Scenario based questions on EKS

1. You need to upgrade the EKS cluster from version 1.24 to 1.27. What's your strategy?

To upgrade an EKS cluster from v1.24 → v1.27, I would follow a safe, phased, and tested approach since AWS supports only sequential upgrades (one minor version at a time).

🔍 **1. Understand the Upgrade Path**

EKS only supports sequential version upgrades, so I need to upgrade in steps:
1.24 → 1.25 → 1.26 → 1.27

🧪 **2. Pre-upgrade Checks (for each version)**

✅ **a. Check Deprecation Notices**

- Review AWS EKS Release Notes
- Check for:
  - Removed APIs
  - Deprecated admission controllers
  - Security policy changes

✅ **b. Audit Workloads & Tools**

- Run:

```bash
kubectl get apiservices
kubectl get ingresses --all-namespaces
```

- Scan Helm charts for deprecated APIs (e.g., extensions/v1beta1)
- Use tools like pluto or kubent to detect deprecated Kubernetes resources

🧪 **3. Test in Staging First**

- Always upgrade a non-prod/staging cluster first to validate workloads and CRDs
- Use canary deployments if feasible

🛠️ **4. Upgrade Control Plane (Sequentially)**

For each version step (1.24 → 1.25 → 1.26 → 1.27):

```bash
aws eks update-cluster-version --name my-cluster --kubernetes-version 1.25
```

- Wait for completion before proceeding to the next version

- Monitor from the EKS Console or using eksctl:
  bash
  eksctl upgrade cluster --name my-cluster --version 1.25

  📦 **5. Upgrade Managed Node Groups (After Control Plane)**
- Upgrade one node group at a time (in-place or with replacement):
  bash
  eksctl upgrade nodegroup --cluster my-cluster --name ng-1 --kubernetes-version 1.25
- Use rolling upgrades to avoid downtime

  🔁 **6. Update Add-ons (After Complete Upgrade)**

  Update EKS-managed add-ons after completing the entire cluster upgrade to v1.27, not at each intermediate step:
- Check add-on compatibility matrix for target Kubernetes version
- Update add-ons like: vpc-cni, kube-proxy, CoreDNS, aws-ebs-csi-driver
  bash
  aws eks update-addon --cluster-name my-cluster --addon-name vpc-cni --addon-version <compatible-version>

  🔬 **7. Post-Upgrade Validation**
- Confirm:
  bash
  - kubectl get nodes Pods are running
  - DNS and network works
  - Metrics/logs integrations are functional

  kubectl get pods --all-namespaces
  kubectl top nodes
- Run smoke tests / health checks for services

  🧠 **8. Document and Automate**
- Use Infrastructure as Code (e.g., eksctl, Terraform) for reproducibility
- Document upgrade steps and issues found for future upgrades

  🎯 **Summary**: "I'd upgrade EKS sequentially from 1.24 to 1.27 by upgrading the control plane, then node groups in each step, and finally updating add-ons after the complete upgrade. I'd test in staging first, validate workloads, and monitor for issues using tools like kubent, eksctl, and AWS CLI."

2 Your company wants to host multiple teams on the same EKS cluster. How will you isolate workloads?

To isolate workloads for different teams (e.g., frontend, backend, data, ML) in the **same EKS cluster**, I'd implement **multi-layered isolation** using Kubernetes and AWS-native tools across **namespaces, IAM, RBAC, network, and resource boundaries**.

