Amazon EKS

Cluster & Upgrades

Elastic Kubernetes Service — Enterprise Production Guide

Architecture • Upgrades • POC • SOP • Interview Prep • Cost

Kubernetes | Managed Nodes | Fargate | Add-ons | Zero-Downtime | GitOps

Production Ready | AWS Container Platform | Enterprise Reference

February 2025 | Version 1.0

■■ What is Amazon EKS?

Amazon **Elastic Kubernetes Service (EKS)** is a **fully managed Kubernetes control plane** on AWS. It runs and scales the Kubernetes API server and etcd across multiple Availability Zones, eliminating the need to install, operate, and maintain your own Kubernetes control plane. EKS is certified Kubernetes-conformant — all standard kubectl commands, Helm charts, and Kubernetes manifests work without modification.

> ■ Interview One-Liner: EKS gives you a 100% upstream Kubernetes control plane managed by AWS (HA, patched, scaled) while you own and manage your worker nodes (EC2 Managed Node Groups, Fargate, or self-managed). You get Kubernetes without the control plane headache.

EKS integrates natively with AWS services: IAM for RBAC, VPC for networking, ALB/NLB for ingress, ECR for container images, CloudWatch for logging/monitoring, EBS/EFS for storage, and AWS Secrets Manager for secrets.

■ Why We Use Amazon EKS

#	Problem	How EKS Solves It
1	Managing K8s control plane is complex & expensive	AWS runs etcd + API server across 3 AZs — HA by default, zero ops
2	K8s version upgrades break production	EKS managed upgrades + add-on versioning + in-place node group rolling u
3	Node fleet management overhead	Managed Node Groups: auto-provisioning, patching, draining via AWS APIs
4	Serverless containers with K8s API	EKS Fargate: run pods without managing EC2 — pay per pod CPU/memory
5	IAM + Kubernetes RBAC gap	IRSA (IAM Roles for Service Accounts) — pod-level AWS IAM without secret
6	Auto-scaling complex to set up	Karpenter: node autoscaler provision right-sized nodes in < 60 seconds
7	Ingress/LB management	AWS Load Balancer Controller: ALB/NLB provisioned from K8s Ingress mani
8	Secrets management in K8s	Secrets Store CSI Driver: mount AWS Secrets Manager / SSM as K8s volum
9	Multi-cluster governance	EKS Connector + Amazon EKS Anywhere for hybrid/multi-cluster visibility
10	Cost visibility for K8s workloads	AWS Cost Allocation Tags + Kubecost integration for per-namespace cost

How Amazon EKS Works

EKS separates the **control plane** (AWS managed) from the **data plane** (you manage). The control plane runs the Kubernetes API server, controller manager, and etcd. The data plane is where your actual workloads (pods) run.

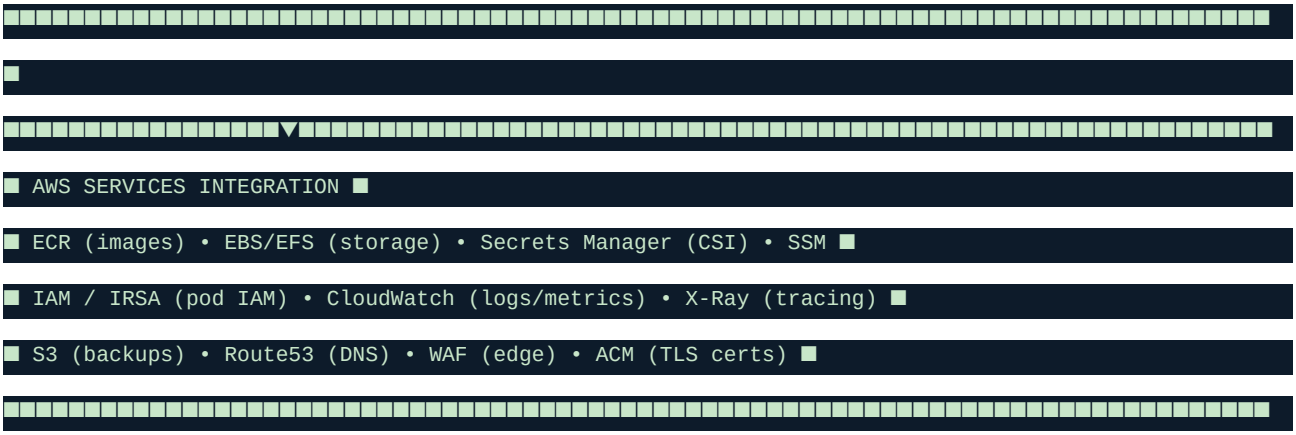
Component	Who Manages	Details
Control Plane	AWS	API Server, etcd, scheduler, controller manager — HA across 3 AZs, auto-patch
Managed Node Groups	AWS + You	AWS provisions, patches, and drains EC2 nodes; you define instance types and
Fargate Profile	AWS	Serverless pods — no node management; AWS picks instance, runs pod, charg
Self-Managed Nodes	You	Full EC2 control — custom AMI, GPU nodes, ARM, bare metal; manual patching
VPC CNI Plugin	AWS (add-on)	Each pod gets a real VPC IP — native VPC networking, security groups per pod
CoreDNS	AWS (add-on)	Cluster DNS for service discovery — managed and versioned via EKS add-ons
kube-proxy	AWS (add-on)	Network rules on each node for service routing — updated with cluster
AWS Load Balancer Controller	Community/AWS	Provisions ALB/NLB from K8s Ingress and Service specs
Karpenter	AWS (OSS)	Node autoscaler — provisions right-sized nodes in < 60s based on pod requirem
EBS/EFS CSI Driver	AWS (add-on)	Dynamic provisioning of EBS volumes and EFS mounts as K8s PersistentVolum
IRSA	AWS	Pod-level IAM: maps K8s ServiceAccount to IAM Role via OIDC — no static cre

