Kubernetes RBAC (Role-Based Access Control) – Complete Guide

1. What is RBAC?

RBAC (Role-Based Access Control) is a security mechanism in Kubernetes used to control **who can do what** inside a cluster.
It ensures users and services only have the minimum required permissions.

RBAC works with:

- **Roles & RoleBindings** → Namespace-level permissions
- **ClusterRoles & ClusterRoleBindings** → Cluster-wide permissions

🔑 **Key Concepts**

- **Authentication** → *Who you are* (IAM user, service account, etc.)
- **Authorization** → *What you can do* (permissions on resources)

---

2. Nouns and Verbs

RBAC rules are written using **nouns (resources)** and **verbs (actions)**.

- **Nouns (Resources)**
  - AWS: EC2, VPC, Route53
  - Kubernetes: Pod, Service, Deployment, PVC
- **Verbs (Actions)**
  - AWS: CreateInstance, GetInstance, DeleteInstance
  - Kubernetes: create, get, list, watch, update, delete

---

3. Example Roles in a Team

Role	Permissions
Trainee	Read-only (list, get, watch) in roboshop namespace
Junior	Can create Pods
Senior	Can create & update resources
Team Lead	Can delete resources

📌 All YAML files for these roles can be found in your Git repo (e.g., **k8-rbac**).

---

4. RBAC Building Blocks:

1. **Role** → Defines permissions within a **namespace**
2. **RoleBinding** → Assigns a Role to a user/group
3. **ClusterRole** → Defines permissions across the **entire cluster**
4. **ClusterRoleBinding** → Assigns a ClusterRole to a user/group

👉 **Role = namespace level**

👉 **ClusterRole = cluster level**

5. AWS IAM + Kubernetes RBAC Integration (EKS):

In **Amazon EKS**, AWS IAM users/roles must be mapped to Kubernetes identities using the **aws-auth ConfigMap**.

Steps:

1. **Create IAM User** (e.g., Suresh)
2. **Attach IAM Policy** → Example: eks:DescribeCluster
3. **Update aws-auth ConfigMap** in kube-system namespace to map the IAM user → Kubernetes user → Groups

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: arn:aws:iam::069233348386:role/eksctl-roboshop-nodegroup-NodeInstanceRole
      username: system:node:{{EC2PrivateDNSName}}
      groups:
      - system:bootstrappers
      - system:nodes
  mapUsers: |
    - userarn: arn:aws:iam::069233348386:user/Suresh
      username: Suresh
      groups:
      - roboshop-trainee-role
      - roboshop-junior-role
      - roboshop-lead-role
      - roboshop-cluster-role
```

✅ **Multiple groups** = combined permissions.

---

6. Example Role and RoleBinding:

Role (Trainee – Read-Only Pods in roboshop namespace)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: roboshop
  name: roboshop-trainee
rules:
- apiGroups: [""]
  resources: ["pods"]
```

```yaml
  verbs: ["get", "watch", "list"]
```

RoleBinding (Bind Suresh to Trainee Role)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: suresh
  namespace: roboshop
subjects:
- kind: User
  name: Suresh
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: roboshop-trainee
  apiGroup: rbac.authorization.k8s.io
```

👉 Repeat the same pattern for **Junior, Senior, and Lead roles** with their respective rules. You can find all the YAML files in this repository: https://gitlab.com/DiviPavanKumar/k8-rbac

---

7. Testing RBAC Access:

(A) Cluster Admin Setup

1. Create Namespace:
   `$kubectl create namespace roboshop`
2. Apply in order:
   - aws-auth ConfigMap
   - Roles
   - RoleBindings

(B) User Side (Suresh)

1. Login with IAM credentials
2. aws sts get-caller-identity

(Confirms AWS identity)

3. Update kubeconfig
   `$aws eks update-kubeconfig --region us-east-1 --name roboshop`
4. Test access:
   - Default namespace (❌ Forbidden):
     `$kubectl get pods`
     Error from server (Forbidden): User "Suresh" cannot list resource "pods" in namespace "default"
   - Roboshop namespace (✅ Allowed):
     `$kubectl get pods -n roboshop`
     No resources found in roboshop namespace.

8. Role vs ClusterRole in Kubernetes:

- **Role → Namespace-level access**
  Example: Can read Pods, PVCs inside a namespace
- **ClusterRole → Cluster-wide access**
  Example: Can read PersistentVolumes (PVs), Nodes, StorageClasses

👉 Why important?

- **PVC** = namespace resource → needs Role
- **PV** = cluster resource → needs ClusterRole

Example: ClusterRole for PV Access:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: roboshop-cluster
rules:
- apiGroups: [""]
  resources: ["persistentvolumes"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: roboshop-cluster
subjects:
- kind: User
  name: Suresh
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: roboshop-cluster
  apiGroup: rbac.authorization.k8s.io
```

✅ This lets **Suresh** read PVs across the cluster.

