EKS Deployment Flow

-

STEP 1: Create Flask App

Files:

- app.py
- requirements.txt

Code:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
   return "Hello World from EKS"
```

Command:

```
> python app.py
```

Purpose: Have a working app before deployment.

-

-

STEP 2: Containerize the App

File: Dockerfile

Dockerfile:

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY app.py .
CMD ["python", "app.py"]
```

Commands:

```
> docker build -t flask-hello-world .
> docker run -p 5000:5000 flask-hello-world
```

Purpose: Package app into a container.

-

STEP 3: Push to Amazon ECR

Commands:

```
> aws ecr create-repository --repository-name flask-hello-world
> aws ecr get-login-password | docker login ...
> docker tag flask-hello-world <ecr-uri>
> docker push <ecr-uri>
```

Purpose: Store image on AWS for use in EKS.
yaml

-

⚙️ STEP 4: Create EKS Cluster

File: cluster-config.yaml

YAML:

```
kind: ClusterConfig
metadata:
 name: flask-demo-cluster
 region: us-west-2
nodeGroups:
 - name: worker-nodes
   instanceType: t3.medium
```

Command:

```
> eksctl create cluster -f cluster-config.yaml
```

Purpose: Provision Kubernetes control plane + nodes.

-

-

📀 STEP 5: Add Storage with PVC

Files: storage-class.yaml, pvc.yaml

storage-class.yaml:

```
kind: StorageClass
provisioner: ebs.csi.aws.com
```

pvc.yaml:

```
kind: PersistentVolumeClaim
spec:
 resources:
   requests:
    storage: 1Gi
```

Commands:
> kubectl apply -f storage-class.yaml
> kubectl apply -f pvc.yaml

Purpose: Provide durable disk space to your app.
yaml
-
🚀 STEP 6: Deploy Flask App

File: flask-deployment.yaml

YAML (simplified):
```
kind: Deployment
spec:
  replicas: 2
  containers:
    - name: flask-app
      image: <your-ecr-image>
      volumeMounts:
        - mountPath: /app/data
          name: flask-storage
  volumes:
    - name: flask-storage
      persistentVolumeClaim:
        claimName: flask-storage
```

Command:
> kubectl apply -f flask-deployment.yaml

Purpose: Start app pods in EKS using container image.
yaml
-
🔁 STEP 7: Create Internal Service

File: flask-service.yaml

YAML:
```
kind: Service
spec:
  selector:
    app: flask-app
  ports:
    - port: 80
      targetPort: 5000
  type: ClusterIP
```

Command:

> kubectl apply -f flask-service.yaml

Purpose: Access pods inside the cluster.
yaml
-
🌍 STEP 8: Expose App to Internet

File: flask-ingress.yaml

YAML:
```
kind: Ingress
metadata:
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
spec:
  rules:
    - http:
        paths:
          - path: /
            backend:
              service:
                name: flask-service
                port:
                  number: 80
```

Command:
> kubectl apply -f flask-ingress.yaml
> kubectl get ingress

Purpose: Give public access via AWS ALB.
yaml
-
🔐 STEP 9: Secure Access (RBAC + IAM)

File: rbac.yaml

YAML:
```
kind: Role
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]

kind: RoleBinding
subjects:
  - kind: ServiceAccount
    name: flask-sa
```

roleRef:
  kind: Role
  name: flask-role

Commands:
> kubectl apply -f rbac.yaml

Purpose: Control what your app or users can access.
yaml
-
🛡️ STEP 10: Harden Security in Deployment

File: flask-deployment-secure.yaml

YAML:
```
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
containers:
  - securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
```

Purpose: Prevent privilege abuse, follow security best practices.
markdown
-
🧪 STEP 11: Verify Everything

Commands:
```
> kubectl get nodes
> kubectl get pods
> kubectl logs <pod-name>
> kubectl get svc
> kubectl get ingress
> curl <ALB-URL>
```

Purpose: Ensure everything is running and accessible.

