ACIT 3495

Technical Report:

Video Streaming System with Kubernetes

By Garen Pham

Submitted to Motasem Aldiab

Introduction:

Kubernetes has revolutionized how modern applications are deployed and scaled. It provides an efficient way to manage containerized applications, including video streaming systems. Kubernetes simplifies the deployment and scaling of complex systems by abstracting away much of the underlying infrastructure, enabling developers and operations teams to focus on the application's logic.

This technical report will focus on deploying a video streaming system using Kubernetes. We will cover the essential components required for a video streaming system, including media servers, databases, load balancers, and frontend web servers, and how to deploy and manage them using Kubernetes. We will explore Kubernetes concepts such as Pods, Deployments, Services, and ConfigMaps.

Deploying MySQL database with Kubernetes

The YAML file defines four Kubernetes API resources: Service, PersistentVolume, PersistentVolumeClaim, and Deployment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  type: ClusterIP
  ports:
    - port: 3306
      targetPort: 3306
  selector:
    app: db
```

Service:

The Service API resource exposes the MySQL database to other Kubernetes pods by creating a stable IP address and DNS name. The service is defined with the name “db” and has a type of ClusterIP, which means that the service is only accessible internally within the Kubernetes cluster. The port is set to 3306, which is the default port for MySQL databases. The selector defines the Kubernetes pod(s) that the service will route traffic to, which in this case is a pod with the label “app: db”.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv
spec:
  capacity:
    storage: 10Mi
  persistentVolumeReclaimPolicy: Delete
  storageClassName: db-storagepv
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data/mysqldata/
```

PersistentVolume:

The PersistentVolume API resource is used to store data for the MySQL database. A PersistentVolume represents a piece of networked storage that has been provisioned by an administrator. In this case, the volume is hosted on the node's file system at "/mnt/data/mysqldata/". The capacity is set to 10Mi, which is a small amount of storage for testing purposes. The PersistentVolumeReclaimPolicy is set to Delete, which means that when the PersistentVolumeClaim is deleted, the storage will be deleted as well. The storageClassName is set to "db-storagepv", which is used to identify the storage class that the PersistentVolumeClaim should use. AccessModes is set to ReadWriteOnce, which means that the volume can be mounted as read-write by a single node.

PersistentVolumeClaim:

