
  name: frontend-manager
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]
```

Bind Role to chinu:

```
# frontend-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: frontend
  name: frontend-manager-binding
subjects:
- kind: User
  name: chinu
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: frontend-manager
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl apply -f frontend-rolebinding.yaml
```

==========

Explicit Restriction:

- chinu can only access the frontend namespace (role / role binding).
- **Denied** access to:
  - Other namespaces (e.g., kube-system, default).
  - Cluster-wide resources (e.g., kubectl get nodes).

Test Access :

```
kubectl config use-context chinu-context
kubectl get pods -n default
kubectl get pods -n kube-system
kubectl get pods -n frontend
```

```
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$ kubectl get pods -n default
Error from server (Forbidden): pods is forbidden: User "chinu" cannot list resource "pods" in API group "" in the namespace "default"
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$ kubectl get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User "chinu" cannot list resource "pods" in API group "" in the namespace "kube-system"
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$ kubectl get pods -n frontend
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-576c6b7b6-czj8d   0/1     Pending   0          24m
nginx-deployment-576c6b7b6-mpd5f   0/1     Pending   0          24m
vishal@vishalk17:~/Downloads/vish/eks/iam_limited_access$
```

13.5.0 : Elastic Container Registry (ECR) image scanning :

Purpose: ECR image scanning helps identify software vulnerabilities in container images stored in your private registries. It's a critical tool for enhancing the security of containerized applications.

Scanning Types:

- **Basic Scanning**:
  - Provided by ECR at no additional cost beyond standard ECR pricing.
  - Uses the Common Vulnerabilities and Exposures (CVE) database to scan for operating system (OS) vulnerabilities.
  - Two versions exist:
    - **AWS Native Basic Scanning**: The recommended, improved version, default for new registries, offering better detection across popular OSes.
    - **Clair Basic Scanning**: An older, deprecated version using the open-source Clair project, set to lose support in all regions by October 1, 2025.
  - Options: Scan on push (automatic when an image is uploaded) or manual scans.
  - Limits: Up to 100,000 scans per 24 hours per registry; images can be scanned once every 24 hours.

Enhanced Scanning:

- Integrates with Amazon Inspector for deeper analysis.
- Scans for both OS and programming language package vulnerabilities (e.g., Python, Java, etc.).
- Offers two frequencies:
  - **Scan on Push**: Triggers a scan when an image is pushed.
  - **Continuous Scanning**: Automatically re-scans images when new CVEs are added to the Inspector database, provided the image was pushed within the last 30 days or pulled within 90 days (configurable via rescan duration).
- Billed through Amazon Inspector, not ECR, with costs based on scan activity.

Configuration:

- Basic scanning is enabled by default for new registries with the AWS native version.
- Enhanced scanning requires activation via the ECR console, AWS CLI, or API, and setting scan filters to specify which repositories to scan continuously or on push.

- Repositories not matching filters default to manual scan frequency (basic) or are not scanned (enhanced).

Findings:

- Basic scan results are viewable in the ECR console or via API (e.g., describe-image-scan-findings).
- Enhanced scan findings are accessible in both ECR and Amazon Inspector, with events sent to Amazon EventBridge for automation (e.g., notifying on new vulnerabilities).
- Images older than the configured rescan window (default 90 days for pulls) may show as "SCAN_ELIGIBILITY_EXPIRED" unless re-pushed.

13.6.0 : Container Image Best Security Practices :

Scan Your images for vulnerabilities (CVEs)

- ECR Basic Scanning (OS Packages)
- ECR enhanced scanning (OS & Programming Packages)

Use Trusted Base Images

- Start with minimal, official images from reputable sources (e.g., Docker Hub's verified images, Alpine, or distroless).
- Avoid unverified or unknown third-party images.

Minimize Image Size

- Use lightweight base images (e.g., Alpine Linux) to reduce attack surface.