Summary

Step	Purpose	Key File / Command
1	Write app	app.py
2	Containerize	Dockerfile, docker build
3	Push to ECR	docker push, aws ecr
4	Create EKS Cluster	cluster-config.yaml, eksctl create
5	Add Storage	storage-class.yaml, pvc.yaml
6	Deploy app	flask-deployment.yaml

Step	Purpose	Key File / Command
7	Expose internally	flask-service.yaml
8	Expose externally (public)	flask-ingress.yaml
9	Add access control	rbac.yaml, kubectl apply
10	Harden container runtime	securityContext in deployment YAML
11	Test and verify	kubectl, curl, browser

1. How do you implement RBAC (Role-Based Access Control) in EKS?

RBAC = "Who can do what?"

RBAC stands for **Role-Based Access Control**, a system used in **Kubernetes** (and therefore in EKS too) to control **who can access what** resources in the cluster.

For example:

- **Dev users** can only view Pods.
- **Admins** can create or delete resources.
- **CI/CD pipelines** may get special access to certain namespaces.

EKS + RBAC = IAM + Kubernetes RBAC

In **Amazon EKS**, you manage access at **two levels**:

1. **IAM level**: Controls which **users/roles from AWS** can authenticate into the cluster.
2. **Kubernetes RBAC level**: Controls **what those users can do** after logging in.

So even if an IAM user is allowed to log in to the cluster, RBAC decides what they can actually do inside Kubernetes.

How IAM and RBAC Work Together in EKS

In EKS:

Layer	Controls	Example
IAM (AWS)	Who can authenticate to the cluster	IAM user kube_test_user
RBAC (K8s)	What that user can do inside Kubernetes	Role allowing list/get pods

✅ **IAM = Authentication**
✅ **RBAC = Authorization**

Example: Grant kube_test_user read-only access to pods

1️⃣ **IAM User Setup (Assumed Already Exists)**

You already have this IAM user:

```ruby
Name: kube_test_user
ARN: arn:aws:iam::<account-id>:user/kube_test_user
```

✅ Make sure the IAM user has permission to call eks:DescribeCluster and sts:AssumeRole.

2️⃣ Update aws-auth ConfigMap to Allow IAM Access

You must tell EKS:
“This IAM user maps to a Kubernetes user”.
Run:
bash

```
kubectl edit configmap aws-auth -n kube-system
```
Add under mapUsers::
yaml

```
mapUsers:
  - userarn: arn:aws:iam::<account-id>:user/kube_test_user
    username: kube_test_user
    groups:
      - read-only-group
```
This says: "Let kube_test_user authenticate, and map them to the Kubernetes group read-only-group."

3️⃣ Create an RBAC Role for the Group

Create file: rbac-readonly.yaml
yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: readonly-pods
  namespace: demo
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-readonly-pods
  namespace: demo
subjects:
- kind: Group
  name: read-only-group      # This matches the group in aws-auth
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: readonly-pods
```

apiGroup: rbac.authorization.k8s.io
Apply it:
bash

```
kubectl apply -f rbac-readonly.yaml
```

4️⃣ Test Access (From kube_test_user CLI)

From a system where the kube_test_user is configured in AWS CLI:
bash

```
aws eks update-kubeconfig --name your-cluster-name --region your-region \
  --role-arn arn:aws:iam::<account-id>:user/kube_test_user
```
Try this:
bash