Node Group Types Comparison

Type	Managed Node Groups	Fargate	Self-Managed Nodes
Who Provisions EC2	AWS	AWS (invisible)	You
Who Patches AMI	AWS	AWS	You
Scaling	Auto Scaling Group	Automatic per pod	ASG (manual config)
GPU / Custom AMI	Limited GPU support	No GPU	Full support
Cost Model	EC2 On-Demand/Spot	Per pod CPU/mem/sec	EC2 On-Demand/Spot
Best For	Standard workloads	Stateless, spiky workloads	Special hw, custom OS
Karpenter Support	Yes	No	Yes

■■ Enterprise Architecture Diagram

Multi-AZ EKS Production Cluster — Full AWS Integration


AWS SERVICES INTEGRATION
ECR (images) • EBS/EFS (storage) • Secrets Manager (CSI) • SSM
IAM / IRSA (pod IAM) • CloudWatch (logs/metrics) • X-Ray (tracing)
S3 (backups) • Route53 (DNS) • WAF (edge) • ACM (TLS certs)

■ How to Create an EKS Cluster — Step-by-Step

Step 1: Pre-requisites & Tooling

- Install: aws CLI v2, kubectl (matching cluster version), eksctl, helm 3, terraform (optional)
- IAM permissions: eks:*, ec2:*, iam:PassRole, cloudformation:* for the deploying principal
- VPC: 3 private subnets (worker nodes) + 3 public subnets (ALB/NAT) across 3 AZs
- Tag private subnets: kubernetes.io/role/internal-elb=1 | public: kubernetes.io/role/elb=1

Step 2: Create EKS Cluster via eksctl (Recommended)

- eksctl create cluster --name prod-cluster --region us-east-1 --version 1.29 \
- --nodegroup-name prod-ng --node-type m5.xlarge --nodes 3 --nodes-min 2 --nodes-max 10 \
- --managed --asg-access --external-dns-access --full-ecr-access \
- --alb-ingress-access --with-oidc --ssh-access --ssh-public-key my-key \
- --vpc-private-subnets subnet-a,subnet-b,subnet-c \
- --vpc-public-subnets subnet-x,subnet-y,subnet-z

Step 3: Configure kubectl

- aws eks update-kubeconfig --name prod-cluster --region us-east-1
- kubectl get nodes # Verify all nodes are Ready
- kubectl get pods -A # Verify CoreDNS, kube-proxy, VPC CNI pods are Running

Step 4: Install Core Add-ons

- # EBS CSI Driver (for PersistentVolumes on EBS)
- eksctl create addon --name aws-ebs-csi-driver --cluster prod-cluster --service-account-role-arn arn:aws:iam::ACCOUNT:role/EBSCSIRole
- # CoreDNS and kube-proxy managed via EKS console Add-ons tab
- # AWS Load Balancer Controller — install via Helm
- helm repo add eks https://aws.github.io/eks-charts
- helm install aws-load-balancer-controller eks/aws-load-balancer-controller -n kube-system \
- --set clusterName=prod-cluster --set serviceAccount.create=false --set serviceAccount.name=aws-load-balancer-controller