- Remove unnecessary tools, libraries, or binaries (e.g., avoid debug tools like curl or vim unless essential).

Keep Images Up-to-Date

- Regularly update base images and dependencies to patch known vulnerabilities.
- Use specific version tags (e.g., node:18.12.1 instead of node:latest) for consistency and control.

Scan for Vulnerabilities

- Use tools like Trivy, Clair, or Docker Scan to identify CVEs in images.
- Integrate scanning into CI/CD pipelines and fail builds on critical vulnerabilities.

Remove Secrets from Images

- Never hardcode credentials, API keys, or sensitive data in images (e.g., in Dockerfile or env vars).
- Use runtime secrets management (e.g., Docker Secrets, Kubernetes Secrets).

Use Multi-Stage Builds

- Separate build and runtime environments to exclude build tools and temporary files from the final image.
- Example: Compile code in one stage, copy only the binary to the final stage.

Run as Non-Root User

- Add USER directive in Dockerfile to run containers as a non-privileged user.
- Avoid default root execution to limit damage from breaches.

Immutable Images

- Treat images as immutable—avoid modifying them post-build.
- Use read-only file systems where possible (e.g., --read-only flag in Docker).

Sign and Verify Images

- Use tools like Docker Content Trust (DCT) or Sigstore's Cosign to sign images.
- Verify image integrity and authenticity before deployment.

Limit Dependencies

- Install only required packages and libraries.
- Regularly audit dependencies for vulnerabilities (e.g., with npm audit or pip check).

Network Security

- Avoid exposing unnecessary ports—explicitly define only what's needed in EXPOSE.
- Use container-specific firewalls or network policies (e.g., Kubernetes NetworkPolicies).

Remove Build Artifacts

- Clear caches, temp files, and unused code after installation (e.g., apt-get clean or rm -rf /var/cache/*).
- Reduces image size and potential exploit paths.

Monitor and Log

- Enable logging in containers but avoid storing sensitive data in logs.
- Use external monitoring tools to detect anomalies in running containers.

Restrict Capabilities

- Drop unnecessary Linux capabilities (e.g., --cap-drop=all in Docker) and add only what's required (e.g., --cap-add=NET_BIND_SERVICE).
- Limits damage from compromised containers.

Follow Least Privilege

- Apply minimal permissions at every layer (file system, user, network).
- Avoid over-privileged containers in orchestration platforms like Kubernetes.

Lint Dockerfiles with Dockle or Hadolint

- Use tools like Dockle or Hadolint to analyze and lint Dockerfiles for security and best practices.
- Identifies issues like missing USER instructions, exposed ports, or inefficient RUN commands that could introduce vulnerabilities.
- Integrate into CI/CD for automated enforcement of secure, optimized Dockerfiles.

13.7.0 : EKS Worker Node Security and CIS Kube-bench Demo :

Use EKS-Optimized AMIs

- Leverage AWS-provided EKS-optimized Amazon Machine Images (AMIs) for worker nodes.
- These AMIs are pre-configured with secure defaults, compatible Kubernetes versions, and necessary bootstrapping scripts.

Harden the OS

- Regularly update the OS and apply security patches.
- Disable unused services and remove unnecessary packages.

Enable Logging and Monitoring

- Enable EKS control plane logging (API, audit, authenticator) and send logs to CloudWatch.
- Use tools like Amazon GuardDuty for runtime threat detection on worker nodes.

CIS Kube-bench Demo for EKS Worker Nodes

CIS kube-bench is an open-source tool by Aqua Security that checks whether Kubernetes is deployed securely by running the checks documented in the Center for Internet Security (CIS) Kubernetes Benchmark. For EKS, it aligns with the **CIS Amazon EKS Benchmark**, focusing on worker node security since AWS manages the control plane. Below is a step-by-step demo to run kube-bench on an EKS cluster:

Demo Steps

Step 1 : Deploy kube-bench as a Kubernetes Job

- Use the official kube-bench Job YAML tailored for EKS worker nodes:

Ref.: https://github.com/aquasecurity/kube-bench/blob/main/job-eks.yaml

The `--benchmark eks-1.0` flag tells kube-bench to use the CIS Benchmark tailored for EKS. Multiple configurations are available, and you can choose the appropriate one.

Run the following command to deploy the job:

```
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-bench/refs/heads/main/job-eks.yaml
```

Step 2: Check the Pod Status:

The Job creates a Pod. List all Pods to find it:

```
kubectl get pods
```

Example output:

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods
NAME                READY   STATUS      RESTARTS   AGE
kube-bench-x8b99    0/1     Completed   0          13s
```

The Pod name will be something like `kube-bench-<random-suffix>` (e.g., `kube-bench-x8b99`).

Wait until the **STATUS** is `Completed`, indicating that the audit has finished.

Step 3: View the Benchmark Results

Once the Pod completes, retrieve the audit results from its logs.

Get the Pod Name:

From the `kubectl get pods` output, note the exact Pod name (e.g., `kube-bench-x8b99`).

==========

View the Logs:

Use the following command to display the results:

```
kubectl logs -f kube-bench-x8b99
```

Replace `kube-bench-x8b99` with your Pod's name.

- The `-f` flag streams logs in real-time.
- Since the Job is already completed, it will simply dump the full output.

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl logs -f kube-bench-x8b99
[INFO] 2 Control Plane Configuration
[INFO] 2.1 Logging
[WARN] 2.1.1 Enable audit Logs (Automated)

== Remediations controlplane ==
2.1.1 audit test did not run: No tests defined

== Summary controlplane ==
0 checks PASS
0 checks FAIL
1 checks WARN
0 checks INFO

[INFO] 3 Worker Node Security Configuration
[INFO] 3.1 Worker Node Configuration Files
[PASS] 3.1.1 Ensure that the kubeconfig file permissions are set to 644 or more restrictive (Automated)
[PASS] 3.1.2 Ensure that the kubelet kubeconfig file ownership is set to root:root (Automated)
[PASS] 3.1.3 Ensure that the kubelet configuration file has permissions set to 644 or more restrictive (Automated)
[PASS] 3.1.4 Ensure that the kubelet configuration file ownership is set to root:root (Automated)
[INFO] 3.2 Kubelet
[PASS] 3.2.1 Ensure that the Anonymous Auth is Not Enabled (Automated)
[PASS] 3.2.2 Ensure that the --authorization-mode argument is not set to AlwaysAllow (Automated)
[PASS] 3.2.3 Ensure that a Client CA File is Configured (Automated)
[PASS] 3.2.4 Ensure that the --read-only-port is disabled (Automated)
[PASS] 3.2.5 Ensure that the --streaming-connection-idle-timeout argument is not set to 0 (Automated)
```

```
== Remediations node ==
3.2.7 If using a Kubelet config file, edit the file to set eventRecordQPS: to an appropriate
level.
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node
and set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service


== Summary node ==
12 checks PASS
1 checks FAIL
0 checks WARN
0 checks INFO

[INFO] 4 Policies
[INFO] 4.1 RBAC and Service Accounts
[WARN] 4.1.1 Ensure that the cluster-admin role is only used where required (Automated)
[WARN] 4.1.2 Minimize access to secrets (Automated)
[WARN] 4.1.3 Minimize wildcard use in Roles and ClusterRoles (Automated)
[WARN] 4.1.4 Minimize access to create pods (Automated)
[WARN] 4.1.5 Ensure that default service accounts are not actively used. ((Automated)
[WARN] 4.1.6 Ensure that Service Account Tokens are only mounted where necessary (Automated)
```

Step 4: Clean Up

After reviewing the results, delete the Job to avoid cluttering your cluster:

```
kubectl delete -f https://raw.githubusercontent.com/aquasecurity/kube-bench/refs/heads/main/job-eks.yaml
```

14.0.0 : EKS with Fargate :

AWS Fargate is a serverless compute engine for containers that integrates with Amazon Elastic Kubernetes Service (EKS) to run Kubernetes workloads without managing Amazon EC2 instances as worker nodes

No NodePort or HostPort: Fargate doesn't support these because there are no traditional nodes to expose ports on. Use LoadBalancer or Ingress instead.

Direct Pod Access: Fargate pods are in private subnets, so direct access to pod IPs isn't practical. Always use a load balancer or Ingress for external access.

14.1.0 : What is AWS Fargate in the Context of EKS?

- **Definition**: Fargate is a serverless compute platform that lets you define and deploy Kubernetes pods without provisioning or managing worker nodes.
- **Role in EKS**: AWS manages the EKS control plane (API server, scheduler, etc.), and Fargate provides a fully managed, per-pod compute environment, replacing the worker node layer.
- **Integration**: Fargate works seamlessly with both Amazon EKS and ECS, offering a serverless experience for containerized workloads.

14.2.0: How Do Pods Run in AWS Fargate Without Nodes?

In AWS **EKS with Fargate**, you don't provision or manage EC2 instances (nodes). Instead, AWS **automatically provisions the underlying infrastructure** to run your pods.

Here's how it works:

1️⃣ **No EC2 Nodes in Fargate**:

- Fargate abstracts away EC2 nodes, meaning users don't have to manage VMs.
- There's no SSH access to worker nodes, as they don't exist for the user.

2️⃣ **Fargate Uses Firecracker MicroVMs**:

- Pods run in isolated Firecracker MicroVMs, providing better security and efficiency.
- Each pod gets its own MicroVM, isolating them from others, unlike traditional Kubernetes where multiple pods share a single EC2 instance's OS.

3️⃣ **AWS Manages the Compute**:

- AWS dynamically assigns CPU and memory resources to each pod based on its needs.
- Scaling is handled automatically by AWS; users don't need to worry about underlying infrastructure.

4️⃣ **Networking with Fargate**:

- Each pod gets a dedicated ENI (Elastic Network Interface) with its own IP.
- Networking can be managed through VPC, Security Groups, and IAM roles, allowing pods to communicate with each other and integrate with other AWS services.

5️⃣ **Fargate Profile Requirement**:

- A Fargate Profile is needed to define which pods should run on Fargate based on their namespace and labels.
- Pods that don't match the profile either fail to schedule or fallback to EC2 nodes if available.

14.3.0: How Pod Scheduling Happening in the background

1. **EKS Cluster Setup**:
   - Create an EKS cluster with a supported Kubernetes version (e.g., 1.30).
2. **Fargate Profile Creation**:
   - Define a Fargate profile to specify which pods run on Fargate (by namespace and labels).
3. **Pod Scheduling**:
   - The EKS scheduler assigns matching pods to Fargate.
4. **Pod Execution**:
   - Fargate provisions a lightweight, isolated micro-VM for each pod, providing dedicated CPU, memory, and networking resources (e.g., an ENI).
   - These micro-VMs are managed by AWS and are not visible as traditional Kubernetes nodes. However, the Kubernetes API represents each Fargate pod as a **virtual node** with a name like fargate-ip-<IP>.