```
kubectl get pods -n demo
```
✅ This should work.
Try this:
bash

```
kubectl delete pod mypod -n demo
```
⛔ This should fail – because the role is read-only.

💡 Real-World Use Case

- You want **developers to view Pods** but **not delete anything**.
- You want **DevOps engineers** to manage deployments only in dev namespace.
- You want **CI/CD systems** to have scoped permissions, not full admin rights.

---

2. What is AWS IAM integration with EKS?

Amazon EKS (Elastic Kubernetes Service) runs **Kubernetes clusters** on AWS.
To secure it, we need to:

1. **Control who can access the cluster** (admins, devs, CI/CD users).
2. **Control what running Pods can access in AWS** (like S3, DynamoDB, etc.).

To do this, EKS integrates with **AWS IAM (Identity and Access Management)**.

Two Key Parts of IAM Integration in EKS

Purpose	Technique	Use Case
Cluster access (users)	aws-auth ConfigMap	Map IAM users/roles to Kubernetes users
Pod access to AWS services	**IAM Roles for Service Accounts (IRSA)**	Let apps in pods access S3, DynamoDB, etc. securely

---

✅ **Part 1: IAM → Kubernetes Access (via aws-auth ConfigMap)**
📌 **What it does:**
Maps **IAM Users or Roles** to **Kubernetes RBAC identities**.
📦 **Where?**
In a special ConfigMap inside the EKS cluster:
bash

```
kubectl edit configmap aws-auth -n kube-system
```
🧾 **Example:**
yaml

```
mapUsers:
- userarn: arn:aws:iam::123456789012:user/dev
  username: dev-user
  groups:
  - system:masters
```
This allows:

- IAM user dev to connect to the EKS cluster as dev-user
- Grants them full access (system:masters) unless you restrict with RBAC

🧪 **Check access:**
bash

```
kubectl auth can-i list pods --as dev-user
```

✅ **Part 2: IAM Roles for Service Accounts (IRSA)**
🧩 **Problem:**
Let's say a **pod** needs to access an **S3 bucket**. How do we give it access **safely**, without hardcoding AWS credentials?
✅ **Solution: Use IRSA (IAM Roles for Service Accounts)**
IRSA = Secure way to attach an IAM Role to a Kubernetes ServiceAccount.
So, the Pod assumes that IAM Role, and gets permissions to AWS services — no hardcoded creds.

🔄 **How IRSA Works**

1. ✅ You create an **IAM Role** with the needed AWS permissions.
2. ✅ Attach an **OIDC trust policy**, so EKS knows this role can be assumed by a Pod.
3. ✅ In Kubernetes, you create a **ServiceAccount** linked to that IAM Role.
4. ✅ Your **Pod uses that ServiceAccount**, and it gets AWS access automatically.

🧾 **Example YAML for IRSA:**
IAM Role trust policy:
json

```
{
  "Effect": "Allow",
  "Principal": {
```

"Federated": "arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-west-2.amazonaws.com/id/EXAMPLED539D4633E53DE1B716D3041E"
 },
 "Action": "sts:AssumeRoleWithWebIdentity",
 "Condition": {
  "StringEquals": {
   "oidc.eks...:sub": "system:serviceaccount:default:myapp-sa"
  }
 }
}

Kubernetes ServiceAccount:

yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
 name: myapp-sa
 annotations:
  eks.amazonaws.com/role-arn: arn:aws:iam::123456789012:role/MyAppIRSA
```

Then in your Deployment/Pod spec:

yaml

```
serviceAccountName: myapp-sa
```

Now the pod can access AWS services like S3 or DynamoDB with **fine-grained permissions**.

🧠 **Summary for Beginners**

Feature	**Purpose**	**Where It Works**
aws-auth ConfigMap	Maps **IAM users/roles** to **K8s users**	Controls **who** can access the EKS cluster
IRSA	Maps **IAM roles** to **Pods**	Controls what **apps** inside EKS can access in AWS

🔐 **Real-Life Use Cases**

- A **developer** logs in using their IAM user to view resources: aws-auth mapping.
- A **Pod uploads files to S3**: IRSA.
- A **CI/CD pipeline** (e.g., GitHub Actions) assumes an IAM role and deploys to EKS.

.

---

3. How do you configure service accounts in EKS?

a🧠 **First: What's the Problem We're Solving?**

Imagine you have a **Pod in EKS** that needs to access:

- An **S3 bucket** to upload logs
- Or a **DynamoDB table** to read/write data

But — **how does that Pod get AWS credentials securely?**

We don't want to:

- Hardcode AWS keys in the Pod
- Use overly broad permissions

✅ **Solution: Use IAM Roles for Service Accounts (IRSA)**

With IRSA, you attach a specific **IAM Role** to a **Kubernetes ServiceAccount**.