Step 5: Install Karpenter (Node Autoscaler)

- # Karpenter replaces Cluster Autoscaler — faster (< 60s) and smarter node provisioning
- helm upgrade --install karpenter oci://public.ecr.aws/karpenter/karpenter \
- --version v0.37.0 --namespace karpenter --create-namespace \

- --set settings.clusterName=prod-cluster \
- --set settings.interruptionQueue=prod-cluster
- # Apply NodePool manifest to define node constraints (instance families, AZs, arch)

Step 6: Setup IRSA (IAM Roles for Service Accounts)

- # Associate OIDC provider with cluster (if not done via eksctl --with-oidc)
- eksctl utils associate-iam-oidc-provider --cluster prod-cluster --approve
- # Create IAM Role for a service account
- eksctl create iamserviceaccount --name my-app-sa --namespace default \
- --cluster prod-cluster --attach-policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess \
- --approve --override-existing-serviceaccounts
- # Pod automatically gets AWS credentials via projected ServiceAccount token — no secrets!

Step 7: Enable CloudWatch Logging & Container Insights

- aws eks update-cluster-config --name prod-cluster --region us-east-1 \
- --logging '{"clusterLogging":[{"types":["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}'
- # Container Insights for pod/node metrics
- aws eks create-addon --cluster-name prod-cluster --addon-name amazon-cloudwatch-observability

Step 8: Deploy Sample Workload & Ingress

- kubectl create deployment nginx --image=nginx:alpine --replicas=3
- kubectl expose deployment nginx --port=80 --type=ClusterIP
- # Ingress with ALB
- kubectl apply -f ingress.yaml # Annotate: kubernetes.io/ingress.class: alb, alb.ingress.kubernetes.io/scheme: internet-facing
- kubectl get ingress # ALB DNS name appears within 2-3 minutes

■■ EKS Cluster Upgrades — Zero Downtime Strategy

EKS upgrades are one of the most operationally critical tasks. Kubernetes releases a new minor version every ~4 months. EKS supports N-3 versions. You must upgrade **one minor version at a time** (1.28 → 1.29 → 1.30). Skipping versions is NOT supported.

> **■■ CRITICAL: Always upgrade the Control Plane FIRST, then Node Groups, then Add-ons. Never upgrade nodes before the control plane. Skipping a minor version will fail.**

EKS Upgrade Order — The Golden Rule

Order	Component	Method	Estimated Time	Risk
1st	**Control Plane**	AWS Console / eksctl / CLI	~15-20 min	Low — AWS managed, HA
2nd	**Managed Add-ons**	AWS Console or eksctl update addon	~5 min each	Low — update independently
3rd	**Managed Node Groups**	Rolling update via AWS Console / CLI	~20-60 min	Medium — pods rescheduled
4th	**Self-Managed Nodes**	Manual: cordon, drain, replace	~Variable	High — manual process
5th	**Fargate Pods**	Delete & redeploy pods	~5-10 min	Low — stateless
6th	**Application Manifests**	Review deprecated K8s APIs	Depends on app	Medium — API breaks

Step-by-Step Upgrade Procedure (1.29 → 1.30)

PRE-FLIGHT: Pre-Upgrade Checklist (DO THIS FIRST)

- 1. Check current version: aws eks describe-cluster --name prod-cluster --query 'cluster.version'
- 2. Review EKS release notes for 1.30: https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-versions.html
- 3. Run kubectl get pods -A | grep -v Running — fix all non-Running pods BEFORE upgrade
- 4. Check deprecated API versions: kubectl convert --dry-run=client -f manifests/ (use pluto tool)
- 5. Verify PodDisruptionBudgets (PDBs) allow eviction: kubectl get pdb -A
- 6. Backup etcd (optional but recommended): take cluster snapshot or export manifests
- 7. Notify stakeholders: schedule upgrade window (off-peak hours)
- 8. Test upgrade in staging cluster FIRST — never upgrade prod without staging validation