- When you run kubectl get nodes, you'll see one virtual node per Fargate pod. For example, if you have two pods in a namespace covered by a Fargate profile, you'll see two nodes.

5. **Networking**:
   - Uses awsvpc mode, assigning each pod an ENI.
6. **Storage**:
   - Supports EFS natively; EBS requires static provisioning.

14.4.0: Key Features of AWS Fargate with EKS

- **No Node Management**: AWS handles all infrastructure provisioning, patching, and scaling.
- **Pay-per pod-Use Pricing**: Billed per second for vCPU and memory used by pods, with no costs for idle resources.
- **Scalability & Elasticity**: Automatically provisions and deallocates compute resources as pods are created or terminated.
- **Enhanced Security**: Each pod runs in its own VM-isolated environment for strong isolation.
- **Optimized Resource Utilization**: Define CPU and memory at the pod level for efficient allocation.
- **Seamless AWS Integration**: Natively integrates with IAM, CloudWatch, EFS, and ALB/NLB.

14.5.0: Fargate Profile: Core Configuration

- **Purpose**: A Fargate profile defines which pods in your EKS cluster run on Fargate. It acts as a filter based on namespaces and labels.
- **Configuration**:
  - **Namespaces**: Specifies the namespace(s) where the profile applies (e.g., default).
  - **Label Selectors**: Optionally filters pods within the namespace by labels (e.g., env: prod).
  - **Subnets and IAM**: Defines the private VPC subnets where Fargate pods will run and the IAM role (AmazonEKSFargatePodExecutionRole) for pod permissions.
- **Behavior**: Pods matching the profile (by namespace and labels) are scheduled on Fargate; others require EC2 nodes or another profile.

14.6.0: Limitations of Fargate with EKS

- **No DaemonSets**: Unsupported; use sidecars for logging/monitoring.

- **No GPU Support**: GPU workloads require EC2.
- **Storage**:
  - EFS supported natively; EBS requires static provisioning (EBS is not supported directly on fargate).
  - **StatefulSets**: Supported but not recommended for heavy stateful workloads (e.g., databases) due to EFS's higher latency compared to EBS. Use EFS for lightweight stateful needs or pre-provision EBS with static provisioning.
- **Networking**: Private subnets only, but IP usage is optimized.
- **No Privileged Pods**: Security restrictions apply.
- **Patching**: AWS patches may evict pods, but eviction handling is improved.

14.7.0: When to Use Fargate with EKS?

✅ Best Use Cases:

- **Dynamic Workloads**: Bursty or short-lived tasks (e.g., CI/CD pipelines).
- **Multi-Tenant Clusters**: Isolation for shared environments.
- **Stateless Apps**: Works well with EFS for lightweight stateful needs.
- **Lightweight Stateful Apps**: StatefulSets with EFS for simple stateful needs.
- **Arm Workloads**: Leverage Graviton for cost savings.
- **Hybrid Mode**: Combine Fargate for stateless apps with EC2 node groups for complex workloads.

🚫 Avoid If:

- **GPU-Intensive Workloads**: No GPU support.
- **DaemonSet Requirements**: Logging/monitoring agents need EC2.
- **Heavy Stateful Apps**: Databases requiring low-latency block storage (e.g., EBS) are better on EC2.

14.8.0: CPU and Memory Allocation for Fargate