The PersistentVolumeClaim API resource is used to request storage resources from a storage class in Kubernetes. The name of the claim is set to "db-pvc". The access mode is set to ReadWriteOnce, which means that the volume can be mounted as read-write by a single node. The resources are set to request 10Mi of storage from the "db-storagepv" storage class.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Mi
  storageClassName: db-storagepv
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-deploy
  labels:
    app: db
spec:
  selector:
    matchLabels:
      app: db
  minReadySeconds: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  template:
    metadata:
      labels:
        app: db
    spec:
      terminationGracePeriodSeconds: 1
      volumes:
        - name: db-storage
          persistentVolumeClaim:
            claimName: db-pvc
        - name: db-env
          configMap:
            name: db-env
      containers:
        - name: db
          image: mysql:5.7
          envFrom:
            - configMapRef:
                name: db-env
          volumeMounts:
            - mountPath: '/var/lib/mysql'
              name: db-storage
          ports:
            - containerPort: 3306
```

Deployment:

The Deployment API resource is used to manage the Kubernetes pods that run the MySQL database. The name of the deployment is set to “db-deploy”, and the label is set to “app: db”. The selector is set to match the label “app: db”. The minReadySeconds is set to 4, which means that the pod will be considered ready after 4 seconds. The strategy is set to RollingUpdate, which means that pods will be updated one at a time with zero downtime. The template defines the pod that will be created by the deployment. The terminationGracePeriodSeconds is set to 1, which means that when the pod is deleted, it will have 1 second to terminate gracefully. The volumes are defined with the name “db-storage”. The “db-storage” volume is mounted to the PersistentVolumeClaim named “db-pvc”. The “db-env” volume is mounted to the ConfigMap named “db-env”, which will be shown below. The container is named “db” and uses the MySQL 5.7 image. The environment variables are set using the ConfigMap. The “db-storage” volume is mounted to the “/var/lib/mysql” path, and the container port is set to 3306.

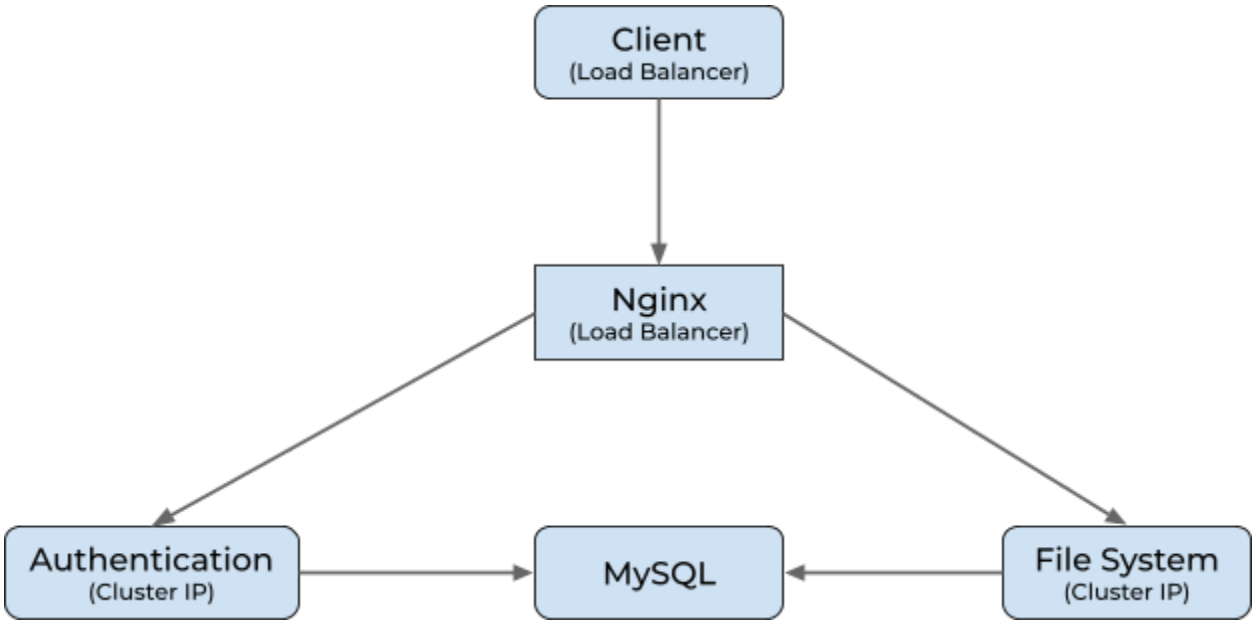
ConfigMap:

The ConfigMap API resource is used to store configuration data in Kubernetes. In this case, the configuration data is used for the MySQL database. The name of the ConfigMap is set to “db-env”, and the data is set to include the MySQL database name, root password, user, and password.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-env
data:
  MYSQL_DATABASE: project1
  MYSQL_PASSWORD: project1
  MYSQL_ROOT_PASSWORD: project1
  MYSQL_USER: project1
```

The MySQL config covers most of the topic information compared with other services. They have similar settings as the database. Before going further, we must explore the overall structure of the application first.

Overall Architecture:



I am implementing Next.js framework for my client service. Next.js front-end normally cannot communicate with other backend services inside the Kubernetes pods. I have included Nginx as the proxy server for establishing connections to internal back-end services.