📦 **1. Use Separate Namespaces for Each Team**
- Create a **namespace per team**:
  team-a, team-b, data-team, etc.
- Benefits:

    - Logical separation
    - Quota & policy enforcement
    - Easier cleanup and auditing

bash

```
kubectl create namespace team-a
```

🛡️ 2. Apply RBAC Policies

- Define **RoleBindings** or **ClusterRoleBindings** to grant team-specific permissions **only in their namespace**.
- Prevent access to shared or other teams' resources.

yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: team-a-admin
  namespace: team-a
subjects:
- kind: User
  name: team-a-dev
roleRef:
  kind: Role
  name: admin
  apiGroup: rbac.authorization.k8s.io
```

🔐 3. Use IAM Roles for Service Accounts (IRSA)

- Assign **team-specific IAM roles** to their app's service accounts.
- Prevent cross-team AWS access (like accessing someone else's S3 bucket or RDS).

yaml

```yaml
serviceAccount:
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::<account>:role/team-a-s3-reader
```

🌐 4. Enforce Network Policies (Optional but Recommended)

- Use **Calico** or **Cilium** to restrict pod-to-pod communication across teams.
- Example: only allow team-a namespace to talk to its own services.

yaml

```yaml
kind: NetworkPolicy
spec:
  podSelector: {}
  policyTypes: [Ingress, Egress]
  ingress:
    - from:
```

```yaml
- namespaceSelector:
    matchLabels:
      team: team-a
```

💰 5. Apply Resource Quotas and Limits

- Prevent noisy neighbors by restricting CPU/memory usage per namespace.

yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-a-quota
  namespace: team-a
spec:
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "10"
    limits.memory: "16Gi"
```

🧪 6. Use Tools for Multi-Tenant Governance

- Use **OPA/Gatekeeper** or **Kyverno** to enforce security rules like:
  - Must use resource limits
  - Must use approved container registries
- Use **ArgoCD** with per-namespace access control for GitOps.

📈 7. Observability per Team

- Set up **per-namespace log/metric collection** (via FluentBit/Prometheus).
- Use labels and namespaces to filter metrics in Grafana or logs in CloudWatch.

🔒 Limit Cluster-Wide Permissions

- Avoid giving teams access to cluster-wide roles like ClusterAdmin.
- Use tools like kube-ops-view with read-only access.

---

3. Your monthly EKS bill has unexpectedly doubled. How do you analyze and reduce costs?

I would use **Cost Explorer**, **Kubecost**, and AWS reports to find out which components (nodes, storage, logs, network) caused the cost spike. Then I'd reduce costs by **right-sizing resources**, **using Spot Instances**, **cleaning up unused assets**, and setting up **autoscaling and observability tools** to prevent future surprises.

If the EKS bill has doubled unexpectedly, I'd follow a **3-phase strategy**:

1. **Analyze the cost spike**
2. **Identify the root cause**
3. **Apply optimization techniques**

🔍 PHASE 1: Analyze Cost Breakdown

✅ a. Use AWS Cost Explorer

- Filter by **Service = EKS**, **EC2**, **EBS**, **ECR**, **CloudWatch**, etc.
- Check for:
    - Node instance hours
    - EBS volume growth
    - Networking spikes (e.g., cross-AZ traffic)
    - Add-on services like NAT Gateway, CloudWatch Logs

✅ b. Use AWS CUR (Cost and Usage Reports)

- Enable detailed cost visibility at **resource level** (e.g., NodeGroupA, namespace, account)

✅ c. Use Kubecost / CloudZero / Finout

- Tools like **Kubecost** break down:
    - Cost per **namespace**
    - Cost per **deployment/pod**
    - Overprovisioning or idle resources

🧱 PHASE 2: Identify Root Causes

Look for:

Issue Type	What to Check
🖥 Node Cost Spike	Increase in node count, larger instance types, spot fallback
🧠 Overprovisioning	High CPU/memory requests not actually used
📦 Unused Resources	Idle pods, failed jobs, orphan volumes, unused ELBs
🌐 Networking Costs	High cross-AZ or internet egress usage
📈 Logging Costs	Excessive CloudWatch logs (e.g., debug logs from all pods)
🔄 Autoscaler Behavior	Cluster Autoscaler/Karpenter scaling too aggressively

---

💰 PHASE 3: Reduce & Optimize EKS Costs

✅ 1. Right-Size Node Groups

- Use **smaller EC2 instance types** or **consolidate underutilized nodes**
- Consider **Graviton instances** (e.g., t4g, m6g) for 20–40% savings