Ref: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html

Fargate supports specific CPU and memory combinations for pods. If not specified, defaults to 0.25 vCPU and 0.5 GB memory. No need to specify limits as they match requests.

Supported Configurations

vCPU	Memory Values
0.25 vCPU	0.5 GB, 1 GB, 2 GB
0.5 vCPU	1 GB, 2 GB, 3 GB, 4 GB
1 vCPU	2 GB, 3 GB, 4 GB, 5 GB, 6 GB, 7 GB, 8 GB
2 vCPU	4 GB to 16 GB (1-GB increments)
4 vCPU	8 GB to 30 GB (1-GB increments)
8 vCPU	16 GB to 60 GB (4-GB increments)
16 vCPU	32 GB to 120 GB (8-GB increments)

14.9.0: Pricing for Fargate with EKS

- **Fargate**: Per-second billing for vCPU (e.g., $0.04048/vCPU-hour) and memory (e.g., $0.004445/GB-hour) in us-west-2; varies by region.
- **EKS**: $0.10/hour (~$72/month) per cluster.
- **Additional Costs**: Data transfer, EFS, NAT Gateway (for public access).

1) Pricing Examples

- **Case 1**: 20 pods, 1 vCPU, 2 GB memory each, running 10 minutes daily for 30 days:
  - Control Plane: $72/month
  - vCPU Cost: 20 pods × 1 vCPU × ($0.04048/(60×60)) × 600 seconds × 30 days = $4.04

==========

- Memory Cost: 20 pods × 2 GB × ($0.004445/(60×60)) × 600 seconds × 30 days = $0.89
- Total: $72 + $4.04 + $0.89 = $76.93/month

- **Case 2**: 20 pods, 1 vCPU, 2 GB memory each, running 12 hours daily for 30 days:
  - Control Plane: $72/month
  - vCPU Cost: 20 pods × 1 vCPU × ($0.04048/(60×60)) × 43200 seconds × 30 days = $291.45
  - Memory Cost: 20 pods × 2 GB × ($0.004445/(60×60)) × 43200 seconds × 30 days = $64.00
  - Total: $72 + $291.45 + $64.00 = $427.45/month

2) EC2 Comparison

- **EKS with 2 m5.large EC2 Instances**:
  - Control Plane: $72/month
  - Worker Nodes: ($0.096/hour × 24 hours × 30 days) × 2 = $138.24/month
  - Total: $72 + $138.24 = $210.24/month

- **Insight**: Fargate is cheaper for short-lived workloads (Case 1: $76.93 vs. $210.24), but EC2 is more cost-effective for long-running workloads (Case 2: $427.45 vs. $210.24).

14.10.0: Comparison between Fargate Vs EC2 in EKS

Fargate vs. EC2 in EKS: Comparison

Feature	Fargate	EC2-based Worker Nodes
Management	Fully managed, no nodes to maintain	Requires provisioning and maintenance
Scaling	Automatic per-pod scaling	Manual or via Cluster Autoscaler
Cost Model	Per vCPU/memory, pay-as-you-go	Per instance, even if idle
Security	Pod-level VM isolation	Node-level isolation (shared kernel)
Customization	Limited to pod specs	Full OS/runtime control
Max Pod Size	16 vCPU, 120 GB memory	Depends on EC2 instance type
Networking	Private subnets only, optimized IP usage	Public/private subnets, HostPort/HostNetwork
CNI	AWS VPC CNI only	Custom CNI possible

14.11.0: Comparison between Fargate Vs Lambda

Fargate vs. AWS Lambda: Comparison

Feature	Fargate	AWS Lambda
Event Integration	No inherent event integration, use Ingress	Inherent integration with AWS services
Stateful Apps	Not recommended (EFS only)	Not supported
Max Resource Limits	16 vCPU, 120 GB memory per pod	10 GB memory, 6 vCPU (proportional)
Max Runtime	No max runtime	15 minutes
Logging/Monitoring	Requires sidecars, CloudWatch	Mature integration (CloudWatch, X-Ray)

14.12.0: Demo Eks with Fargate

1) Create an EKS Cluster with Fargate (Using Kubernetes 1.30)

We'll use eksctl to create an EKS cluster with Fargate enabled, specifying Kubernetes version 1.30.