Pods using that ServiceAccount will **automatically get AWS credentials** — securely and scoped.

🔧 Steps to Configure IRSA in EKS (Beginner Friendly)

🔷 Step 1: Enable IAM OIDC Provider

Each EKS cluster needs an **OIDC provider** (OpenID Connect) so IAM can trust Kubernetes.

✅ Run this command:

bash

```
eksctl utils associate-iam-oidc-provider \
  --region <your-region> \
  --cluster <your-cluster-name> \
  --approve
```

This allows your IAM roles to trust tokens from your EKS cluster.

🔷 Step 2: Create IAM Policy

Create a policy with **least-privilege AWS permissions**.

✅ Example: S3 Read/Write

json

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

Save it as s3-access-policy.json, then run:

bash

```
aws iam create-policy \
  --policy-name S3AccessPolicy \
  --policy-document file://s3-access-policy.json
```

🔷 Step 3: Create IAM Role with Trust Policy

This IAM Role:

- Can only be assumed by EKS Pods
- Only when using a specific ServiceAccount

✅ Use this trust policy (replace placeholders):

json

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "Federated": "arn:aws:iam::<account-id>:oidc-provider/oidc.eks.<region>.amazonaws.com/id/<eks-id>"
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "oidc.eks.<region>.amazonaws.com/id/<eks-id>:sub": "system:serviceaccount:default:s3-access-sa"
      }
    }
  }]
}
```

Then create the IAM role:

bash

```
aws iam create-role \
  --role-name S3AccessRole \
  --assume-role-policy-document file://trust-policy.json
```

And attach the S3 policy to it:

bash

```
aws iam attach-role-policy \
  --role-name S3AccessRole \
  --policy-arn arn:aws:iam::<account-id>:policy/S3AccessPolicy
```

🔷 Step 4: Create a Kubernetes ServiceAccount (with annotation)

This is the **link between your pod and the IAM role**.

📃 Example YAML:

yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: s3-access-sa
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::<account-id>:role/S3AccessRole
```

Apply it:

bash

```bash
kubectl apply -f sa.yaml
```

🔷 Step 5: Use the ServiceAccount in Your Pod/Deployment

Now update your Pod or Deployment YAML to **use the service account**.

📜 Example:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      serviceAccountName: s3-access-sa
      containers:
      - name: mycontainer
        image: myapp:latest
        # App can now access AWS S3 via SDK without hardcoding credentials
```

✅ Why Is This Great?

- 🔐 **Secure**: No need to store access keys in pods
- 🎯 **Fine-grained access**: Each pod gets only the permissions it needs
- 📦 **Cloud-native**: Works seamlessly with AWS SDKs and CLI in your app

---

Summary

Step	What You Do	Why
1️⃣	Enable OIDC	So IAM can trust EKS tokens
2️⃣	Create IAM Policy	Define what AWS resources are allowed
3️⃣	Create IAM Role with Trust	Link the IAM Role to EKS ServiceAccount

Step	What You Do	Why
4	Create Kubernetes ServiceAccount	Annotate it with IAM Role ARN
5	Use ServiceAccount in Pod	Your app gets secure AWS access automatically

---

4. What are Pod Security Standards and how do you implement them?

🚧 **In simple terms:**

PSS are **rules that control how secure your Pods are** — like what they're allowed to do, and what they're **not** allowed to do.

Kubernetes defines **3 security levels**:

Level	Purpose
privileged	No restrictions. Good for debugging, but unsafe for production.
baseline	Prevents known bad practices (e.g., running as root). Safer for most workloads.
restricted	Very strict. Follows best practices for least privilege and hardening. Ideal for production.

❗ Why Do You Need PSS in EKS?

Without any restrictions:

- Pods could run as **root**
- Mount **host paths** (accessing the EC2 machine itself!)
- Use **privileged mode** (full admin rights inside container)

This opens up security risks like container breakout attacks.

So, PSS lets you **enforce rules** at the **namespace level** to protect the cluster.

🔧 How to Implement Pod Security Standards (Step-by-Step)

Kubernetes 1.25+ (used in modern EKS versions) comes with a built-in **PodSecurity admission controller**.

✅ Step 1: Choose Your Namespace

Let's say your namespace is dev.

You want all pods in dev to follow restricted security policies.

✅ Step 2: Add Labels to Enforce PSS

Use this command:

bash