Nginx web server:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: LoadBalancer
  ports:
    - port: 4010
      targetPort: 80
  selector:
    app: nginx
```

Load Balancer:

The Deployment is associated with a Kubernetes Service named "nginx", which acts as a Load Balancer to distribute the incoming traffic among the available Pods. The Service is defined as a LoadBalancer type, which provisions an external DNS name by the cloud provider.

The Service listens on port 4010 and forwards the incoming traffic to the targetPort 80 of the Pods. The Load Balancer selects the Pods based on the label selector "app: nginx", which then matches the label assigned to the Pods by the Deployment.

Pods:

The Deployment is defined with the name "nginx-deploy" and labels "app: nginx". The Deployment ensures that the specified number of Pods with the label "app: nginx" is always running and replaces any failed or unresponsive Pods.

The Pods are defined with a template that includes metadata labels and a container specification. The container runs an Nginx web server using the Docker image "phamminhtan2k2/video-streaming-nginx:v1". The container listens on port 80 to serve incoming traffic.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 1
      containers:
        - name: nginx
          image: phamminhtan2k2/video-streaming-nginx:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 80
```

Client front-end:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: client
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 3000
  selector:
    app: client
```

Service:

The client's service configuration has the same characteristic as the Nginx web server. When deployed, it will receive a DNS name provided by the cloud provider, which directs the user from port 80 to the container on port 3000.

Config map:

A config map is used to store the Nginx external DNS name that is generated as the Load Balancer. The key is "NEXT_PUBLIC_BASE_URL"

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: base-url
data:
  NEXT_PUBLIC_BASE_URL: http://<nginxDNS>:4010
```

```yaml
volumes:
  - name: videos-storage
    persistentVolumeClaim:
      claimName: shared-pvc
  - name: base-url
    configMap:
      name: base-url
containers:
  - name: client
    image: phamminhtan2k2/video-streaming-client:v1
    imagePullPolicy: Always
    envFrom:
      - configMapRef:
          name: base-url
    ports:
      - containerPort: 3000
    volumeMounts:
      - mountPath: '/app/public/videos'
        name: videos-storage
```

Deployment:

It has a volume named "videos-storage" that mounted at "/app/public/videos" inside the deployed container. It claims from the "shared-pvc" persistent volume container, which is for storing the user-uploaded videos with the File System back-end.

“Shared-storage.yaml” configuration (PV and PVC):

This configuration of Persistent Volume and Persistent Volume claim is used to store the user-uploaded videos that can be accessed from the Client and File Systemt containers.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: shared-pv
spec:
  capacity:
    storage: 40Mi
  persistentVolumeReclaimPolicy: Delete
  storageClassName: shared-storage
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/shared
```

The Persistent Volume (PV) is defined using the v1 API version and kind. The metadata section defines the name of the PV as "shared-pv". The spec section defines the capacity of the PV, which is set to 40Mi. The persistentVolumeReclaimPolicy is set to Delete, which means that the PV will be deleted when it is no longer needed. The storageClassName is set to "shared-storage", which is used to match the PV and PVC. The accessModes section defines the access modes that are available for the PV. In this case, the access mode is set to ReadWriteOnce, which means that the PV can be mounted as read-write by a single node at a time. Finally, the hostPath section specifies the path on the host machine where the PV is stored. In this case, the path is set to "/data/shared".

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 40Mi
  storageClassName: shared-storage
```

The resources section defines the storage requested by the Persistent Volume Claim (PVC), which is set to 40Mi to match the capacity of the PV. The storageClassName is set to "shared-storage" to match the PV. When a PVC is created, Kubernetes will dynamically provision a PV with the matching storageClassName if one is not already available.

Configmaps.yaml:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cors
data:
  LOCAL_URL: http://<clientDNS>
```

This Config Map is used for storing Client's external DNS name provided by the cloud provider. It then can be accessed from the Authentication and the File System as an environment variable. Those two services need it for setting the CORS header 'Access-Control-Allow-Origin'.

