

AWS Production — EknathaLabs Interview Guide

Platform, SRE & DevOps interview guide

Questions	170 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/aws_interview_guide.html
Printed	2026-10-21

Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

Q1 What is the IAM model in AWS — how do principals, policies, and roles work?

IAM & Security

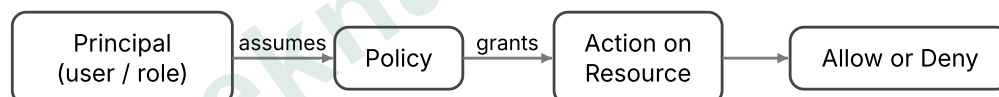
🗣 Spoken answer · ~77 s

IAM controls who can do what on which AWS resources. Principals are the actors: IAM users, roles, federated identities, and AWS services. Policies are JSON documents with statements of effect, actions, resources, and optional conditions. Identity-based policies attach to users, groups, or roles; resource-based policies attach to resources such as S3 buckets or KMS keys. Roles are identities without long-term credentials: a principal assumes a role through STS and receives temporary credentials. A role has two parts: a trust policy saying who may assume it, and permission policies saying what it can do. The evaluation default is implicit deny; an explicit allow is needed, and an explicit deny always wins. In production I avoid IAM users with access keys, use roles for workloads, federate human access through IAM Identity Center, and apply least privilege, starting from what is needed and refining using Access Analyzer and last-accessed data. I use permission boundaries and service control policies to cap what can be granted, require MFA for sensitive actions, and keep the root account locked down. Everything is managed as code, so changes are reviewed and auditable.

Likely follow-ups

- What is the difference between a trust policy and a permission policy?
- How do you move toward least privilege over time?

🗺 IAM model



Principal types: IAM User, IAM Role, AWS Service, Federated identity

Policy types: Identity-based, Resource-based, SCP (org-level), Permission boundary

Create role for EC2 to access S3:

```
aws iam create-role --role-name ec2-s3-role --assume-role-policy-document file://trust.json
```

```
aws iam attach-role-policy --role-name ec2-s3-role --policy-arn  
arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
```

Assume role for cross-account access:

```
aws sts assume-role --role-arn arn:aws:iam::123456789:role/CrossAccountRole --role-session-  
name session1
```

Least privilege principle: grant only the permissions needed for the task. IAM Roles are preferred over IAM Users for services — roles use temporary credentials via STS. SCPs (Service Control Policies) set maximum permissions for an entire AWS Organization account — they override IAM policies. Permission boundaries limit the maximum permissions an IAM entity can have, even if their policies allow more.

IAM Access Analyzer: Automatically identifies resource policies granting external access. Run regularly to find unintended public S3 buckets, overly permissive role trust policies, and cross-account access. Essential for security posture management.

Official docs

- [AWS IAM: Roles](#)
- [Amazon S3 User Guide](#)

Q2 What is AWS STS and how does it work — how do you implement cross-account access?

IAM & Security

Spoken answer · ~75 s

STS, the Security Token Service, issues temporary, limited-privilege credentials. The most common call is AssumeRole: a principal presents its identity, STS checks the role's trust policy, and returns an access key, secret key, and session token that expire, by default after an hour. For cross-account access, there are two sides. In the target account, create a role whose trust policy names the source account or specific role as a trusted principal, with permission policies for what it may do. In the source account, grant the calling identity permission to call sts:AssumeRole on that role's ARN. The caller then assumes the role and works in the target account with the role's permissions. For third parties, add an external ID condition to prevent the confused-deputy problem, and require MFA where appropriate. Related calls are AssumeRoleWithWebIdentity, used by EKS service accounts and CI systems through OIDC, and AssumeRoleWithSAML for enterprise federation. Benefits are no long-lived keys, automatic expiry, and a clear audit trail in CloudTrail showing the session name. I keep session durations short, scope trust policies tightly, and use session tags for attribution.

Likely follow-ups

- What is the confused-deputy problem and how does an external ID help?
- How does AssumeRoleWithWebIdentity avoid static keys in CI?

```
# Cross-account role assumption:
```

```
aws sts assume-role --role-arn arn:aws:iam::ACCOUNT-B:role/DeployRole --role-session-name deploy
```

```
# Returns temporary credentials (expire in 1-12 hours):
```

```
export AWS_ACCESS_KEY_ID=... AWS_SECRET_ACCESS_KEY=... AWS_SESSION_TOKEN=...
```

```
# Trust policy on target account role:
```

```
{"Principal": {"AWS": "arn:aws:iam::ACCOUNT-A:role/CIRole"}, "Action": "sts:AssumeRole"}
```

```
# OIDC federation for GitHub Actions:
```

```
aws iam create-open-id-connect-provider --url https://token.actions.githubusercontent.com
```

STS issues temporary credentials for role assumption. Cross-account pattern: Account A has a role, Account B has a role with trust policy allowing Account A to assume it. Account A's CI pipeline assumes Account B's DeployRole to deploy resources there. No long-lived credentials needed. OIDC federation for GitHub Actions: GitHub OIDC token exchanges for AWS credentials — no IAM User access keys in CI.

Official docs

- [AWS IAM: Roles](#)
- [AWS IAM: Temporary credentials \(STS\)](#)

Q3 What is a VPC — how do you design a production-grade VPC architecture?

Networking

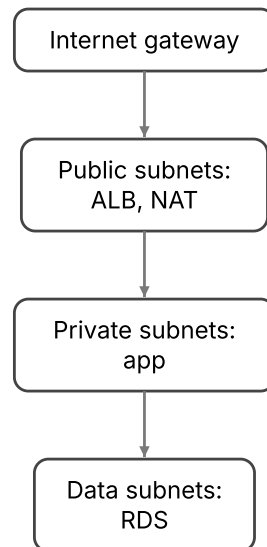
Spoken answer · ~78 s

A VPC is your logically isolated virtual network in AWS, defined by a CIDR block, subnets, route tables, and gateways. A production design starts with address planning: choose a CIDR large enough and non-overlapping with other VPCs and on-premises networks, because overlap blocks peering and VPN later. Span at least three availability zones. In each zone, create public subnets for load balancers and NAT gateways, private subnets for application workloads, and isolated data subnets with no route to the internet for databases. Public subnets route to an internet gateway; private subnets route outbound through a NAT gateway, ideally one per zone for resilience. Add gateway endpoints for S3 and DynamoDB, and interface endpoints for services like ECR and Secrets Manager, to keep traffic off the internet and cut NAT cost. Control traffic with security groups as the main tool and NACLs as a coarse backstop. Turn on VPC flow logs for visibility. Connect to other networks with Transit Gateway or peering, and to on-premises with Direct Connect or VPN. For EKS, size subnets for pod IPs. Define everything as code and use separate VPCs or accounts per environment.

Likely follow-ups

- Why plan CIDR ranges before building the VPC?
- What do VPC endpoints save you?