STEP 1: Upgrade the Control Plane

- # Via CLI
- aws eks update-cluster-version --name prod-cluster --kubernetes-version 1.30
- # Monitor until ACTIVE
- aws eks describe-cluster --name prod-cluster --query 'cluster.status'
- aws eks describe-update --name prod-cluster --update-id
- # Wait for status = Successful (~15-20 minutes)
- kubectl version # Confirm server version is now 1.30

STEP 2: Update Managed Add-ons

- # List current add-on versions
- aws eks describe-addon-versions --kubernetes-version 1.30 | grep addonVersion
- # Update each add-on
- aws eks update-addon --cluster-name prod-cluster --addon-name vpc-cni --addon-version v1.18.x
- aws eks update-addon --cluster-name prod-cluster --addon-name coredns --addon-version v1.11.x
- aws eks update-addon --cluster-name prod-cluster --addon-name kube-proxy --addon-version v1.30.x
- aws eks update-addon --cluster-name prod-cluster --addon-name aws-ebs-csi-driver --addon-version v1.30.x
- # Verify each: aws eks describe-addon --cluster-name prod-cluster --addon-name vpc-cni

STEP 3: Upgrade Managed Node Groups (Rolling Update)

- # Trigger rolling update — AWS cordons, drains, terminates old nodes, launches new ones
- aws eks update-nodegroup-version --cluster-name prod-cluster --nodegroup-name prod-ng
- # Monitor node group update
- aws eks describe-nodegroup --cluster-name prod-cluster --nodegroup-name prod-ng --query 'nodegroup.status'
- # Watch pods reschedule in real-time
- kubectl get nodes -w
- kubectl get pods -A -w
- # Rolling update respects maxUnavailable (default 1 node at a time)

STEP 4: Upgrade Self-Managed Nodes (Manual Rolling)

- # For each node (one at a time):
- kubectl cordon # Mark unschedulable
- kubectl drain --ignore-daemonsets --delete-emptydir-data --grace-period=60
- # Terminate the EC2 instance — new one launches from updated launch template (new AMI)
- aws ec2 terminate-instances --instance-ids
- # Wait for new node to join and become Ready
- kubectl get nodes -w

- # Repeat for each node — NEVER drain all nodes simultaneously

STEP 5: Upgrade Fargate Pods

- # Fargate pods must be recycled — delete them and let controllers recreate
- kubectl rollout restart deployment --selector=eks.amazonaws.com/fargate-profile=my-profile -A
- # Or delete pods manually — Deployment/ReplicaSet will recreate on new Fargate runtime
- kubectl delete pods -l app=my-fargate-app -n my-namespace

STEP 6: Post-Upgrade Validation

- kubectl get nodes # All nodes show correct Kubernetes version
- kubectl get pods -A # All pods Running/Completed — no CrashLoopBackOff
- kubectl get events -A --sort-by=.lastTimestamp | tail -30 # Check for errors
- # Run smoke tests / integration tests against staging then prod
- # Verify Ingress, services, PVCs, HPA, RBAC all functioning
- # Update monitoring dashboards, runbooks with new version number
- # Document: upgrade date, version, operator, any issues encountered

■ POC — Deploy Microservices on EKS

This POC deploys a multi-tier application on EKS: React frontend + Node.js API + Redis cache, with ALB Ingress, HPA, and IRSA for S3 access.

1. Karpenter NodePool (Right-Sized Node Provisioning)

```
apiVersion: karpenter.sh/v1beta1 kind: NodePool metadata: name: default spec: template: spec: requirements: - key: kubernetes.io/arch operator: In values: ["amd64", "arm64"] - key: karpenter.sh/capacity-type operator: In values: ["spot", "on-demand"] - key: karpenter.k8s.aws/instance-family operator: In values: ["m5", "m6g", "c5", "c6g"] nodeClassRef: apiVersion: karpenter.k8s.aws/v1beta1 kind: EC2NodeClass name: default limits: cpu: "100" # Max 100 vCPUs in cluster memory: 400Gi disruption: consolidationPolicy: WhenUnderutilized consolidateAfter: 30s # Remove idle nodes after 30s
```