```
kubectl label namespace dev \
  pod-security.kubernetes.io/enforce=restricted
```

✅ This means: **Pods in the dev namespace must follow the restricted policy.**

You can also add optional labels for **audit** or **warn**:

```bash
pod-security.kubernetes.io/warn=baseline
pod-security.kubernetes.io/audit=privileged
```

So Kubernetes will **warn** or **log** if a pod violates less strict policies.

🧾 Example Label Setup:

Label	Meaning
pod-security.kubernetes.io/enforce=restricted	Block anything below restricted
pod-security.kubernetes.io/warn=baseline	Show warning if policy is below baseline
pod-security.kubernetes.io/audit=privileged	Log policy violations of privileged level

---

🧪 Example: What Gets Blocked?

If someone tries to apply this Pod YAML:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: badpod
spec:
  containers:
  - name: alpine
    image: alpine
    securityContext:
      privileged: true
```

It will get **denied** in a namespace labeled restricted because:

- Running a privileged container is **not allowed** under restricted mode.

🚀 Bonus: Advanced Policy with OPA Gatekeeper or Kyverno

If you want **custom rules** (e.g., "pods must use a specific image registry" or "no containers with hostPath"), use:

Tool	Use Case
OPA Gatekeeper	Policy as code using Rego language
Kyverno	Simpler YAML-based policy engine, very Kubernetes-native

These tools provide **richer validation**, auditing, and auto-remediation.

🧠 Summary for Beginners

Concept	Meaning
Pod Security Standards (PSS)	Built-in security levels for Pods
privileged	No restrictions (not safe)
baseline	Medium safety — blocks known bad configs
restricted	Strong safety — best for production
Namespace labels	Used to enforce security levels

Concept	Meaning
OPA/Gatekeeper/Kyverno	Used for custom advanced policy enforcement

---

💡 **Real-World Use Case in EKS**

- You run a **multi-tenant EKS cluster** (e.g., dev, staging, prod).
- You want to make sure **developers can't run insecure containers**.
- You label dev and prod namespaces with different PSS levels.
- In prod, only hardened, secure workloads can run.

---

5. How do you handle secrets management in EKS?

s 🔐 What Are Secrets and Why Manage Them?

In applications, **secrets** = sensitive info like:

- Database passwords
- API tokens
- SSH keys
- AWS credentials

If these secrets are not stored securely, **anyone who compromises a pod** can steal them — leading to data leaks or system compromise.
So in EKS (Kubernetes on AWS), managing secrets safely is **critical**.

🧰 3 Options for Secrets Management in EKS

Let's go through them **from basic to advanced**, so beginners can understand and build confidence.

✅ Option 1: Kubernetes Secrets (Basic)

- Secrets are stored inside Kubernetes using built-in Secret resources.
- They are **base64-encoded**, not encrypted by default.
- Stored in **etcd**, which should be encrypted (in EKS, it is by default).

📦 **Example YAML:**

yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
data:
  username: YWRtaW4=      # "admin"
  password: MWYyZDFlMmU2N2Rm # "1f2d1e2e67df"
```

To create it:

bash

```bash
kubectl apply -f secret.yaml
```

Then in a Pod:

yaml

```yaml
env:
- name: DB_USER
  valueFrom:
    secretKeyRef:
      name: db-creds
      key: username
```

⚠️ **Limitations:**

- Not encrypted at rest (unless etcd encryption is configured).
- Secrets visible to anyone with access to the namespace.
- Better for **non-critical** dev/test use.

✅ Option 2: AWS Secrets Manager with IRSA (Production-Ready)

AWS Secrets Manager is a **fully managed secrets service**:

- Secrets are encrypted with KMS.
- You can **rotate secrets automatically**.
- Access is controlled via **IAM** (and in EKS, via **IRSA**).

🔄 **Flow:**

1. Store your secret in AWS Secrets Manager.
2. Create a Kubernetes **ServiceAccount with IRSA**.
3. Pod uses AWS SDK (or init/sidecar container) to **fetch the secret securely**.

🔧 **Example:**

Store a secret:

bash