🚧 Production VPC



```
aws ec2 create-vpc --cidr-block 10.0.0.0/16
```

```
# 3-tier subnet layout per AZ:
```

```
Public subnets: 10.0.0.0/24, 10.0.1.0/24 -- ALB, NAT Gateway
```

```
Private subnets: 10.0.10.0/24, 10.0.11.0/24 -- App servers, EKS nodes
```

```
DB subnets: 10.0.20.0/24, 10.0.21.0/24 -- RDS, ElastiCache
```

```
# NAT Gateway (private subnet internet outbound):
```

```
aws ec2 create-nat-gateway --subnet-id public-subnet-id --allocation-id eip-id
```

```
# VPC Flow Logs for security audit:
```

```
aws ec2 create-flow-logs --resource-type VPC --resource-ids vpc-id --traffic-type ALL --log-destination-type cloud-watch-logs
```

3-tier VPC: public subnets for load balancers and NAT Gateways, private subnets for application workloads, isolated subnets for databases. Resources in private subnets cannot be reached from the internet but can reach out via NAT Gateway. Security Groups act as stateful firewalls per resource. NACLs act as stateless firewalls per subnet. Always deploy across at least 2 AZs for high availability.

📖 Official docs

- [Amazon VPC: What is Amazon VPC?](#)

 Spoken answer · ~82 s

Both filter traffic, but they differ in level and behaviour. A security group is attached to an elastic network interface, so it works at the instance or resource level. It is stateful: if inbound traffic is allowed, the response is automatically allowed regardless of outbound rules. It has only allow rules, everything not allowed is denied, and all rules are evaluated together. Rules can reference other security groups, which is powerful for tiered architectures, for example allowing the app tier's group to reach the database group. A network ACL is attached to a subnet and applies to all traffic crossing the subnet boundary. It is stateless, so you must explicitly allow return traffic, including ephemeral ports. It supports both allow and deny rules, evaluated in numbered order, with the first match winning. In practice, security groups are the primary control and handle nearly everything. I use NACLs sparingly, for example to block a specific malicious IP range quickly, or as a coarse additional layer for compliance, since explicit deny is something security groups can't do. Troubleshooting tip: if traffic works one way but responses fail, suspect a NACL; if nothing connects, check security groups and routes first.

Likely follow-ups

- Why does stateless filtering need ephemeral port rules?
- When would you actually use a NACL?

```
# Security Group (stateful - per instance/ENI):
```

```
aws ec2 authorize-security-group-ingress --group-id sg-xxx --protocol tcp --port 443 --cidr 0.0.0.0/0
```

```
aws ec2 describe-security-groups --group-ids sg-xxx
```

```
# NACL (stateless - per subnet):
```

```
aws ec2 create-network-acl-entry --network-acl-id acl-xxx --rule-number 100 --protocol tcp --port-range From=443,To=443 --cidr-block 0.0.0.0/0 --rule-action allow
```

```
# NACL needs explicit DENY for outbound 443 response too (stateless)
```

Security Groups: stateful (return traffic automatically allowed), attached to instances/ENIs, allow-only rules (no explicit deny). NACLs: stateless (must explicitly allow return traffic), subnet-level, allow and deny rules, numbered rules evaluated in order. Use Security Groups for most cases (simpler). Use NACLs as an additional layer for subnet-level blocking (block specific CIDR ranges, add defence-in-depth).

 Official docs

- [Amazon VPC: Security groups](#)
- [Amazon VPC: Network ACLs](#)

Q5 **What is EC2 — how do you choose the right instance type and purchase model?**

Compute

🔊 Spoken answer · ~77 s

EC2 provides virtual servers. Choosing an instance starts with the workload profile. General purpose families, like M, balance CPU and memory; compute-optimised C suits CPU-heavy work; memory-optimised R or X suits databases and caches; storage-optimised I suits high disk throughput; and accelerated P or G families provide GPUs. Newer generations and Graviton ARM instances usually give better price-performance if your software supports ARM. I pick a starting size, then right-size using CloudWatch and Compute Optimizer data. Purchase models: on-demand is flexible with no commitment; Savings Plans or reserved instances give up to roughly seventy percent discount for one- or three-year commitments on steady baseline usage; spot instances give up to about ninety percent off for fault-tolerant workloads that handle interruption with two minutes' notice, such as batch, CI, and stateless workers; dedicated hosts address licensing or compliance. A typical production mix is a committed baseline for steady load, on-demand for variable load, and spot for tolerant capacity, diversified across instance types and zones. Also consider network bandwidth, EBS optimisation, and burstable T instances for light workloads, where credit exhaustion is a risk. Measure, don't guess.

Likely follow-ups

- When do spot instances make sense?
- How do you right-size an instance?

Instance families:

t4g/t3 – burstable general purpose (dev/test)

m7i/m6i – balanced compute/memory (web servers)

c7i/c6i – compute optimised (CPU-intensive workloads)

r7i/r6i – memory optimised (databases, caches)

p4/p3 – GPU instances (ML training)

Purchase models:

On-Demand: pay per hour, no commitment, full price

Reserved: 1-3 year commitment, up to 72% discount

Savings Plans: flexible commitment (\$/hour), up to 66% discount

Spot: spare capacity, up to 90% discount, can be interrupted

Right-size first, then optimise purchase model. Graviton (arm64) instances (t4g, m7g, c7g) offer 20-40% better price/performance for most workloads. Spot instances are ideal for fault-tolerant batch jobs, CI runners, and stateless auto-scaled workloads. Use Compute Savings Plans for baseline capacity (predictable load), Spot for peak/burst. AWS Cost Explorer Recommendations identifies right-sizing opportunities automatically.



Official docs

- [Amazon EC2 User Guide](#)

Q6 What is EKS — how does it differ from self-managed Kubernetes?

Containers

Spoken answer · ~75 s

EKS is AWS's managed Kubernetes service. The key difference from self-managed Kubernetes is that AWS runs and maintains the control plane: the API servers and etcd are replicated across multiple availability zones, patched, backed up, and scaled for you. In a self-managed cluster you operate etcd, control-plane upgrades, certificates, and high availability yourself, which is significant ongoing work. EKS integrates with AWS: IAM for authentication, IAM roles for service accounts or Pod Identity for pod permissions, VPC CNI giving pods real VPC IP addresses, load balancers through the AWS Load Balancer Controller, and CloudWatch and CloudTrail for logs. You still own the data plane decisions: worker nodes through managed node groups, self-managed nodes, Karpenter, or Fargate; add-ons; networking; security; and application upgrades. The control plane costs a fixed hourly fee per cluster. Upgrades are simpler but still require care: you upgrade the control plane, then add-ons and nodes, and must watch for API deprecations. Self-managed gives total control, such as custom control-plane flags or unusual environments, but most teams choose EKS to focus on workloads rather than operating Kubernetes itself.

Likely follow-ups

- What does AWS manage and what do you still own in EKS?
- How do pods get IAM permissions on EKS?

```
eksctl create cluster --name prod --region us-east-1 --nodes 3 --node-type m6i.large
```

```
aws eks update-kubeconfig --region us-east-1 --name prod
```

```
# EKS managed node groups:
```

```
eksctl create nodegroup --cluster prod --name workers --node-type m6i.large --nodes-min 2 --nodes-max 10
```

```
# Fargate profile (serverless pods):
```

```
eksctl create fargateprofile --cluster prod --name fp-default --namespace default
```

```
aws eks update-cluster-version --name prod --kubernetes-version 1.30
```

EKS: AWS manages the control plane (etcd, API server, scheduler) — you manage the data plane (nodes). Managed node groups handle node provisioning, AMI updates, and draining during upgrades. Fargate removes node management entirely — AWS runs each pod on isolated compute. Key EKS add-ons: aws-load-balancer-controller (ALB integration), EBS CSI driver, EFS CSI driver, VPC CNI. EKS Anywhere extends EKS to on-prem. Cluster upgrades: update control plane first, then node groups.