2. Application Deployment with IRSA

```
# api-deployment.yaml apiVersion: apps/v1 kind: Deployment metadata: name: api-server namespace: production spec: replicas: 3 selector: matchLabels: app: api-server template: metadata: labels: app: api-server spec: serviceAccountName: api-server-sa # Has IRSA — maps to IAM Role with S3 access containers: - name: api image: 123456789.dkr.ecr.us-east-1.amazonaws.com/api:v1.5.2 ports: - containerPort: 3000 resources: requests: cpu: "250m" memory: "256Mi" limits: cpu: "500m" memory: "512Mi" readinessProbe: httpGet: path: /health port: 3000 initialDelaySeconds: 10 periodSeconds: 5 livenessProbe: httpGet: path: /health port: 3000 initialDelaySeconds: 30 periodSeconds: 10 env: - name: AWS_REGION value: us-east-1 - name: S3_BUCKET valueFrom: configMapKeyRef: name: api-config key: s3_bucket
```

3. Horizontal Pod Autoscaler (HPA)

```
apiVersion: autoscaling/v2 kind: HorizontalPodAutoscaler metadata: name: api-server-hpa namespace: production spec: scaleTargetRef: apiVersion: apps/v1 kind: Deployment name: api-server minReplicas: 3 maxReplicas: 50 metrics: - type: Resource resource: name: cpu target: type: Utilization averageUtilization: 70 - type: Resource resource: name: memory target: type: Utilization averageUtilization: 80 behavior: scaleUp: stabilizationWindowSeconds: 30 policies: - type: Pods value: 5 periodSeconds: 60 scaleDown: stabilizationWindowSeconds: 300 # 5 min cooldown before scale-in
```

4. ALB Ingress with TLS

```
apiVersion: networking.k8s.io/v1 kind: Ingress metadata: name: production-ingress namespace: production annotations: kubernetes.io/ingress.class: alb alb.ingress.kubernetes.io/scheme: internet-facing alb.ingress.kubernetes.io/target-type: ip alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:123:certificate/abc alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS":443},{"HTTP":80}]' alb.ingress.kubernetes.io/ssl-redirect: "443" alb.ingress.kubernetes.io/healthcheck-path: /health alb.ingress.kubernetes.io/wafv2-acl-arn: arn:aws:wafv2:us-east-1:123:regional/webacl/prod spec: rules: - host: app.company.com http: paths: - path: /api pathType: Prefix backend: service: name: api-server-svc port: number: 3000 - path: / pathType: Prefix backend: service: name: frontend-svc port: number: 80
```

5. PodDisruptionBudget (Critical for Zero-Downtime Upgrades)

```
apiVersion: policy/v1 kind: PodDisruptionBudget metadata: name: api-server-pdb namespace: production spec: minAvailable: 2 # At least 2 pods must be up during node drain selector: matchLabels: app:
```

```
api-server # This ensures node drains during upgrades don't take all api-server pods offline #
Karpenter and Node Group rolling updates both respect PDBs
```

■ SOP — Standard Operating Procedure

	Cluster Naming Convention	Pattern: {env}-{region}-eks-{version} e.g. prod-use1-eks-130. Tag: Environment, Team, Co
SOP-02	**Access Control**	Never use cluster-admin for apps. Use RBAC: ClusterRole/RoleBinding per team. Enable A
SOP-03	**Network Policy**	Enable NetworkPolicy (Calico or VPC CNI). Default-deny all. Allow explicitly. Isolate names
SOP-04	**Image Security**	Pull only from ECR. Enable ECR image scanning on push. Enforce via OPA Gatekeeper o
SOP-05	**Resource Requests/Limits**	All pods must have CPU/memory requests and limits defined. LimitRange in each namespa
SOP-06	**PodDisruptionBudgets**	All production Deployments must have PDB with minAvailable >= 2. Required for zero-dow
SOP-07	**Upgrade Cadence**	Track EKS version release calendar. Upgrade within 30 days of new release. Never run en
SOP-08	**Upgrade Staging First**	All upgrades must be tested on staging cluster (same config) 1 week before production. Do
SOP-09	**Add-on Version Matrix**	Maintain a spreadsheet: cluster version ↔ compatible add-on versions. Update after every
SOP-10	**Deprecated API Check**	Run 'pluto detect-all-in-cluster' before every upgrade. Fix all deprecated API usage before
SOP-11	**Node Drain Safety**	Never drain more than 1 node at a time manually. Verify PDBs respected. Drain timeout: 5
SOP-12	**Rollback Plan**	Control plane upgrade is irreversible. Node groups can be reverted (AMI rollback). Docume
SOP-13	**Monitoring During Upgrade**	Watch: kubectl get nodes -w, kubectl get pods -A -w, CloudWatch Container Insights during
SOP-14	**Post-Upgrade Smoke Test**	Run automated integration tests: API health, DB connectivity, ingress routing, HPA trigger,
SOP-15	**Cost Audit**	Monthly: review EC2 instance utilization via CloudWatch. Enable Spot where possible. Del

■ Cost Analysis (us-east-1, 2025 Pricing)

EKS costs include: cluster fee, EC2 worker nodes (or Fargate), load balancers, data transfer, EBS volumes, and CloudWatch. Below are 3 deployment tier estimates.