---

9. Final Summary:

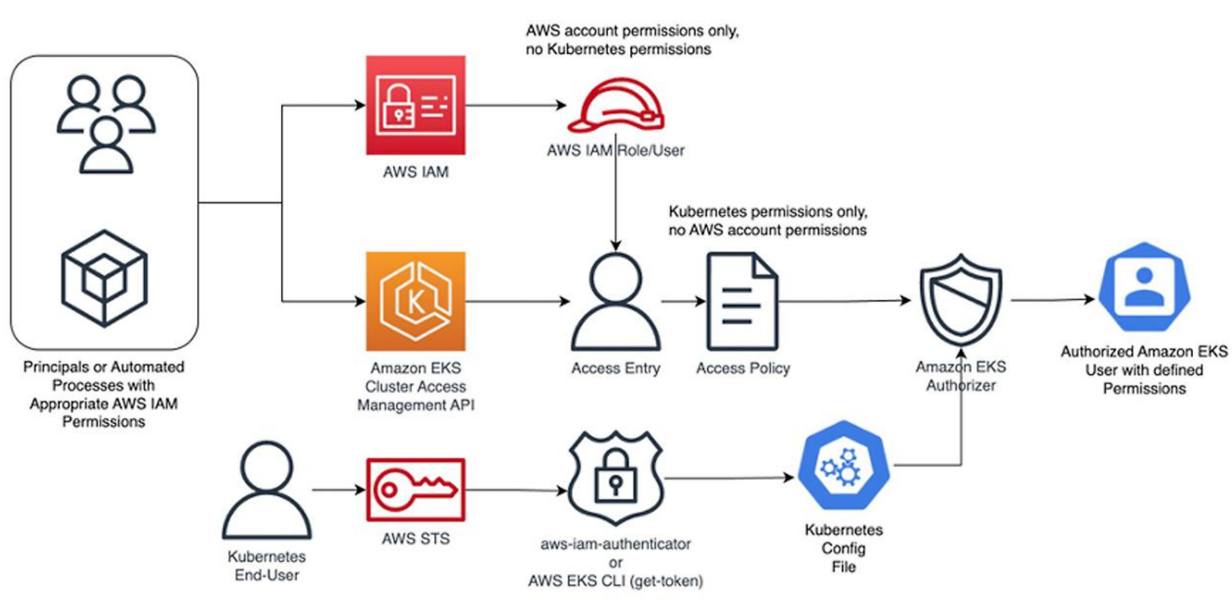
- **Role + RoleBinding** → Namespace-level access (Pods, PVCs, Services)
- **ClusterRole + ClusterRoleBinding** → Cluster-wide access (PVs, Nodes, StorageClasses)

In this example:

- roboshop-lead Role → Full namespace access
- roboshop-cluster ClusterRole → PV access

👉 Together, Suresh can now access **both namespace-level** and **cluster-level** resources.

✅ **Authentication** works → IAM user → mapped via aws-auth
✅ **Authorization** works → RBAC enforces namespace/cluster access
✅ **Result** → Secure, least-privilege access in EKS



Kubernetes RBAC – ServiceAccounts & IAM Integration

1. Why ServiceAccounts?

So far, we covered **Users, Roles, RoleBindings, ClusterRoles, and ClusterRoleBindings** — these are mainly for **human users** (e.g., IAM users like Suresh).
But Kubernetes workloads (Pods) also need a way to securely access resources. That's where