```
eksctl create cluster \
  --name my-cluster \
  --region us-west-2 \
  --fargate \
  --version 1.30
```

==========

Explanation:

- --name my-cluster: Names the cluster my-cluster.
- --region us-west-2: Deploys the cluster in the us-west-2 region.
- --fargate: Enables Fargate and creates a default Fargate profile for default and kube-system namespaces.
- --version 1.30: Uses the latest Kubernetes version supported by EKS as of March 2025.

Output: This command takes 15–20 minutes to complete. It sets up the EKS cluster, configures your kubectl context, and creates a default Fargate profile.

Verify:

```
kubectl get nodes
```

```
vishal@vishalk17:~/Downloads/vish/eks/fargate$ kubectl get nodes
NAME                                                STATUS   ROLES    AGE   VERSION
fargate-ip-192-168-103-31.us-west-2.compute.internal   Ready    <none>   12m   v1.30.8-eks-2d5f260
fargate-ip-192-168-115-10.us-west-2.compute.internal   Ready    <none>   12m   v1.30.8-eks-2d5f260
vishal@vishalk17:~/Downloads/vish/eks/fargate$ kubectl get pods
No resources found in default namespace.
vishal@vishalk17:~/Downloads/vish/eks/fargate$ kubectl get pods  -A
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE
kube-system   coredns-5f8c4b75f7-fsnwb   1/1     Running   0          14m
kube-system   coredns-5f8c4b75f7-xmt6g   1/1     Running   0          14m
vishal@vishalk17:~/Downloads/vish/eks/fargate$ kubectl get pods  -A -o wide
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE   IP               NODE
kube-system   coredns-5f8c4b75f7-fsnwb   1/1     Running   0          14m   192.168.115.10   fargate-ip-192-168-115-10.us-west-2.compute.internal
kube-system   coredns-5f8c4b75f7-xmt6g   1/1     Running   0          14m   192.168.103.31   fargate-ip-192-168-103-31.us-west-2.compute.internal
```

- Fargate provisions a lightweight, isolated micro-VM for each pod, providing dedicated CPU, memory, and networking resources (e.g., an ENI).
- These micro-VMs are managed by AWS and are not visible as traditional Kubernetes nodes. However, the Kubernetes API represents each Fargate pod as a **virtual node** with a name like fargate-ip-<IP>.
- When you run kubectl get nodes, you'll see one virtual node per Fargate pod. For example, if you have two pods in a namespace covered by a Fargate profile, you'll see two nodes.

2. Create a New Namespace web-prod

We'll create a new namespace called web-prod for our Nginx deployment.

```
kubectl create namespace web-prod
```

3. Create a Fargate Profile for the web-prod Namespace (Using CLI, No Subnets Specified)

Instead of using a YAML file and manually specifying subnets, we'll use the eksctl create fargateprofile command with the --namespace flag to target the web-prod namespace.

By not specifying subnets, eksctl will automatically select the private subnets associated with your EKS cluster's VPC.

```
eksctl create fargateprofile \
  --cluster my-cluster \
  --region us-west-2 \
  --name fp-web-prod \
  --namespace web-prod
```

Explanation:

- --cluster my-cluster: Specifies the EKS cluster.
- --region us-west-2: Specifies the region.
- --name fp-web-prod: Names the Fargate profile.
- --namespace web-prod: Targets the web-prod namespace.
- **Subnets**: Since we didn't specify subnets, eksctl automatically uses the private subnets in the cluster's VPC. This is convenient and reduces manual configuration.

Output: This command takes a few minutes and creates the Fargate profile for the web-prod namespace

```
vishal@vishalk17:~/Downloads/vish/eks/fargate$ eksctl create fargateprofile \
  --cluster my-cluster \
  --region us-west-2 \
  --name fp-web-prod \
  --namespace web-prod
2025-03-23 21:15:38 [ℹ]  creating Fargate profile "fp-web-prod" on EKS cluster "my-cluster"
2025-03-23 21:15:57 [ℹ]  created Fargate profile "fp-web-prod" on EKS cluster "my-cluster"
vishal@vishalk17:~/Downloads/vish/eks/fargate$
```

Verify:

```
eksctl get fargateprofile --cluster my-cluster --region us-west-2
```

You should see fp-web-prod in the list, along with the default profile.