Component	Unit Price	Dev/Test	Production	Enterprise
EKS Cluster Fee	$0.10/hr	~$72/mo	~$72/mo	~$144/mo (2 clusters)
Worker Nodes (m5.xlarge On-Demand)	$0.192/hr	2 nodes ~$277/mo	6 nodes ~$831/mo	20 nodes ~$2,765/mo
Worker Nodes (Spot — ~70% saving)	~$0.058/hr	~$83/mo	~$249/mo	~$830/mo
Fargate (0.25 vCPU, 0.5 GB)	$0.04048/vCPU-hr	~$10/mo	~$100/mo	~$500/mo
ALB (Application Load Balancer)	$0.008/LCU-hr + $16.43 base	~$20/mo	~$60/mo	~$200/mo
EBS Volumes (gp3, 100GB/node)	$0.08/GB-mo	~$16/mo	~$50/mo	~$200/mo
CloudWatch Logs (5 GB/mo)	$0.50/GB ingested	~$5/mo	~$25/mo	~$100/mo
NAT Gateway	$0.045/hr + $0.045/GB	~$33/mo	~$60/mo	~$150/mo
Data Transfer (egress)	$0.09/GB	~$5/mo	~$50/mo	~$300/mo
TOTAL (On-Demand)		**~$430/mo**	**~$1,200/mo**	**~$4,360/mo**
TOTAL (With Spot Nodes)		**~$240/mo**	**~$620/mo**	**~$2,420/mo**

■ Cost Tips: (1) Use Spot Instances via Karpenter — save 60-70% on EC2. (2) Enable Karpenter consolidation to bin-pack and remove underutilized nodes. (3) Use Savings Plans for baseline On-Demand capacity. (4) Set resource requests accurately — oversizing wastes money on underutilized nodes.

■ Interview Q&A; — EKS Deep Dive

Q1: What is the difference between EKS Managed Node Groups and Karpenter?

Managed Node Groups use Auto Scaling Groups — you pre-define instance types and let ASG scale. Karpenter is a newer node autoscaler that watches unschedulable pods, picks the optimal instance type (right-sized, Spot or On-Demand), and provisions it in < 60 seconds — much faster than ASG (which takes 3–5 minutes). Karpenter also consolidates underutilized nodes automatically.

Q2: What is IRSA and why is it better than EC2 instance profiles for pods?

IRSA (IAM Roles for Service Accounts) maps a Kubernetes ServiceAccount to an AWS IAM Role via OIDC federation. Each pod gets a temporary, auto-rotating token mounted as a projected volume. Instance profiles grant IAM permissions to the entire EC2 node — any pod can assume those permissions. IRSA gives per-pod granular IAM — the principle of least privilege at the pod level, with no static credentials.

Q3: Walk me through a zero-downtime EKS upgrade.

Pre-flight: check for deprecated APIs (pluto), verify all pods Running, ensure PDBs are set. Order: (1) Control Plane — AWS managed, < 20 min, zero downtime. (2) Add-ons — update vpc-cni, CoreDNS, kube-proxy, CSI drivers individually. (3) Node Groups — rolling update: AWS cordons/drains one node at a time; PDBs protect availability. (4) Fargate — restart deployments to get new runtime. (5) Validate: all nodes correct version, all pods healthy, smoke tests pass.

Q4: How does EKS handle etcd? Do I need to back it up?

AWS fully manages etcd in the EKS control plane. You cannot access etcd directly. AWS runs etcd across 3 AZs with automatic backups. You do not need to back up etcd yourself. However, you should export your Kubernetes manifests (Deployments, Services, ConfigMaps, Secrets) to version control (GitOps with ArgoCD/Flux) as your source of truth — not etcd.

Q5: What is the VPC CNI and what is 'Security Groups for Pods'?

VPC CNI (Container Network Interface) assigns each pod a real VPC IP address from the node's ENI. This means pods are first-class VPC citizens — they can be targeted by security groups, route tables, and VPC flow logs. Security Groups for Pods (v1.11+) extends this: you can attach a specific EC2 Security Group to individual pods, enabling fine-grained network access control at the pod level, not just node level.