Official docs

- [Amazon EKS: User guide](#)

Q7 What is the AWS Load Balancer Controller — how do you expose Kubernetes services on EKS?

Containers

Spoken answer · ~84 s

The AWS Load Balancer Controller runs in the cluster and watches Kubernetes resources, creating AWS load balancers on your behalf. For an Ingress resource, it provisions an Application Load Balancer, using annotations to configure scheme, certificates from ACM, health checks, and routing rules. For a Service of type LoadBalancer with the right annotations, it provisions a Network Load Balancer. The preferred target mode is IP, where the load balancer sends traffic directly to pod IPs, which works well with the VPC CNI, avoids an extra hop through node ports, and gives accurate health checks and even distribution. Setup: install the controller with Helm, give it an IAM role through IRSA or Pod Identity with the required permissions, and make sure subnets carry the right tags so it can discover where to place load balancers. Then create an Ingress with the ingress class set to alb, plus host or path rules. Operational points: use readiness gates or pod readiness to ensure pods are registered and healthy before old ones terminate, which avoids dropped connections during rollouts; add a preStop sleep and graceful shutdown; share one ALB across many Ingresses with ingress groups to control cost; and attach WAF or security groups as needed.

Likely follow-ups

- Why prefer IP target mode over instance mode?
- How do you avoid dropped requests during rollouts behind an ALB?

```
# Install ALB controller:
```

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller -n kube-system \
  --set clusterName=prod --set serviceAccountName=aws-load-balancer-controller
```

```
# Ingress with ALB:
```

```
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:xxx:certificate/xxx
```

```
# NLB for TCP/UDP:
```

```
service.beta.kubernetes.io/aws-load-balancer-type: "external"
```

ALB controller provisions AWS ALBs/NLBs from Ingress and Service resources. `target-type: ip` routes directly to pod IPs (bypasses NodePort, more efficient). Use NLB for TCP/UDP or when you need static IPs. ALB supports path-based and host-based routing, HTTPS termination with ACM certificates, WAF integration, and authentication via Cognito/OIDC. One ALB per Ingress by default — use `IngressGroup` annotation to share an ALB across multiple Ingress resources (reduces cost).



Official docs

- [Cloudflare Learning: Load balancing](#)
- [AWS Load Balancer Controller](#)
- [Elastic Load Balancing](#)

Q8 What are the storage options in AWS — EBS, EFS, S3 — when do you use each?

Storage

Spoken answer · ~80 s

The three solve different storage problems. EBS is block storage attached to a single EC2 instance in one availability zone, behaving like a disk. Use it for boot volumes, databases, and anything needing low-latency block access. Choose `gp3` for general purpose, `io2` for high, consistent IOPS, and snapshots for backup. It is not shared across zones, though multi-attach exists for specific io volumes. EFS is a managed NFS file system that many instances or pods can mount simultaneously across zones. Use it for shared content, home directories, and workloads needing a shared POSIX file system. It costs more per gigabyte and has higher latency than EBS. S3 is object storage, accessed over an API, practically unlimited, highly durable, and cheap, with storage classes and lifecycle rules. Use it for backups, logs, static assets, data lakes, and artefacts. It is not a file system, so applications must use the object API. A simple rule: one server needing a fast disk, use EBS; many servers sharing files, use EFS; large-scale, durable, unstructured data accessed by API, use S3. For Kubernetes, EBS volumes back single-pod persistent volumes through the EBS CSI driver, and EFS supports `ReadWriteMany`.

Likely follow-ups

- Why can't an EBS volume be used across availability zones?
- When would you choose EFS over S3?

EBS: block storage, single AZ, attached to one instance, high IOPS

gp3: general purpose SSD, 3000 IOPS baseline, up to 16000 IOPS

io2 Block Express: high performance DB (up to 256,000 IOPS)

EFS: NFS shared filesystem, multi-AZ, auto-scales, multiple instances

Use for: shared config, user uploads accessible from many pods

S3: object storage, unlimited scale, 11 nines durability

Use for: backups, static assets, data lake, ML training data

```
aws s3 cp largefile.tar.gz s3://mybucket/ --storage-class INTELLIGENT_TIERING
```

EBS: databases (MySQL, PostgreSQL, MongoDB) — single-instance block storage with consistent low latency. EFS: shared storage for multiple EC2 instances or Kubernetes pods (ReadWriteMany PVC via EFS CSI driver). S3: everything else at scale — virtually unlimited, 99.999999999% durability, versioning, lifecycle policies, event notifications. S3 Intelligent-Tiering automatically moves objects between access tiers based on usage patterns — significant cost savings for data with unpredictable access.

Official docs

- [Amazon EBS](#)
- [Amazon EFS](#)
- [Amazon S3 User Guide](#)

Spoken answer · ~76 s

For production PostgreSQL on RDS, I make availability, security, recovery, and observability explicit. Deployment: a Multi-AZ instance, or a Multi-AZ cluster, so a standby in another zone takes over automatically on failure, and Aurora PostgreSQL if I want faster failover and storage that scales automatically. Place it in private data subnets with a DB subnet group, with no public access, and a security group allowing only the application tier. Encrypt at rest with KMS and enforce TLS in transit. Backups: enable automated backups with point-in-time recovery, set sensible retention, take manual snapshots before major changes, and periodically test restores. Scaling: choose an instance class by measured need, use read replicas for read-heavy workloads, and front connections with RDS Proxy or a pooler to prevent connection storms, especially from Lambda or many pods. Parameter groups hold tuned settings. Credentials: store them in Secrets Manager with rotation, or use IAM database authentication. Monitoring: enable Performance Insights, Enhanced Monitoring, and CloudWatch alarms on CPU, free storage, connections, replica lag, and IOPS. Plan maintenance windows and minor-version upgrades, and enable deletion protection. Define it all in Terraform.

Likely follow-ups

- What does RDS Proxy solve?
- How do you verify that your backups actually work?

```
aws rds create-db-instance --db-instance-identifier mydb \
```

```
--db-instance-class db.r6g.large \
```

```
--engine postgres --engine-version 16 \
```

```
--master-username admin --manage-master-user-password \
```

```
--vpc-security-group-ids sg-xxx --db-subnet-group mysubnetgroup \
```

```
--multi-az --storage-encrypted --allocated-storage 100
```

```
# Read replica:
```

```
aws rds create-db-instance-read-replica --source-db-instance-identifier mydb --db-instance-identifier mydb-read
```