```
app.use(
  cors({
    origin: [
      process.env.LOCAL_URL,
      'http://localhost:4000',
      'http://localhost:4004',
      'http://localhost:3000',
    ],
    methods: ['GET', 'POST'],
    credentials: true,
  }),
);
```

Authentication deployment:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: authentication
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 3004
  selector:
    app: authentication
```

Service:

The Kubernetes Service is named "authentication" and is of type ClusterIP. The service is created as a ClusterIP type, which means that it is only accessible within the Kubernetes cluster. The service listens on port 80 and forwards traffic to the authentication pods running on port 3004. The selector for the Service is set to "app: authentication". This means that the Service will route traffic to all pods with the label "app: authentication".

```yaml
volumes:
  - name: cors
    configMap:
      name: cors
containers:
  - name: authentication
    image: phamminhtan2k2/video-streaming-authentication:v1
    imagePullPolicy: Always
    envFrom:
      - configMapRef:
          name: cors
    ports:
      - containerPort: 3004
```

Deployment:

The deployment configuration is similar to the client service. The authentication pods are created from a Docker image (phamminhtan2k2/video-streaming-authentication:v1) and the variable for setting the origin for the CORS header 'Access-Control-Allow-Origin' are passed to the container using a configMap. In this case, the configMap is named "cors" and contains CORS-related configuration options.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: authentication-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: authentication-deploy
  minReplicas: 1
  maxReplicas: 4
  targetCPUUtilizationPercentage: 10
```

Horizontal Pod Autoscaler:

Lastly, an HPA (HorizontalPodAutoscaler) is defined to scale the number of replicas based on CPU utilization. The HPA scales the number of replicas between 1 and 4, depending on the current CPU utilization. The targetCPUUtilizationPercentage is set to 10%, which means that the HPA will try to keep the average CPU utilization across all replicas at or below 10%.

File System back-end server:

Service:

The service, named filesystem, is of type ClusterIP and exposes port 80. It routes incoming traffic to the pod running the video streaming file system container through port 3010. The selector field specifies that the service should route traffic to pods with the label app set to the filesystem.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: filesystem
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 3010
  selector:
    app: filesystem
```

```yaml
spec:
  terminationGracePeriodSeconds: 1
  volumes:
    - name: videos-storage
      persistentVolumeClaim:
        claimName: shared-pvc
    - name: cors
      configMap:
        name: cors
  containers:
    - name: filesystem
      image: phamminhtan2k2/video-streaming-file:v1
      imagePullPolicy: Always
      envFrom:
        - configMapRef:
            name: cors
      ports:
        - containerPort: 3010
      volumeMounts:
        - mountPath: '/app/public/videos'
          name: videos-storage
```

Deployment:

The deployment is similar to the Authentication service. The container also has an envFrom field that references the cors configMap. The volumeMounts field maps the videos-storage volume to the container's '/app/public/videos' directory for sharing with the Client Service.

Horizontal Pod Autoscaler:

Minimally it is running from 2 pods, and can get up to 10 pods running. The HPA will try to keep the average CPU utilization across all replicas at or below 14%

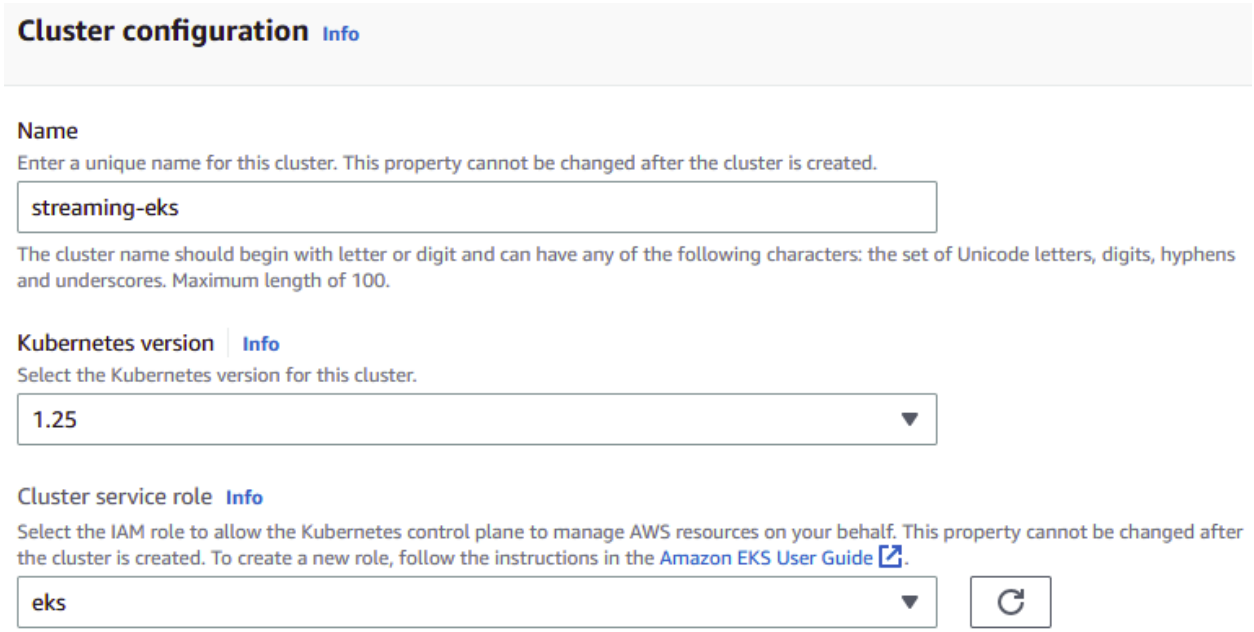
```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: filesystem-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: filesystem-deploy
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 14
```