Q6: How would you handle a failed EKS upgrade?

Control plane upgrades cannot be rolled back — once upgraded, it stays. This is why staging validation is essential. For node groups: terminate new nodes and launch old AMI nodes (rollback via launch template version). For add-ons: downgrade to previous version via eksctl update addon --addon-version . Mitigation: use canary/blue-green deployments for apps, PDBs to protect availability during rollout.

Q7: How does Cluster Autoscaler differ from Karpenter?

Cluster Autoscaler (CAS) scales existing Auto Scaling Groups — limited to instance types pre-defined in the ASG. It reacts slowly (3–5 min) due to ASG + node join time. It cannot bin-pack or consolidate efficiently. Karpenter watches pending pods directly, selects the ideal instance type on the fly, and provisions in < 60s. Karpenter's consolidation mode also removes underutilized nodes, saving cost automatically.

Q8: What are EKS Add-ons and why should I use them?

EKS Add-ons are managed versions of core Kubernetes operational software: vpc-cni, CoreDNS, kube-proxy, EBS CSI Driver, EFS CSI Driver, CloudWatch Observability, etc. Benefits: AWS tests compatibility with each K8s version, handles upgrades automatically (or on your command), and prevents configuration drift. Without managed add-ons, you manage Helm charts/DaemonSets yourself and risk version incompatibilities after cluster upgrades.

■■ EKS vs Alternatives

Criteria	EKS	ECS	EKS Fargate	kOps / Self-Managed
Control Plane	**AWS Managed**	AWS Managed	AWS Managed	You Manage
Orchestrator	**Kubernetes**	AWS ECS proprietary	Kubernetes	Kubernetes

Portability	**High (K8s standard)**	Low (ECS-specific)	Medium	High
Ops Complexity	**Medium**	Low	Low	Very High
Node Management	**Managed NG / Karpenter**	Auto (EC2 or Fargate)	Serverless	Manual
GPU Support	**Yes**	Yes	No	Yes
Helm / GitOps	**Yes (ArgoCD, Flux)**	Limited (CDK, Copilot)	Yes	Yes
Best For	**K8s-native, multi-cloud apps**	Simple AWS-only containers	Serverless K8s pods	Full K8s control on-prem/cloud
Cost (cluster fee)	**$0.10/hr**	Free control plane	$0.10/hr	EC2 cost only

Quick Reference Summary

Attribute	Details
Service Type	Fully Managed Kubernetes Control Plane + Multiple Data Plane Options
Control Plane HA	Runs across 3 AZs automatically — API server, etcd, scheduler
Node Options	Managed Node Groups, Fargate (serverless), Self-Managed EC2
Kubernetes Versions	Supports N-3 minor versions. Must upgrade one minor version at a time
Upgrade Order	Control Plane → Add-ons → Node Groups → Fargate → App Manifests
Control Plane Upgrade Time	~15–20 minutes, zero downtime (AWS managed)
Node Group Upgrade	Rolling update: one node at a time, respects PodDisruptionBudgets
Networking	VPC CNI: each pod gets real VPC IP. Security Groups per pod supported
IAM for Pods	IRSA: ServiceAccount → IAM Role via OIDC. No static credentials in pods
Autoscaling	HPA (pod scaling), Karpenter (node provisioning < 60s), KEDA (event-driven)
Load Balancing	AWS Load Balancer Controller: ALB (HTTP/S, path routing) / NLB (TCP/UDP)
Add-ons Managed	vpc-cni, CoreDNS, kube-proxy, EBS CSI, EFS CSI, CloudWatch Observability
Security	RBAC, NetworkPolicy, PodSecurity, OPA/Kyverno, Secrets Store CSI, KMS etcd encrypt
Monitoring	CloudWatch Container Insights, Prometheus, Grafana, Jaeger/X-Ray tracing
Dev/Test Cost	~$240/month (cluster + 2 Spot nodes + ALB + monitoring)
Production Cost	~$620/month (cluster + 6 Spot nodes + ALB + EBS + CloudWatch)
Enterprise Cost	~$2,420/month (multi-cluster + 20 Spot nodes + full observability stack)
Key Tools	eksctl, kubectl, helm, karpenter, argocd, pluto (deprecated API checker), kubecost