```
# RDS Proxy (connection pooling):
```

```
aws rds create-db-proxy --db-proxy-name myproxy --engine-family POSTGRES
```

Production RDS: Multi-AZ (synchronous replication to standby, auto-failover in 60-120 seconds), encryption at rest and in transit, Private subnet only (no public access), automated backups (7-35 day retention), Performance Insights enabled. RDS Proxy: connection pooling reduces connection overhead for Lambda or

Fargate workloads. --manage-master-user-password stores credentials in Secrets Manager automatically — no password in IaC code.

Official docs

- [Amazon RDS User Guide](#)

Q10 What is CloudWatch — how do you use it for production monitoring?

Monitoring

Spoken answer · ~78 s

CloudWatch is AWS's monitoring service for metrics, logs, alarms, and dashboards. Metrics are time series; AWS services publish defaults, such as EC2 CPU or ALB request counts, and applications can publish custom metrics. Logs are collected into log groups, from the CloudWatch agent, Lambda, ECS, or EKS via Fluent Bit, with Logs Insights for querying and metric filters to turn log patterns into metrics. For production I build monitoring around user-facing signals. Alarms on the golden signals: ALB 5xx rate and latency, target health, error rates, queue depth, and saturation such as CPU, memory, disk, and database connections. I use composite alarms to reduce noise, anomaly detection for variable baselines, and route notifications through SNS to paging and chat tools, with runbook links. Dashboards give a service-level view per application. Container Insights covers EKS and ECS metrics. Add synthetic canaries to test critical journeys from outside, and Application Signals or X-Ray for tracing. Control cost by setting log retention, avoiding high-cardinality custom metrics, and sampling noisy logs. Define alarms and dashboards as code so they're consistent across environments. Combine CloudWatch with SLOs so alerts reflect real user impact.

Likely follow-ups

- What are composite alarms for?
- How do you control CloudWatch costs?

```
# Custom metric:
```

```
aws cloudwatch put-metric-data --namespace MyApp --metric-name ErrorCount --value 5 --unit Count
```

```
# Alarm:
```

```
aws cloudwatch put-metric-alarm --alarm-name HighErrorRate \
```

```
--metric-name ErrorCount --namespace MyApp \
```

```
--statistic Sum --period 60 --threshold 10 \
```

```
--comparison-operator GreaterThanThreshold \
```

```
--alarm-actions arn:aws:sns:us-east-1:xxx:alerts
```

```
# Container Insights (EKS/ECS metrics):
```

```
aws eks update-addon --cluster-name prod --addon-name amazon-cloudwatch-observability
```

```
# Log Insights query:
```

```
fields @timestamp, @message | filter @message like "ERROR" | stats count(*) by bin(5m)
```

CloudWatch covers metrics (custom + AWS service metrics), logs (CloudWatch Logs), alarms (SNS/PagerDuty integration), dashboards, and Logs Insights (query language for structured logs). Container Insights adds K8s-aware metrics (per pod, per namespace) to EKS. For complex observability: Managed Prometheus + Grafana is more powerful than CloudWatch dashboards, but CloudWatch is zero-setup for AWS services.

Official docs

- [Amazon CloudWatch User Guide](#)

Q11 How do you set up AWS X-Ray for distributed tracing?

Monitoring

 Spoken answer · ~81 s

X-Ray gives distributed tracing: it follows a request across services, showing the path, timing of each segment, and where errors occur. Setup has three parts. First, instrument the application, using the AWS Distro for OpenTelemetry or the X-Ray SDK, which creates segments and subsegments for incoming requests and outgoing calls to AWS services, HTTP endpoints, and databases. Second, run a collector: the X-Ray daemon or the ADOT collector, as a sidecar, a DaemonSet in EKS, or built into the platform for Lambda and App Runner, receives the trace data and sends it to the X-Ray service. Third, grant IAM permissions to send traces, via the task role or IRSA. Propagate the trace header between services so segments are stitched into one trace; this is usually handled automatically by the instrumentation for HTTP clients. Configure sampling rules to control cost and volume, for example a low rate for healthy traffic but higher for errors. Then use the service map to see dependencies and hotspots, and the trace view to find slow segments. Add annotations for searchable fields, such as customer tier or build version. Increasingly teams use OpenTelemetry so the instrumentation isn't tied to one backend.

Likely follow-ups

- What do sampling rules control?
- Why use OpenTelemetry instead of vendor-specific SDKs?

```
# Python instrumentation:
```

```
from aws_xray_sdk.core import xray_recorder, patch_all
```

```
patch_all() # auto-patches boto3, requests, mysql, etc.
```

```
xray_recorder.configure(service="payments-service")
```

```
# EKS: X-Ray daemon as DaemonSet:
```

```
kubectl apply -f https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-container-insights/latest/k8s-deployment-manifest-templates/deployment-mode/daemonset/container-insights-monitoring/x-ray/xray-k8s-daemonset.yaml
```

```
# OR: use OpenTelemetry → AWS Distro for OpenTelemetry (ADOT)
```

```
OTEL_EXPORTER_OTLP_ENDPOINT=http://aws-otel-collector:4317
```

X-Ray traces requests across Lambda, EC2, ECS, EKS, and API Gateway automatically with minimal SDK integration. Use AWS Distro for OpenTelemetry (ADOT) for standards-based tracing that works with both X-Ray and other backends — avoids vendor lock-in. X-Ray Service Map visualises service dependencies and latency between them automatically.

Official docs

- [AWS X-Ray](#)
- [OpenTelemetry: Signals \(traces, metrics, logs\)](#)

Q12 How do you reduce AWS costs — what are the key levers?

Cost Opt

Spoken answer · ~81 s

The main levers, in order of typical impact. First, visibility: tag resources, use Cost Explorer and Cost and Usage Reports, and set budgets and anomaly alerts so you know where money goes. Second, eliminate waste: shut down idle and orphaned resources, such as unattached volumes, old snapshots, unused load balancers and elastic IPs, and run non-production environments only during working hours. Third, right-size: use utilisation data and Compute Optimizer to move over-provisioned instances and databases to smaller sizes. Fourth, commitments: cover the steady baseline with Savings Plans or reserved instances, which can save around thirty to seventy percent. Fifth, use spot instances for fault-tolerant workloads, and Graviton for better price-performance. Sixth, storage: apply S3 lifecycle policies and Intelligent-Tiering, delete old data, migrate gp2 to gp3, and cap log retention. Seventh, data transfer, often a hidden cost: keep traffic within a zone where possible, use VPC endpoints and a CDN, and avoid cross-region transfers. Eighth, architecture: autoscale to demand, use serverless for spiky workloads, and cache. Then govern it: assign cost ownership per team, review monthly, and track unit costs such as cost per request. Changes should be made safely, with measurement before and after.

Likely follow-ups

- Why is data transfer a hidden cost?
- How do commitments like Savings Plans work?