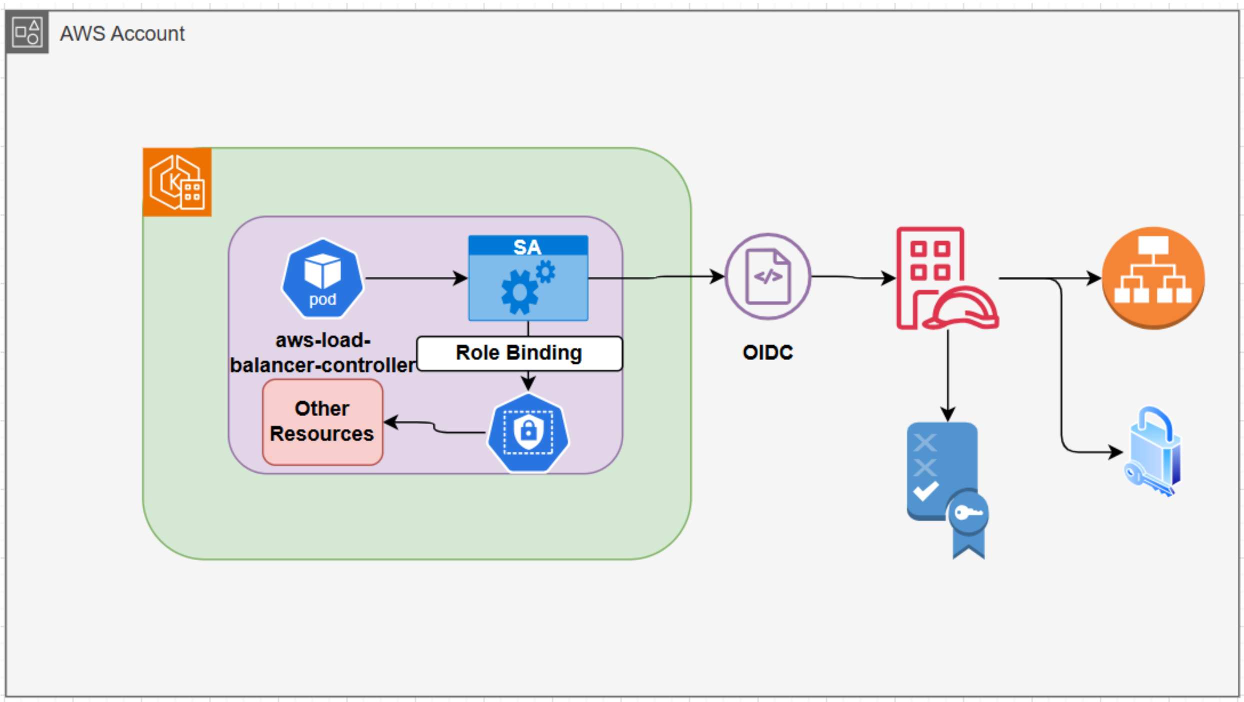
ServiceAccounts (SA) come in.

📌 **ServiceAccount Basics**

- A **ServiceAccount** is a *non-human user* that **Pods** use to authenticate against the Kubernetes API.
- Every **namespace** automatically gets a **default ServiceAccount** when it is created.
- By default, the default ServiceAccount has **no permissions**.
- Pods automatically run using a ServiceAccount (if none is specified → they use default).
- We can check ServiceAccounts with:
  $kubectl get sa -n roboshop
  $kubectl describe sa default -n roboshop
  $kubectl describe pod <pod-name> -n roboshop

- Best practice: **Create custom ServiceAccounts with limited, specific permissions.**
- A ServiceAccount can:
  - Access internal cluster resources (Pods, ConfigMaps, Secrets).
  - Access external resources (AWS S3, AWS Secrets Manager, databases) when linked to IAM Roles.

👉 Example: To allow Pods to read from **AWS Secrets Manager**, we create a ServiceAccount that's mapped to an IAM Role with secretsmanager:GetSecretValue permissions.



---

2. Why ServiceAccount + IAM Integration in EKS?

In **Amazon EKS**, we use **OIDC (OpenID Connect)** to connect Kubernetes ServiceAccounts with IAM Roles.

📌 OIDC Provider

- OIDC allows AWS to trust Kubernetes ServiceAccount identities.
- With OIDC, each ServiceAccount can assume a **dedicated IAM Role**.
- This replaces the old insecure model where **all pods on a node shared the same IAM Role**.

✅ Benefits

- No need to hardcode or mount AWS keys into Pods.
- Each Pod only gets the exact AWS permissions it needs.
- Follows **least privilege** principle.