```bash
aws secretsmanager create-secret \
  --name db-creds \
  --secret-string '{"username":"admin","password":"s3cr3t"}'
```

In your app (Node.js, Python, etc.), use the AWS SDK:

js

```js
const client = new SecretsManager();
const secret = await client.getSecretValue({ SecretId: 'db-creds' }).promise();
```

✅ Best for production. Supports **audit logging, rotation, and IAM-based fine control**.

✅ Option 3: Secrets Store CSI Driver (Recommended Integration)

This lets Kubernetes **mount external secrets** (like from Secrets Manager or Vault) as **volumes**, so your app doesn't need to call AWS SDK directly.

This is great if:

- You don't want to modify your app code.
- You want secret values injected directly as files.

🔧 **How It Works:**

1. Install **Secrets Store CSI Driver** in your EKS cluster.
2. Install AWS provider plugin.
3. Create a SecretProviderClass that tells which secrets to pull.
4. Mount those secrets in Pod using a CSI volume.

📄 Example:

yaml

```yaml
--------
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
 name: aws-secrets
spec:
 provider: aws
 parameters:
  objects: |
   - objectName: "db-creds"
    objectType: "secretsmanager"
```

Then in your pod:

```yaml
--------
volumeMounts:
- name: secrets-store
 mountPath: "/mnt/secrets"

volumes:
- name: secrets-store
 csi:
  driver: secrets-store.csi.k8s.io
  readOnly: true
  volumeAttributes:
   secretProviderClass: "aws-secrets"
```

Now /mnt/secrets/db-creds contains the secret securely — no need for base64 decoding or SDK calls.

🧠 Summary for Beginners

Method	Description	Best For
Kubernetes Secrets	Built-in, simple, stored in etcd	Dev/test environments
AWS Secrets Manager + IRSA	Securely access AWS secrets via IAM roles	Production apps needing AWS secrets
Secrets Store CSI Driver	Mount secrets as volumes from AWS Secrets Manager/Vault	No-code change, production-grade

---

✅ Recommendation for Beginners in EKS

- **Start with Kubernetes Secrets** in test/dev.
- Learn **IRSA** to fetch from **AWS Secrets Manager** in production.
- Use **Secrets Store CSI Driver** for **secure volume-based access** — especially useful in regulated environments.

---

6. What are the security best practices for EKS?

✅ Overview: Why Security in EKS Matters

EKS (Elastic Kubernetes Service) gives you the power of Kubernetes on AWS.
But with great power comes great responsibility — especially when:

- You're deploying apps in the cloud
- There are many users, teams, and microservices
- A single misconfigured Pod or secret can compromise your system

So let's break down the **main security areas**.

🔒 1. Access Control (Who can access what)

✅ Use RBAC (Role-Based Access Control)

- Give users only the **permissions they need**.
- Example: Devs can view Pods, but only Admins can delete Deployments.

✅ Use IRSA (IAM Roles for Service Accounts)

- Let Pods access AWS securely without hardcoded keys.
- Example: A logging pod gets permission to write to S3, and nothing else.

🧠 Analogy: Think of RBAC like keycards in an office — each person can only enter specific rooms.

🌐 2. Network Security

✅ Use Network Policies to Control Pod-to-Pod Traffic

- Default in Kubernetes: All Pods can talk to each other 🤯
- Use Calico or Cilium to define **who can talk to whom**.

✅ Isolate Environments

- Use **namespaces** to separate dev/test/prod.
- Use **Security Groups** to restrict access at the EC2/network level.

🧠 Analogy: Think of Pods as rooms in a building — network policies act like doors that can be locked or opened only for specific rooms.

🔐 3. Secrets Management

✅ Prefer AWS Secrets Manager or Parameter Store

- These encrypt secrets with KMS and offer rotation, auditing, and fine-grained control.

✅ Avoid:

- Plain YAML secrets
- Storing secrets directly in environment variables

🧠 Tip: Use IRSA + AWS SDK or Secrets Store CSI Driver to fetch secrets at runtime.

🛡️ 4. Node Security

✅ Use Managed Node Groups

- AWS patches the OS and handles lifecycle updates.

✅ Choose Minimal AMIs:

- **Bottlerocket** (by AWS) or **EKS-Optimized Amazon Linux 2**
- Fewer packages = smaller attack surface

✅ Apply regular updates

- Use **EKS Managed Node Groups** with scheduled updates or lifecycle automation.

🧱 5. Pod Security

✅ **Enforce Pod Security Standards (PSS)**

- Label namespaces with enforce=restricted
- Prevent running as root, limit privilege escalation

✅ **Harden Pod Config:**

- Use readOnlyRootFilesystem: true
- Drop Linux capabilities
- Add resource limits (CPU/memory)

🧠 Example:

yaml