```
vishal@vishalk17:~/Downloads/vish/eks/fargate$ eksctl get fargateprofile --cluster my-cluster --
NAME            SELECTOR_NAMESPACE      SELECTOR_LABELS POD_EXECUTION_ROLE_ARN
fp-default      default                 <none>          arn:aws:iam::533267266374:role/eksctl-my
fp-default      kube-system             <none>          arn:aws:iam::533267266374:role/eksctl-my
fp-web-prod     web-prod                <none>          arn:aws:iam::533267266374:role/eksctl-my
vishal@vishalk17:~/Downloads/vish/eks/fargate$
```

Alternative: Create a Fargate Profile via the AWS Management Console (Web)

4. Deploy Nginx with a LoadBalancer in the web-prod Namespace

We'll deploy an Nginx application in the web-prod namespace and expose it using a Service of type LoadBalancer, which will provision an NLB.

- **Create the Nginx Deployment**: Create a file named nginx-deployment.yaml with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 namespace: web-prod
spec:
 replicas: 2
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - name: nginx
         image: nginx:latest
         ports:
           - containerPort: 80
         resources:
           requests:
             cpu: "0.5"
             memory: "512Mi"
---
apiVersion: v1
kind: Service
metadata:
 name: nginx-service
 namespace: web-prod
 annotations:
   service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
```

```
   service.beta.kubernetes.io/aws-load-balancer-security-groups: "sg-0d388e5b8da5eb114"  # Replace with the ID
of nlb-sg-my-cluster
spec:
 type: LoadBalancer
 ports:
   - port: 80
     targetPort: 80
     protocol: TCP
 selector:
   app: nginx
```

Deployment :

namespace: web-prod: Deploys the Nginx pods in the web-prod namespace.
replicas: 2: Runs 2 Nginx pods for redundancy.
app: nginx: Labels the pods for the service to target.
containerPort: 80: Exposes port 80 on the Nginx container.
resources: Requests 0.5 vCPU and 512 MiB of memory, which fits within Fargate's supported configurations.

Service :

namespace: web-prod: Places the service in the web-prod namespace.
type: LoadBalancer: Provisions an NLB to expose the service externally.
annotations: Specifies that an NLB should be created.
ports: Maps port 80 on the NLB to port 80 on the Nginx pods.
selector: app: nginx: Targets the Nginx pods created by the deployment.

Deploy the Nginx Application and Service:

```
kubectl apply -f nginx-deployment.yaml
```

==========

5. Verify the Deployment and Access the Application

```
kubectl get pods -A -o wide
kubectl get nodes
kubectl get svc -n web-prod
```

The EXTERNAL-IP is the NLB's DNS name. It may take a few minutes for the NLB to be fully provisioned and the DNS name to resolve.

```
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get nodes
NAME                                              STATUS   ROLES    AGE   VERSION
fargate-ip-192-168-102-24.us-west-2.compute.internal   Ready    <none>   22s   v1.30.8-eks-2d5f260
fargate-ip-192-168-103-31.us-west-2.compute.internal   Ready    <none>   37m   v1.30.8-eks-2d5f260
fargate-ip-192-168-115-10.us-west-2.compute.internal   Ready    <none>   37m   v1.30.8-eks-2d5f260
fargate-ip-192-168-164-44.us-west-2.compute.internal   Ready    <none>   37s   v1.30.8-eks-2d5f260
vishal@vishalk17:~/Downloads/vish/eks$
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get pods -A -o wide
NAMESPACE     NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
S
kube-system   coredns-5f8c4b75f7-fsnwb            1/1     Running   0          38m   192.168.115.10   fargate-ip-192-168-115-10.u
kube-system   coredns-5f8c4b75f7-xmt6g            1/1     Running   0          38m   192.168.103.31   fargate-ip-192-168-103-31.u
web-prod      nginx-deployment-7859f5875b-nrkxw   1/1     Running   0          90s   192.168.164.44   fargate-ip-192-168-164-44.u
web-prod      nginx-deployment-7859f5875b-spdsf   1/1     Running   0          90s   192.168.102.24   fargate-ip-192-168-102-24.u
vishal@vishalk17:~/Downloads/vish/eks$ kubectl get svc -n web-prod
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP                                                                  POR
nginx-service   LoadBalancer   10.100.57.79   abb44b0002e2d4b9ea0be2d07fb2238e-419a5f5d4cb37e70.elb.us-west-2.amazonaws.com   80:
```

Delete Eks cluster :

```
eksctl delete cluster --name my-cluster --region us-west-2
```