✅ 2. Enable Horizontal Pod Autoscaler (HPA)

- Scale pods based on actual CPU/memory usage
- Avoid overprovisioned workloads

✅ 3. Use Spot Instances (with fallback)

- Use **Spot nodes** for stateless or batch workloads
- Combine with **Managed Node Groups** or **Karpenter**

✅ 4. Clean Up Unused Resources

- Delete:
  - Idle services
  - Orphaned PVCs
  - Zombie ELBs
  - Completed jobs or test deployments

✅ **5. Optimize CloudWatch Logs**

- Reduce log volume:
  - Set appropriate log levels (INFO instead of DEBUG)
  - Use **FluentBit filters**
  - Retention policy (e.g., 7 days)

✅ **6. Apply Resource Requests & Limits**

- Avoid over-requesting CPU/memory:

yaml

```yaml
requests:
  cpu: "100m"
  memory: "128Mi"
limits:
  cpu: "200m"
  memory: "256Mi"
```

✅ **7. Schedule Workloads Off-Hours**

- Use **KEDA** or scheduled jobs to run non-critical workloads only during working hours.

---

4. Your production EKS cluster goes down due to a region failure. What's your DR plan?

To handle a **region-wide failure of an EKS cluster**, I'd implement a **cross-region disaster recovery plan** with a mix of **proactive backups**, **infrastructure automation**, and **data replication**. Here's the strategy:

📦 **1. EKS Infrastructure as Code (IaC)**

- Use **eksctl**, **Terraform**, or **CDK** to define EKS clusters and node groups.
- This ensures we can quickly spin up a **standby or hot cluster in another region** (e.g., from us-east-1 to us-west-2) in minutes.

bash

```bash
eksctl create cluster -f eks-cluster-config.yaml --region us-west-2
```

💾 **2. Backup and Replicate Critical Data**

✅ **a. Persistent Volumes (EBS)**

- Use **Velero** to back up Kubernetes resources and EBS volumes.
- Store backups in **S3** with cross-region replication.

bash

-----

velero install --provider aws --bucket my-eks-backup --backup-location-config region=us-east-1

✅ **b. Databases**

- Use **Amazon RDS with cross-region read replicas** or **Aurora Global Databases** for near real-time replication.
- Promote read replica in DR region upon failover.

🚀 3. Deploy Workloads to DR Cluster

- Use **CI/CD pipelines** (e.g., GitHub Actions, ArgoCD, CodePipeline) to redeploy apps to the DR cluster.
- Store Kubernetes manifests in Git (GitOps).

bash

```
kubectl apply -f deployment.yaml --context=dr-cluster
```

🌐 4. DNS-Level Failover

- Use **Route 53 with health checks** to:
  - Detect regional failure
  - Automatically failover to DR cluster's endpoints (ALB/NLB)

text

```
Primary ALB (us-east-1) —> Unhealthy —> Switch to DR ALB (us-west-2)
```

🔐 5. Secrets & IAM Access

- Replicate secrets using:
  - **AWS Secrets Manager (cross-region replication)** or
  - **External Secrets Operator** with multi-region SSM/Secrets Manager
- Use **IRSA** in both regions to manage pod-to-AWS access securely.

🧪 6. Run Regular DR Drills

- Practice failover every quarter.
- Validate:
  - RTO (Recovery Time Objective)
  - RPO (Recovery Point Objective)
  - End-to-end availability in DR region

Warm vs. Hot DR Strategy

DR Type	Characteristics
🔥 Hot DR	EKS & workloads always running in 2 regions (active-active or active-passive)
🌤️ Warm DR	Infrastructure is ready, workloads are deployed on failover
❄️ Cold DR	Infra created on-demand (slower RTO)

Final Summary:

“My DR plan includes **IaC for cluster setup**, **data & secret replication**, **GitOps-based workload redeployments**, and **Route 53 DNS failover**. I’d test failovers regularly to ensure low RTO/RPO and be ready to bring up the system in another region with minimal downtime.”