---

3. Steps: ServiceAccount with IAM Role (Secrets Manager Example)

1️⃣ Enable OIDC Provider

```
REGION_CODE=us-east-1
CLUSTER_NAME=roboshop
```

```
ACC_ID=069233348386

eksctl utils associate-iam-oidc-provider \
  --region $REGION_CODE \
  --cluster $CLUSTER_NAME \
  --approve
```

2️⃣ Create IAM Policy for Secrets Manager

arn:aws:iam::069233348386:policy/RoboshopMySQLSecretReader

This policy should allow:

- secretsmanager:GetSecretValue
- for the ARN of the MySQL password secret.

3️⃣ Create ServiceAccount with IAM Role

```
eksctl create iamserviceaccount \
  --cluster=$CLUSTER_NAME \
  --namespace=roboshop \
  --name=roboshop-mysql-secret-reader \
  --attach-policy-arn=arn:aws:iam::069233348386:policy/RoboshopMySQLSecretReader \
  --override-existing-serviceaccounts \
  --region $REGION_CODE \
  --approve
```

👉 This automatically:

- Creates the IAM Role.
- Creates the ServiceAccount.
- Maps them together.

Check:

```
$kubectl get sa -n roboshop
```

---

4. Kubernetes RBAC for ServiceAccounts:

Even though the ServiceAccount has IAM permissions, inside Kubernetes we still need RBAC to allow it to read cluster resources.

Example: Role & RoleBinding for ServiceAccount

📌 Role: Read-only access

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

```yaml
  name: serviceaccount-role
  namespace: roboshop
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "watch", "list"]
```

📌 RoleBinding: Bind Role to ServiceAccount

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name  serviceaccount-rolebinding
  namespace: roboshop
subjects:
- kind: ServiceAccount
  name: roboshop-mysql-secret-reader
  namespace: roboshop
roleRef:
  kind: Role
  name: serviceaccount-role
```

5. Do We Need aws-auth for ServiceAccounts?

- **ServiceAccounts** are native Kubernetes identities. When bound with Roles/ClusterRoles, Pods using them automatically get those permissions.
- **IAM Users/Roles** (like human user Suresh) require updates in aws-auth ConfigMap to map them to Kubernetes groups.

👉 **Summary**

- **ServiceAccounts** → Just Role/RoleBinding
- **IAM Users/Roles** → Update aws-auth + Role/RoleBinding

---

6. Testing with AWS CLI Pod

Example Pod Using ServiceAccount

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: aws-cli
  namespace: roboshop
spec:
```

```yaml
  serviceAccountName: roboshop-mysql-secret-reader
  containers:
  - name: awscli
    image: amazon/aws-cli
    command: ["sleep", "100000"] # Keep pod running
```

Run & test:

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

```
$kubectl exec -it aws-cli -n roboshop -- bash
```

Inside pod:

```
$aws secretsmanager get-secret-value --secret-id roboshop/mysql/password
```

✅ Output:

```json
{
  "Name": "roboshop/mysql/password",
  "SecretString": "{\"MYSQL_ROOT_PASSWORD\":\"RoboShop@1\"}"
}
```

7. Init Containers for Secrets (Best Practice)

We don’t usually run **aws-cli** in production Pods.
Instead, we use **Init Containers** to fetch secrets and pass them to the application via an **EmptyDir volume**.

📌 **How Init Containers Work**

- Run before the main app container.
- Always finish before the app starts.
- Can share data with the main container using emptyDir.

🐳 Init Containers in Kubernetes

🔹 What are Init Containers?

An **Init Container** is a special type of container in Kubernetes that runs **before the main application containers** inside a Pod start.

Its job is to **prepare the environment** so that the main container can run smoothly.

👉 Think of init containers like a **setup crew** that ensures everything is ready before the main show begins.

Common Use Cases:

- Fetching secrets or config files from external systems
- Waiting for another service (like a database) to become available
- Setting up required dependencies or configuration

🔹 How Init Containers Work

- Init containers always run **to completion** (they do their task and exit).
- They **don't run continuously** like application containers.
- A Pod can have **multiple init containers** – each one must finish successfully before the next one starts.
- The **main container** only starts after all init containers have completed.

🔹 Sharing Data Between Init and Main Containers:

Very often, the init container needs to **pass information** (like a database password or config file) to the main container.
This is usually done using a **shared ephemeral volume**.

Steps:

1. The init container writes data (like a secret) into the shared volume.
2. The main container reads from that shared volume.
3. The main container can then use it as a file or load it into environment variables.

🔹 Ephemeral Volumes

Ephemeral volumes provide **temporary storage** inside a Pod.

- Data lasts **only as long as the Pod exists**.
- If the Pod is deleted, data is gone.