```
# Right-sizing (biggest lever):
```

```
aws compute-optimizer get-ec2-instance-recommendations --region us-east-1
```

Commit-based discounts:

Savings Plans: 1-3 year commitment, up to 66% off On-Demand

Reserved Instances: 1-3 year, up to 72% off (less flexible)

Spot for non-critical workloads:

EKS Karpenter: automatically uses Spot instances for node groups

S3 cost optimisation:

```
aws s3api put-bucket-lifecycle-configuration --lifecycle-rules file://lifecycle.json
```

S3 Intelligent-Tiering for unpredictable access patterns

Monitor spend:

```
aws ce get-cost-and-usage --time-period Start=2024-01-01,End=2024-01-31 --granularity MONTHLY --metrics AmortizedCost
```

Cost optimisation hierarchy: (1) Eliminate waste (unused resources, over-provisioned instances), (2) Right-size (match instance type to actual utilisation), (3) Purchase model (Savings Plans for predictable baseline, Spot for burst), (4) Architecture (Fargate vs EC2, Lambda for variable workloads). AWS Compute Optimizer analyses usage and recommends right-sized instances. Budget alerts prevent surprise bills.

Official docs

- [AWS Cost Explorer](#)

Spoken answer · ~78 s

CodePipeline is AWS's managed CI/CD orchestration service. A pipeline is a series of stages, such as source, build, test, and deploy, each with actions. Source can be CodeCommit, GitHub through a connection, or S3; builds run in CodeBuild; deploy actions target CodeDeploy, ECS, CloudFormation, Lambda, or S3. Manual approval steps and cross-account deployments are supported, and IAM roles control permissions. Compared with GitHub Actions, CodePipeline's strengths are deep AWS integration, IAM-native permissions without managing external secrets, and running within your AWS account, which suits compliance-heavy environments. Its weaknesses are a less flexible and less friendly pipeline definition, a smaller ecosystem, slower iteration, and weaker pull-request-centred workflows. GitHub Actions has a huge marketplace of reusable actions, simple YAML workflows next to the code, tight pull-request integration, and matrix builds, and it can authenticate to AWS using OIDC with short-lived credentials, avoiding stored keys. Many teams use GitHub Actions for build and test and either trigger deployments from it or use CodePipeline for AWS-side deployment orchestration. Choose based on where code lives, compliance needs, and team familiarity. Whichever, keep the pipeline definition in code and use least-privilege roles.

Likely follow-ups

- How does GitHub Actions authenticate to AWS without stored keys?
- When does CodePipeline's IAM-native model help?

```
# CodePipeline stages:
```

```
Source (CodeCommit/GitHub) → Build (CodeBuild) → Test (CodeBuild) → Deploy  
(CodeDeploy/ECS/EKS)
```

```
# CodeBuild buildspec.yaml:
```

```
phases:
```

```
  build:
```

```
    commands:
```

```
      - docker build -t $IMAGE_URI .
```

```
      - docker push $IMAGE_URI
```

```
  post_build:
```

```
    commands:
```

```
      - kubectl apply -f k8s/ --record
```

```
# CodeDeploy for blue/green ECS:
```

```
aws deploy create-deployment --application-name myapp --deployment-group-name prod
```

CodePipeline is fully managed, integrates natively with AWS services, and keeps everything within your AWS account (no external secrets). CodeBuild runs build jobs in ephemeral containers. GitHub Actions is more flexible, has a larger action marketplace, and is preferred when the team already uses GitHub. For AWS-native shops: CodePipeline + CodeBuild + CodeDeploy is a coherent zero-ops CI/CD stack. For multi-cloud or GitHub-centric teams: GitHub Actions + AWS OIDC federation is simpler.

Official docs

- [AWS CodePipeline](#)
- [GitHub Actions documentation](#)

Q14 What is Karpenter — how does it improve EKS auto-scaling?

Containers

Spoken answer · ~78 s

Karpenter is a Kubernetes node autoscaler that provisions nodes directly, based on pending pod requirements, rather than scaling pre-defined node groups. The traditional Cluster Autoscaler adjusts the size of auto scaling groups, which means you have to pre-define instance types and groups, and scaling is slower and coarser. Karpenter watches for unschedulable pods, looks at their resource requests, node selectors, affinities, and tolerations, and then launches the best-fit instance directly through the EC2 API, choosing from many instance types and sizes, including spot, within the constraints you define in NodePool and EC2NodeClass resources. This gives faster scaling, better bin-packing and lower cost, and less configuration. Karpenter also handles consolidation: it removes or replaces underutilised nodes with cheaper or fewer ones, and expires nodes after a set lifetime so they pick up new AMIs, which helps patching. For spot, it handles interruption notices and diversifies across types. Considerations: set pod resource requests accurately, define disruption budgets and PodDisruptionBudgets to control consolidation impact, use limits in NodePools to cap spend, and run Karpenter itself on a small stable node group or Fargate. It works best with workloads that tolerate rescheduling.

Likely follow-ups

- How does consolidation reduce cost?
- Why must pod resource requests be accurate with Karpenter?

```
helm install karpenter oci://public.ecr.aws/karpenter/karpenter --version v0.33 -n karpenter
```

```
# NodePool defines what nodes Karpenter can provision:
```

```
spec:
```

```
template:
```

```
spec:
```

```
  requirements:
```

```
    - key: karpenter.sh/capacity-type
```

```
      values: [spot, on-demand]
```

```
    - key: node.kubernetes.io/instance-type
```

```
      values: [m6i.large, m6i.xlarge, m5.large, m5.xlarge]
```

```
  nodeClassRef:
```

```
    name: default
```

Karpenter replaces the Cluster Autoscaler with a more intelligent approach: it provisions nodes specifically shaped for pending pods (right instance type, right size) rather than scaling predefined node groups. Results: faster scaling (seconds vs minutes), better bin-packing (higher utilisation), automatic Spot diversification (uses many instance types to reduce interruption risk), and consolidation (replaces multiple underutilised nodes with one larger node). Reduces both cost and scaling latency significantly.

Official docs

- [Amazon EKS best practices guide](#)
- [Karpenter documentation](#)
- [Amazon EKS: User guide](#)

Q15 What is AWS PrivateLink — when do you use it vs VPC peering vs Transit Gateway?

Networking

🗣 Spoken answer · ~82 s

These three connect networks in different ways. PrivateLink exposes a specific service privately: the provider creates an endpoint service backed by a Network Load Balancer, and consumers create an interface endpoint in their VPC. Traffic stays on the AWS network, only that service is reachable, and overlapping CIDR ranges aren't a problem. Use it to consume SaaS, share an internal service with many accounts, or reach AWS services privately. VPC peering is a one-to-one, full-network connection between two VPCs. It is simple and cheap with low latency, but it is not transitive, so with many VPCs the number of connections grows quickly, CIDRs cannot overlap, and routes must be managed in each. Transit Gateway is a regional hub: you attach many VPCs, VPNs, and Direct Connect, and route between them centrally with route tables, supporting transitive routing and segmentation. It costs per attachment and per gigabyte, but scales operationally. Rule of thumb: a single service shared one-way, use PrivateLink; two or three VPCs needing full connectivity, peering; many VPCs or hybrid connectivity with central control, Transit Gateway. They can be combined, for example Transit Gateway for internal routing and PrivateLink for exposing specific services to external partners.

Likely follow-ups

- Why is VPC peering not transitive and why does that matter?
- When does overlapping CIDR rule out peering?

🚧 Connectivity options

PrivateLink	Peering / Transit Gateway
One service, one direction	Full network reach
No CIDR overlap issue	Needs non-overlapping CIDRs

VPC Peering: direct 1:1 VPC connection (non-transitive)

PrivateLink: expose a service privately (NLB endpoint service)

Transit Gateway: hub-and-spoke router for many VPCs

VPN/Direct Connect: on-premises to AWS

PrivateLink endpoint:

```
aws ec2 create-vpc-endpoint --vpc-id vpc-xxx --service-name com.amazonaws.us-east-1.s3 --vpc-endpoint-type Interface
```

Transit Gateway (connects 3+ VPCs):