RTO - Recovery Time Objective

Definition: The maximum acceptable amount of time it takes to restore a system or service after a disaster occurs.

Key Points:

- Measures **how quickly** you need to recover
- Starts from the moment of failure until full service restoration
- Directly impacts business operations and user experience
- Usually measured in minutes, hours, or days

Example: "Our EKS application must be restored within 30 minutes of a regional failure" (RTO = 30 minutes)

RPO - Recovery Point Objective

Definition: The maximum acceptable amount of data loss measured in time - essentially how far back in time you can afford to go when recovering data.

Key Points:

- Measures **how much data** you can afford to lose
- Determines backup frequency requirements
- Focuses on data currency, not recovery speed
- Usually measured in minutes, hours, or days

Example: "We can tolerate losing up to 15 minutes of transaction data" (RPO = 15 minutes)

Practical EKS Examples:

E-commerce Application:

- **RTO**: 15 minutes (customers can't shop for long)
- **RPO**: 5 minutes (recent orders/payments are critical)
- **Solution**: Hot DR with real-time database replication

Internal Analytics Dashboard:

- **RTO**: 4 hours (not customer-facing)
- **RPO**: 1 hour (some data loss acceptable)
- **Solution**: Warm DR with hourly backups

Batch Processing System:

- **RTO**: 24 hours (can run next day)
- **RPO**: 24 hours (can reprocess yesterday's data)
- **Solution**: Cold DR with daily backups

RTO vs RPO Relationship:

```
Time →
[Normal Operations] → [DISASTER] → [Recovery Period] → [Restored Operations]
                    ↑           ↑              ↑
              Failure Time   Last Backup      Full Recovery
                 ←— RPO —→
                 ←————————— RTO ——————————→
```

Cost vs Requirements:

Requirements	Cost	Infrastructure Needed
RTO: Minutes, RPO: Minutes	Very High	Active-active setup, real-time replication
RTO: Hours, RPO: Hours	Medium	Warm standby, frequent backups
RTO: Days, RPO: Days	Low	Cold standby, daily backups

How This Affects Your EKS DR Strategy:

For Aggressive Requirements (RTO: 5 min, RPO: 1 min):

- Multi-region active-active EKS clusters
- Real-time database replication (Aurora Global)
- Continuous data synchronization
- Automated failover with Route 53

For Moderate Requirements (RTO: 30 min, RPO: 15 min):

- Warm standby cluster ready to activate
- 15-minute backup intervals with Velero
- Automated deployment pipelines
- DNS failover with health checks

For Relaxed Requirements (RTO: 4 hours, RPO: 1 hour):

- Infrastructure as Code for cluster recreation
- Hourly backups to cross-region S3
- Manual failover procedures
- Acceptable brief service interruption

Understanding RTO and RPO helps you design the right balance between cost, complexity, and business requirements for your disaster recovery strategy.

1. AWS EKS supports upgrading between:
   a) Any two versions directly
   b) Only one minor version at a time
   c) Only patch versions
   d) Major version jumps only

b) Only one minor version at a time ✅

2. Add-ons like vpc-cni and CoreDNS should be updated:
   a) Before starting the upgrade
   b) After each intermediate version
   c) After completing the final version upgrade
   d) Never

c) After completing the final version upgrade
✅

Which tool can enforce security policies like “must use resource limits”?
a) Prometheus

b) OPA/Gatekeeper
c) FluentBit
d) CloudWatch

b) OPA/Gatekeeper ✅

Which AWS tool provides per-pod cost breakdown in EKS?
a) Cost Explorer
b) Kubecost
c) CloudTrail
d) Route 53

b) Kubecost ✅
Which tool can back up EKS workloads and EBS volumes?
a) Velero ✅
b) Prometheus
c) ArgoCD
d) Karpenter

a) Velero ✅