Common Types:

- **emptyDir** → A temporary empty directory shared between containers in the Pod.
- **hostPath** → A path on the node's filesystem.

👉 If you need **persistent data**, use Persistent Volumes (PV/PVC) with proper retention policy.

✅ Example: Using Init Container to Fetch Secrets

Let's look at a real-world example where we:

- Store the MySQL root password in **AWS Secrets Manager**.

- Use an **init container** to fetch the secret.
- Share it with the main MySQL container using an **emptyDir volume**.

Deployment YAML


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: roboshop
  labels:
    component: mysql
    project: roboshop
    tier: database
spec:
  replicas: 1
  selector:
    matchLabels:
      component: mysql
      project: roboshop
      tier: database
  template:
    metadata:
      labels:
        component: mysql
        project: roboshop
        tier: database
    spec:
      serviceAccountName: roboshop-mysql-secret-reader
      volumes:
      - name: mysql-secret
        emptyDir: {}
      containers:
      - name: mysql
        image: pavandivi/mysql:v2
        imagePullPolicy: Always
        volumeMounts:
        - mountPath: /tmp
          name: mysql-secret
      initContainers:
      - name: fetch-secret
        image: amazon/aws-cli
        command:
        - sh
```

```
        - -c
        - |
          aws secretsmanager get-secret-value --secret-id roboshop/mysql/password --query
SecretString --output text | jq -r .MYSQL_ROOT_PASSWORD >
/tmp/mysql_root_password.txt
        volumeMounts:
        - mountPath: /tmp
          name: mysql-secret
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: roboshop
  labels:
    component: mysql
    project: roboshop
    tier: database
spec:
  selector:
    component: mysql
    project: roboshop
    tier: database
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

🔹 **Verification**

After deploying, let's check if the init container correctly fetched the secret:

```
$ kubectl exec -it mysql-6b6df9d6cc-6b9b9 – bash
```

```
Defaulted container "mysql" out of: mysql, fetch-secret (init)
bash-5.1# cd /tmp
bash-5.1# cat mysql_root_password.txt
RoboShop@1
bash-5.1# exit
exit
```

✅ We can see that the init container successfully fetched the secret and stored it in /tmp/mysql_root_password.txt, which is accessible to the main MySQL container.

📝 **Summary:**

- Init containers are **setup containers** that run before the main application starts.

- They are **great for fetching configs, secrets, and waiting on dependencies**.
- Data can be shared between init and main containers using **ephemeral volumes** like emptyDir.
- In production, you often use init containers with **Secrets Manager, ConfigMaps, or dependency checks**.

👉 This ties back nicely to your **ServiceAccount + IAM integration** example, because the init container needed IAM permissions (through the ServiceAccount) to fetch the secret from AWS Secrets Manager.

Entry Script (to Read & Cleanup Secret)

We wrap MySQL's entrypoint in a **custom script** that:

1. Reads password from /tmp/mysql_root_password.txt
2. Exports it as MYSQL_ROOT_PASSWORD
3. Deletes the temp file ✅
4. Starts MySQL normally

mysql-entrypoint.sh:

```
#!/bin/bash
# Check if password file exists
if [ -f /tmp/mysql_root_password.txt ]; then
  PASSWORD=$(cat /tmp/mysql_root_password.txt)
  echo "✅ Accessed Root Password"
else
  echo "❌ Password file not found"
  exit 1
fi
# Export env variable
export MYSQL_ROOT_PASSWORD=$PASSWORD
# Cleanup
rm -rf /tmp/mysql_root_password.txt
# Start MySQL
exec /entrypoint.sh mysqld
```

Make sure Dockerfile sets:

```
COPY mysql-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["mysql-entrypoint.sh"]
```

Verify from Pod

Check inside pod after secret cleanup:

$ kubectl exec -it mysql-0 -n roboshop – bash

<u>Inside container:</u>

```
bash-5.1# cd /tmp
bash-5.1# ls -l
total 0   # ✅ No password file here

bash-5.1# mysql -u root -pRoboShop@1
Welcome to the MySQL monitor.
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| cities             |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

<u>✅ Benefits of This Approach</u>

- **No hardcoded secrets** in YAML/Docker image
- **AWS Secrets Manager** ensures rotation support
- **Temporary storage** → secret is deleted after use
- **Pod security** → even if pod is compromised, secrets are not lingering in /tmp