```
aws ec2 create-transit-gateway
```

```
aws ec2 attach-transit-gateway-vpc-attachment --transit-gateway-id tgw-xxx --vpc-id vpc-xxx
```

VPC Peering: simple 1:1 direct connection, non-transitive ($A \leftrightarrow B$ and $B \leftrightarrow C$ does not give $A \leftrightarrow C$). PrivateLink: expose a service (NLB) to other VPCs privately — no full VPC peering, no CIDR overlap issues. Use for: SaaS services, internal platform services consumed across many VPCs. Transit Gateway: hub-and-spoke for 10+ VPCs — one attachment per VPC, full mesh routing. Preferred for large-scale multi-VPC architectures. Direct Connect: dedicated physical connection to AWS data centre for consistent low-latency or compliance requirements.

Official docs

- [Amazon VPC: VPC peering](#)
- [AWS Transit Gateway](#)
- [AWS PrivateLink](#)

Q16 **An EC2 instance has high CPU but your application is not doing extra work — how do you investigate?**

Troubleshooting

Spoken answer · ~86 s

If an instance shows high CPU but the application isn't doing extra work, the CPU is going somewhere else, so I find where. First, confirm the metric: CloudWatch CPU is measured by the hypervisor, and for burstable T instances check credit balance and whether it's throttled; steal time points to noisy neighbours. Then log in, or use Session Manager, and look at top or pidstat to see which processes consume CPU, and whether it's user, system, iowait, or softirq time. High user time in the application suggests a code path, such as a garbage-collection loop, a runaway thread, or a regex; take profiles or thread dumps. High system time suggests syscalls or networking overhead. High softirq suggests packet processing, perhaps a traffic spike or attack. Check for other processes: monitoring agents, antivirus, log shipping, cron jobs, or a cryptominer after compromise. Look at recent changes: deployments, configuration, new traffic patterns, and request mix, using ALB metrics. Check whether the instance type changed or the load balancer is skewing traffic to this instance. Use perf or flame graphs to attribute CPU. Remediate based on cause, whether that's fixing the code, moving the agent, scaling out, or isolating a compromised host, and add alarms to catch it earlier.

Likely follow-ups

- What does steal time indicate?
- How would you spot a compromised instance from CPU behaviour?

```
ssh ec2-user@instance-ip
```

```
top -bn1 | head -20
```

```
# overall CPU, top processes
```

```
ps aux --sort=-%cpu | head -10          # top CPU-consuming processes
```

```
mpstat -P ALL 1 5                      # per-CPU utilisation
```

```
sar -u 1 10                            # system activity over time
```

```
iostat -x 1 5                          # CPU wait time (iowait)
```

```
# Check for steal time (hypervisor contention):
```

```
top → %st (steal) – if high: instance is CPU-constrained by hypervisor
```

```
# Check CloudWatch metrics:
```

```
aws cloudwatch get-metric-statistics --metric-name CPUUtilization --namespace AWS/EC2
```

Common causes: noisy neighbour (high steal time on shared hardware → move to dedicated instance or Nitro-based type), runaway process (find with `ps aux`), kernel processes (`kworker`, `ksoftirqd` — check `dmesg` for errors), memory pressure causing swap usage (check `free -h`), and for `t2/t3`: CPU credit exhaustion (check `CPUCreditBalance` CloudWatch metric — unlimited mode prevents throttling).



Official docs

- [Amazon EC2 User Guide](#)

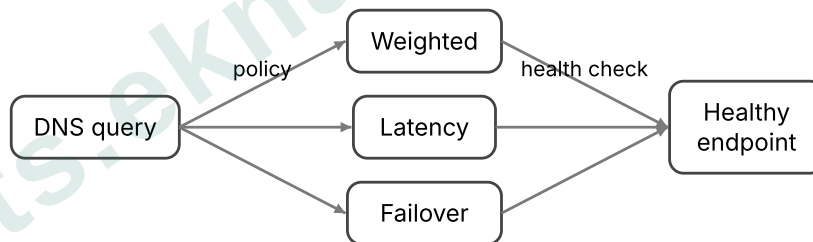
 Spoken answer · ~80 s

Route 53 is AWS's DNS service, providing domain registration, hosted zones, health checks, and routing policies. The production routing policies I use: simple, for one resource; weighted, to split traffic by percentage, useful for canary releases and gradual migrations; latency-based, to send users to the region with the lowest latency; failover, with active-passive routing driven by health checks, sending traffic to a standby when the primary is unhealthy; geolocation, to route by user location for compliance or localisation; geoproximity, which shifts traffic by distance with bias; and multivalue answer, to return several healthy records. Alias records map names to AWS resources like ALBs, CloudFront, and S3 website endpoints, work at the zone apex, and are free for queries, so I prefer them over CNAMEs. Health checks monitor endpoints, and can use CloudWatch alarms for private resources. Keep TTLs low enough for failover to take effect, but not so low that query volume and cost explode; remember resolvers and clients may cache beyond TTL. Combine policies, for example latency routing to regions with failover within each. Use private hosted zones for internal names. Manage records as code, and protect critical zones against accidental deletion.

Likely follow-ups

- Why prefer alias records over CNAMEs on AWS?
- How do health checks drive failover routing?

Route 53 policies



```
aws route53 list-hosted-zones
```

```
aws route53 change-resource-record-sets --hosted-zone-id Z123 --change-batch
file://record.json
```

```
# Routing policies:
```

```
Simple:           single resource, no health checks
```

```
Failover:        primary + secondary, health-check-based failover
```

```
Weighted:        split traffic by weight (canary: 90%/10%)
```

```
Latency:         route to lowest-latency region
```

Geolocation: route by user location (GDPR compliance, content localisation)

Multivalue: return multiple IPs (basic load distribution)

Health check:

```
aws route53 create-health-check --caller-reference $(date +%s) --health-check-config
Type=HTTPS,FullyQualifiedDomainName=api.example.com,Port=443,ResourcePath=/health
```

Failover routing + health checks = DNS-based disaster recovery. Weighted routing enables DNS-level canary deployments (10% to new region). Latency routing reduces global P99 latency significantly. Always attach health checks to production DNS records — Route 53 stops routing to unhealthy endpoints automatically. Route 53 Resolver enables DNS resolution between VPCs and on-premises networks.

Official docs

- [Amazon Route 53](#)
- [Route 53: Routing policies](#)

Q18 What is AWS Organizations and SCPs — how do you use them for multi-account governance?

IAM & Security

 Spoken answer · ~81 s

AWS Organizations lets you manage multiple accounts centrally, grouping them into organisational units in a hierarchy, with consolidated billing. Multi-account is the foundation of AWS governance because accounts are the strongest isolation boundary for security, blast radius, and billing. Service control policies are guardrails attached to the root, OUs, or accounts. They don't grant permissions; they define the maximum permissions available, so an action is allowed only if both the SCP and the identity's policies permit it. They apply to all principals in the member accounts, including the root user, but not the management account. Typical uses: deny leaving the organisation, deny disabling CloudTrail, GuardDuty, or Config, restrict usage to approved regions, deny creation of resources without encryption, prevent deletion of security tooling, and block risky services. Two approaches: a deny-list strategy, starting with allow-all and adding explicit denies, which is simpler to manage; or an allow-list, which is stricter but harder. Structure OUs by function, such as security, infrastructure, workloads, sandbox, and apply SCPs there. Use Control Tower or landing zone tooling to set up accounts with baselines. Test SCPs in a sandbox OU first, because a bad SCP can lock out whole accounts.

Likely follow-ups

- Why don't SCPs grant permissions?
- What are typical guardrails you apply with SCPs?

SCP that prevents disabling CloudTrail:

```
{"Effect": "Deny", "Action":  
["cloudtrail:StopLogging", "cloudtrail:DeleteTrail"], "Resource": "*"}
```

SCP that restricts to approved regions:

```
{"Effect": "Deny", "Action": "*", "Resource": "*", "Condition": {"StringNotEquals":  
{"aws:RequestedRegion": ["us-east-1", "eu-west-1"]}}}
```

Check effective permissions (SCP + IAM):

```
aws iam simulate-principal-policy --policy-source-arn arn:aws:iam::xxx:role/MyRole --  
action-names s3:DeleteBucket
```

Multi-account AWS: separate accounts for dev/staging/production, sandbox, security, logging. SCPs define the maximum permissions ceiling for an entire account — even the root user cannot exceed them. Common SCPs: prevent disabling security services (CloudTrail, GuardDuty, Security Hub), restrict to approved regions, prevent creating IAM users with console access, require MFA for sensitive actions. AWS Control Tower automates account vending and baseline security controls for new accounts.

Official docs

- [AWS Organizations: Service control policies](#)
- [AWS Organizations: What is AWS Organizations?](#)

Q19 How do you troubleshoot connectivity issues between AWS services?

Troubleshooting

Spoken answer · ~80 s

I troubleshoot connectivity by walking the path in layers, checking each component that could block it. First define source, destination, port, and protocol, and the symptom: timeout, refused, or a DNS failure. Timeout usually means something is dropping packets; refused means the packet arrived but nothing listens. Check DNS resolution from the source. Then check the source's security group outbound rules and the destination's inbound rules, with the right source reference. Check NACLs on both subnets, remembering they're stateless. Check route tables: does the source subnet have a route to the destination, through peering, transit gateway, NAT, or endpoint, and does the return path exist? Check for overlapping CIDRs, missing peering routes, and gateway attachments. For AWS service access, check endpoint policies and DNS settings for private endpoints. On the host, check that the service is listening on the expected interface, the host firewall, and the application configuration. Use the tools: VPC Reachability Analyzer analyses the configured path and shows what blocks it; VPC flow logs show accepted and rejected traffic; tcpdump and curl verbose give packet-level evidence; and Network Access Analyzer finds unintended access. Fix the root cause, and document it.

Likely follow-ups

- How does Reachability Analyzer help?
- What do flow logs tell you about a blocked connection?

```
# Test connectivity from EC2/EKS pod:
```

```
curl -v https://rds-endpoint:5432          # does TCP connect?
```

```
nmap -p 5432 rds-endpoint                  # port scan
```

```
tracert rds-endpoint                      # path to host
```

```
# VPC Flow Logs: check for REJECT
```

```
aws logs filter-log-events --log-group-name /vpc-flowlogs --filter-pattern "REJECT"
```

```
# Security Group: check inbound rules on RDS SG
```

```
aws ec2 describe-security-groups --group-ids sg-rds-xxx
```

```
# Route table: does private subnet route to target?
```

```
aws ec2 describe-route-tables --filters Name=association.subnet-id,Values=subnet-xxx
```

```
# Reachability Analyzer:
```

```
aws ec2 start-network-insights-analysis --network-insights-path-id path-xxx
```

Systematic connectivity debugging: (1) Security Group inbound rules on the destination, (2) NACL rules on source and destination subnets, (3) Route table has a route to the destination, (4) VPC Flow Logs show REJECT if any, (5) EC2 Reachability Analyzer gives a visual path analysis with exact blocking point. Most connectivity issues are Security Group misconfiguration — check both source SG (outbound) and destination SG (inbound).



Official docs

- [Amazon VPC: Security groups](#)
- [Amazon RDS User Guide](#)

Spoken answer · ~75 s

S3 is object storage organised into buckets, with keys identifying objects, designed for eleven nines of durability and high availability. Key production features. Storage classes: Standard, Intelligent-Tiering, Standard-IA, One Zone-IA, and Glacier tiers, with lifecycle rules to transition or expire data automatically. Versioning protects against overwrites and deletions, and combined with MFA delete or Object Lock gives immutability for compliance. Replication, cross-region or same-region, copies objects for disaster recovery or locality. Security: block public access at the account and bucket level, bucket policies and IAM for access control, encryption at rest with SSE-S3 or SSE-KMS, enforced TLS, VPC endpoints, and access logging. Consistency is strong for reads after writes. Performance scales by using many prefixes and parallel requests, multipart upload for large objects, and Transfer Acceleration or CloudFront for distant clients. Event notifications to SNS, SQS, Lambda, or EventBridge enable event-driven processing. Presigned URLs give temporary access. S3 Inventory and Storage Lens give visibility. Costs come from storage, requests, and data transfer, so lifecycle policies and avoiding many tiny objects matter. It is not a file system, so design applications accordingly.

Likely follow-ups

- How does Object Lock support compliance?
- How do lifecycle rules reduce cost?

```
aws s3 mb s3://mybucket --region us-east-1
```

```
# Versioning:
```

```
aws s3api put-bucket-versioning --bucket mybucket --versioning-configuration Status=Enabled
```

```
# Block public access (essential):
```

```
aws s3api put-public-access-block --bucket mybucket --public-access-block-configuration BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=true
```

```
# Lifecycle rule (transition to Glacier after 90 days):
```

```
aws s3api put-bucket-lifecycle-configuration --bucket mybucket --lifecycle-configuration file://lifecycle.json
```

```
# Pre-signed URL (temporary access):
```

```
aws s3 presign s3://mybucket/file.txt --expires-in 3600
```

S3 production checklist: versioning enabled (recover deleted/overwritten objects), Block Public Access enabled (prevent accidental public buckets), bucket policies over ACLs, encryption at rest (SSE-S3 or SSE-KMS), access logging, and lifecycle policies to manage cost. Pre-signed URLs grant time-limited access to

private objects without making them public — use for user file downloads. S3 Object Lock enables WORM (Write Once Read Many) for compliance data that must not be modified or deleted.

Official docs

- [Amazon S3 User Guide](#)

Q21 What is AWS Fargate — when do you use EKS on Fargate vs EC2 node groups?

Containers

Spoken answer · ~78 s

Fargate is a serverless compute engine for containers: you define pod or task resources and AWS provisions isolated compute, so there are no nodes to patch, scale, or manage. On EKS, a Fargate profile selects which pods, by namespace and labels, run on Fargate; each pod gets its own micro-VM. The benefits are no node management, strong isolation per pod, and paying for requested vCPU and memory per second. Trade-offs: higher cost per unit of compute than well-utilised EC2 or spot, no DaemonSets, no privileged containers or host access, limited GPU and storage options such as no EBS volumes in some cases, slower start because each pod needs capacity provisioned, and fixed size combinations. Choose Fargate for spiky, small, or isolated workloads, for teams that want to avoid node operations, for batch jobs, or for system pods in small clusters. Choose EC2 node groups, perhaps managed by Karpenter, for steady, large-scale workloads where bin-packing and spot give better economics; for workloads needing DaemonSets, GPUs, custom AMIs, privileged access, or high-performance networking. Many clusters mix both. Evaluate by cost modelling, operational burden, and technical constraints rather than preference.

Likely follow-ups

- Which Kubernetes features does Fargate not support?
- When is EC2 with Karpenter cheaper than Fargate?