```
securityContext:
 runAsNonRoot: true
 readOnlyRootFilesystem: true
 capabilities:
  drop: ["ALL"]
```

👁 6. Observability & Auditing

✅ **Enable:**

- **CloudTrail**: Logs AWS API calls
- **VPC Flow Logs**: Logs network traffic
- **EKS Audit Logs**: Enable via CloudWatch for visibility into cluster events
- **GuardDuty**: Detects unusual behavior or attacks

🧠 Tip: Always **monitor your cluster and set up alerts**.

📦 7. Image and Supply Chain Security

✅ **Use Trusted and Minimal Base Images**

- Avoid "ubuntu:latest" — use slim or distroless images

✅ **Scan Images Before Deployment**

- Tools: **Trivy**, **Clair**, **Grype**
- Can be automated in CI/CD

✅ **Use Image Signing with Sigstore / Cosign**

- Verify that images are trusted and not tampered with

✅ Quick Recap: EKS Security Checklist

Area	Best Practice
Access Control	RBAC + IRSA
Network Security	NetworkPolicies + Namespaces + Security Groups
Secrets	Use AWS Secrets Manager + IRSA + CSI Driver
Nodes	Use Bottlerocket / Amazon Linux 2 AMIs + auto patching
Pods	Enforce PSS, no root, drop caps, read-only FS
Auditing	Enable CloudTrail, VPC Flow Logs, Audit Logs
Image Security	Scan, sign, and use minimal trusted images

How to Start

1. **Set up RBAC** (user access)

2. **Enable IRSA** for apps that use AWS services
3. **Use Kubernetes Secrets initially**, then migrate to **AWS Secrets Manager**
4. **Use EKS Managed Node Groups**
5. **Label namespaces with restricted** for pod security
6. **Enable audit logging** in EKS settings
7. **Scan your container images** using Trivy locally or in CI/CD

1. What is the purpose of using RBAC in EKS?
A. To encrypt secrets inside Kubernetes
B. To define which Pods can access external services
C. To control which users or roles can access Kubernetes resources
D. To control network traffic between Pods
✅ **Answer: C**

---

2. Why should you use IAM Roles for Service Accounts (IRSA) in EKS?
A. To allow Secrets Manager to authenticate with Kubernetes
B. To let pods securely access AWS services without hardcoding credentials
C. To allow Kubelet to install pods faster
D. To enable external CI/CD pipelines to deploy applications
✅ **Answer: B**

Where are Kubernetes Secrets stored by default in EKS?
A. In S3 buckets
B. In AWS Secrets Manager
C. In DynamoDB
D. In etcd (Kubernetes data store)
✅ **Answer: D**

What is the main limitation of Kubernetes Secrets by default?
A. They cannot be accessed by pods
B. They are encrypted with KMS
C. They are stored as plaintext unless etcd encryption is enabled
D. They require IRSA to be accessed
✅ **Answer: C**