Cloud Deployment process using Amazon EKS:

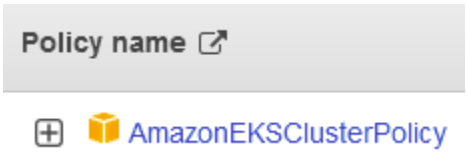
Amazon Elastic Kubernetes Service (EKS) is a fully managed container orchestration service provided by Amazon Web Services (AWS). It is based on the Kubernetes open-source platform, and it allows users to easily deploy, manage, and scale containerized applications using the same APIs, tools, and security controls they use with their on-premises Kubernetes deployments.

EKS simplifies the process of building and operating Kubernetes clusters, removing the need for users to manually manage the underlying infrastructure, including servers, load balancers, and storage. Instead, EKS takes care of these tasks, providing a scalable, highly available, and secure environment for running containerized applications.

Creating an EKS Cluster:



For creating a cluster, we just need to set the Cluster service role. Other settings are left to default values. Go to the IAM Management Console and create a role with **AmazonEKSClusterPolicy**.

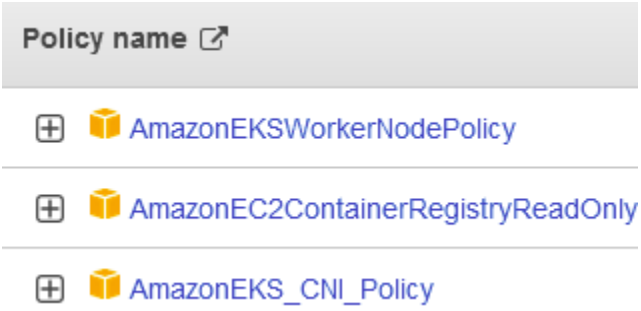


Node Group:

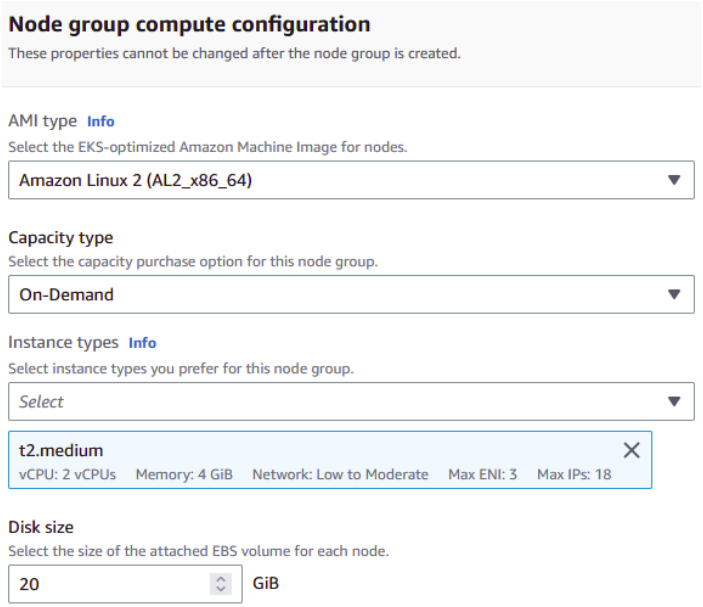
In Amazon Elastic Kubernetes Service (EKS), a node group is a set of Amazon Elastic Compute Cloud (EC2) instances that are deployed to run Kubernetes pods. Node groups are managed by EKS and are created within an Amazon EC2 Auto Scaling group, which automatically adjusts the number of instances in response to changes in demand for resources.

Node groups provide a way to scale the number of worker nodes in a Kubernetes cluster on-demand and can be used to provide additional resources as needed for different applications or workloads. By using node groups, you can more easily manage and scale your Kubernetes clusters running on EKS.

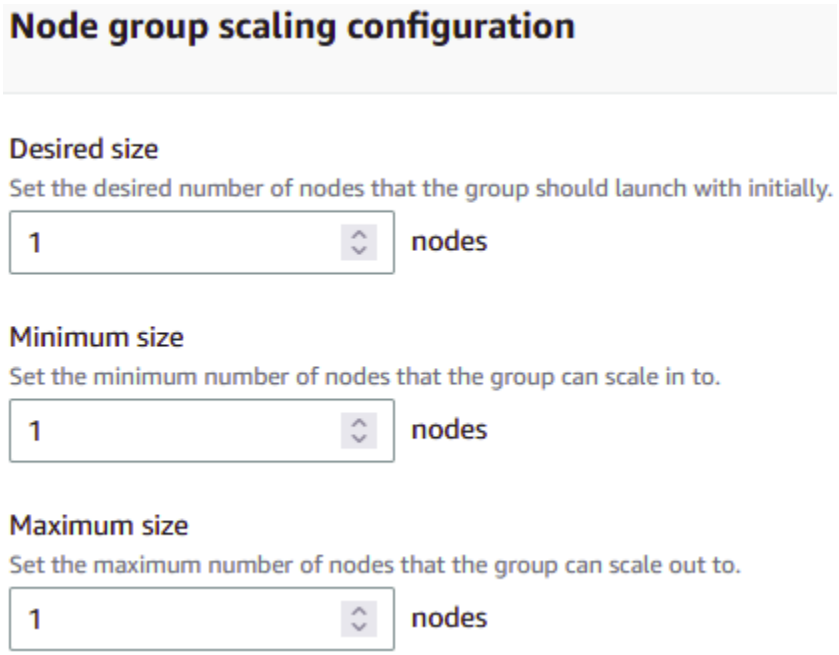
Firstly we need to create a role that contains the necessary policies for the Node Group, which are **AmazonEKSWorkerNodePolicy**, **AmazonEC2ContainerRegistryReadOnly**, and **AmazonEKS_CNI_Policy**.



Next, select a Node group to compute the configuration. For this project, I chose t2.medium EC2 instance.



For the Node group scaling configuration, I want to keep the size for only 1 node to minimize cost. It is possible to set more nodes for the application scaling purpose.



Those are the key things for setting up a Node Group for this project. Others can be left at default values.

Connect to the EKS Cluster using AWS CLI:

Firstly, connect to the Cluster using the command: aws eks update-kubeconfig --region <region-code> --name <my-cluster>

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> aws eks update-kubeconfig --region us-west-2 --name streaming-eks
Updated context arn:aws:eks:us-west-2:828590063271:cluster/streaming-eks in C:\Users\Admin\.kube\config
```

By running kubectl get no, we can see the node that created from the Node Group is running:

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> kubectl get no
NAME                                           STATUS   ROLES    AGE     VERSION
ip-172-31-4-164.us-west-2.compute.internal     Ready    <none>   4m50s   v1.25.6-eks-48e63af
```

Deploying the services to EKS:

From the deployment folder, I used the command kubectl apply -f db.yaml -f shared-storage.yaml -f nginx.yaml.

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> kubectl apply -f db.yaml -f shared-storage.yaml -f nginx.yaml
service/db created
persistentvolume/db-pv created
persistentvolumeclaim/db-pvc created
configmap/db-env created
deployment.apps/db-deploy created
persistentvolume/shared-pv created
persistentvolumeclaim/shared-pvc created
service/nginx created
deployment.apps/nginx-deploy created
```

Next, I used kubectl get svc to get the DNS name of the Nginx service to copy to the Config Map in the Client configuration:

```
PS C:\Users\Admin> kubectl get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)          AGE
db           ClusterIP      10.100.238.222   <none>                                                                   3306/TCP         54s
kubernetes   ClusterIP      10.100.0.1       <none>                                                                   443/TCP          26m
nginx        LoadBalancer   10.100.222.73    a4c202d19512342edb3bf54eb9b6161d-1674851257.us-west-2.elb.amazonaws.com   4010:30106/TCP   53s
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: base-url
data:
  NEXT_PUBLIC_BASE_URL: http://a4c202d19512342edb3bf54eb9b6161d-1674851257.us-west-2.elb.amazonaws.com:4010
```

Then, I launched the Client with kubectl apply -f client.yaml.

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> kubectl apply -f client.yaml
service/client created
configmap/base-url created
deployment.apps/client-deploy created
```

I ran the command kubectl get svc again to get the Client DNS name, and then paste it in the configmaps.yaml. The Authentication and File System service now can use this value for setting up the CORS header 'Access-Control-Allow-Origin'.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cors
data:
  LOCAL_URL: http://abac961ac56da4846a82dbacda516d81-1851637641.us-west-2.elb.amazonaws.com
```

I started the Authentication service first for it to make a connection to MySQL database server: kubectl apply -f configmaps.yaml -f authentication.yaml

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> kubectl apply -f configmaps.yaml -f authentication.yaml
configmap/cors created
service/authentication created
deployment.apps/authentication-deploy created
horizontalpodautoscaler.autoscaling/authentication-hpa created
```

Finally, I started the File System using kubectl apply -f file-system.yaml.

```
PS C:\Amplify\CIT\3495\videostreaming\deployment> kubectl apply -f file-system.yaml
service/filesystem created
deployment.apps/filesystem-deploy created
horizontalpodautoscaler.autoscaling/filesystem-hpa created
```

By running kubectl get pods, we can see that the application is fully running without any errors:

```
PS C:\Users\Admin> kubectl get pods
NAME                                 READY   STATUS    RESTARTS        AGE
authentication-deploy-6db67d79ff-ppbjs   1/1     Running   0               3m40s
client-deploy-5c65c78fdb-shmnd       1/1     Running   0               9m49s
db-deploy-7f5685fc5-xss65            1/1     Running   0               12m
filesystem-deploy-597799c7d6-bwcn7   1/1     Running   0               2m11s
filesystem-deploy-597799c7d6-sz2tn   1/1     Running   0               116s
nginx-deploy-5f4888ccb9-m55v2        1/1     Running   7 (7m13s ago)   12m
```

Summary:

The Kubernetes video streaming services deployed on the EKS project involve deploying a video streaming application on an EKS cluster using various Kubernetes resources to ensure high availability, scalability, and fault tolerance.

The project utilizes Load Balancer and Cluster IP services to expose the backend server and frontend client application to the outside world. Config Maps are used to define environment variables for the pods, including the base URL of the backend server and the local URL of the frontend application. This enables easy configuration of the application while decoupling the configuration from the application code.

PV (Persistent Volume) and PVC (Persistent Volume Claim) are used to manage the storage required by the video streaming services, enabling video files to be stored persistently across pod restarts. This ensures that video files are not lost when pods are terminated or restarted.

Additionally, the project uses HPA (Horizontal Pod Autoscaler) to automatically scale the number of pods based on the incoming traffic. This ensures that the application can handle increased traffic during peak usage periods without downtime.

Overall, this project demonstrates the effective use of various Kubernetes resources to deploy and manage a scalable and highly available video streaming application on EKS, ensuring a seamless experience for users.