```
# EKS Fargate profile:
```

```
eksctl create fargateprofile --cluster prod --name fp-app \
```

```
--namespace production --labels app=myapp
```

```
# Pod spec: no node selector needed – profile matches namespace/label
```

```
# Each pod gets its own micro-VM (gVisor isolation)
```

```
# No DaemonSets on Fargate (no node to run on)
```

```
# EC2 node groups – use when you need:
```

```
DaemonSets, privileged containers, GPU instances, custom AMIs
```

Fargate: no node management, per-pod billing, stronger isolation (each pod in its own micro-VM), scales to zero. EC2: more control, cheaper at scale, required for DaemonSets/GPU/privileged containers. Hybrid: use Fargate for variable/bursty workloads (auto-scales without node management), EC2 for steady baseline (Savings Plan pricing). Fargate does not support node-level metrics (no Node Exporter) — use Container Insights for pod-level metrics instead.

Official docs

- [Amazon EKS: Managed node groups](#)
- [Amazon EKS: User guide](#)
- [Amazon ECS Developer Guide](#)

Q22 What is CloudTrail — how do you use it for compliance and audit?

Monitoring

Spoken answer · ~78 s

CloudTrail records API activity in your AWS account: who called what, from where, when, with which parameters, and whether it succeeded. Management events cover control-plane actions like creating an instance or changing a policy; data events cover object-level actions such as S3 reads or Lambda invocations, which must be enabled and cost more. For compliance and audit, I configure an organisation trail covering all accounts and regions, delivered to a central S3 bucket in a dedicated log-archive account. Protect the logs: enable log file integrity validation, which makes tampering detectable, encrypt with KMS, restrict bucket access, enable versioning and Object Lock, and deny deletion through SCPs. Send events to CloudWatch Logs or EventBridge for real-time alerting on important actions: root user login, IAM policy changes, console logins without MFA, disabling of CloudTrail, security group changes, and KMS key deletion. For investigation, use CloudTrail Lake or Athena to query historical events. Retain logs according to regulatory requirements. Combine with AWS Config for resource configuration history, and GuardDuty for threat detection. During an incident, CloudTrail answers who did what and when. Review regularly and test that alerts fire.

Likely follow-ups

- How does log file integrity validation work?
- Which CloudTrail events deserve real-time alerts?

```
aws cloudtrail create-trail --name prod-trail --s3-bucket-name my-audit-bucket --is-multi-region-trail
```

```
aws cloudtrail start-logging --name prod-trail
```

```
# Query CloudTrail with Athena:
```

```
SELECT eventname, useridentity.arn, sourceipaddress, eventtime
```

```
FROM cloudtrail_logs
```

```
WHERE eventname = "DeleteBucket" AND eventtime > "2024-01-01"
```

```
# Real-time: CloudWatch Events rule on CloudTrail
```

```
aws events put-rule --event-pattern file://console-login-event.json
```

CloudTrail logs every AWS API call — who did what, when, and from where. Enable in all regions (--is-multi-region-trail) and log to an S3 bucket with versioning and MFA delete protection. Enable CloudTrail Insights to detect unusual API activity (burst of DescribeInstances calls might indicate reconnaissance). Atop CloudTrail, AWS Config continuously records resource configuration and detects drift. GuardDuty analyses CloudTrail, VPC Flow Logs, and DNS logs for threat detection.

Official docs

- [AWS CloudTrail](#)

Q23 What is ElastiCache — Redis vs Memcached — how do you use it with EKS?

Databases

Spoken answer · ~78 s

ElastiCache is managed in-memory caching, supporting Redis, now also Valkey, and Memcached. Redis offers rich data structures such as lists, sets, sorted sets, and streams; persistence options; replication with automatic failover across zones; cluster mode for sharding; pub/sub; Lua scripting; and encryption and authentication. Memcached is simpler: a multi-threaded key-value cache, with no persistence or replication, easy horizontal scaling by adding nodes, and suited to simple object caching where losing data is fine. Choose Redis when you need data structures, high availability, persistence, session storage, leaderboards, rate limiting, or queues; choose Memcached for a simple, multi-threaded cache with the lowest complexity. With EKS: place the cluster in private subnets of the same VPC; allow access from the node or pod security groups; use the cluster endpoint in application configuration, delivered through ConfigMaps or Secrets; enable in-transit encryption and AUTH or IAM authentication; use a cluster-aware client when cluster mode is on; and set connection pooling and timeouts. Design for cache failure so the application falls back to the database gracefully. Monitor hit rate, evictions, memory, CPU, and replication lag, and set TTLs and an eviction policy deliberately.

Likely follow-ups

- Why would you pick Redis over Memcached?
- How should the application behave if the cache is unavailable?

```
aws elasticache create-replication-group --replication-group-id myredis \
```

```
--replication-group-description "prod redis" \
```

```
--engine redis --cache-node-type cache.r6g.large \
```

```
--num-cache-clusters 2 --multi-az-enabled
```

```
# Access from EKS (in same VPC, private subnet):
```

```
redis-cli -h myredis.abc.cache.amazonaws.com -p 6379
```

```
# Redis connection in Python:
```

```
r = redis.Redis(host=os.environ["REDIS_HOST"], port=6379, ssl=True)
```

Redis vs Memcached: Redis supports persistence, replication, data structures (sorted sets, streams), pub/sub, Lua scripting — use for session store, leaderboards, queues, rate limiting. Memcached: pure in-memory, simpler, multi-threaded — use when you need only simple key-value caching and maximum throughput. For production: ElastiCache Redis with Multi-AZ (automatic failover), encryption in transit (TLS), encryption at rest, and AUTH token. Access from EKS only via private subnet — never expose cache publicly.

Official docs

- [Amazon ElastiCache](#)
- [Amazon EKS: User guide](#)
- [Redis documentation](#)

Q24 An EKS pod cannot reach an external API — how do you debug it?

Troubleshooting

Spoken answer · ~87 s

When a pod cannot reach an external API, I work outward from the pod. First reproduce from inside: exec into the pod, or run a debug container, and test DNS resolution with nslookup or dig, then connect with curl using verbose output, to see whether it fails at DNS, TCP connect, TLS, or HTTP. If DNS fails, check CoreDNS pods and logs, the pod's resolv.conf, and network policies blocking port 53. If the connection times out, check egress network policies in the namespace, then the node's security group and the subnet's route table: private subnets need a route to a NAT gateway, and the NAT gateway must be healthy and in a public subnet with an internet gateway route. Check NACLs, because they're stateless. With the VPC CNI and custom networking or security groups for pods, verify the pod security group allows egress. Check whether the destination requires an allow-listed source IP, since NAT gateway elastic IPs are the source address. Check proxy settings and TLS issues such as missing CA certificates. Use VPC flow logs to see whether packets leave and are rejected, and Reachability Analyzer to analyse the route. Also confirm the external API itself isn't down, and rate limits aren't the issue. Fix and add monitoring.

Likely follow-ups

- How do you tell DNS failures from connection failures?
- Why does the NAT gateway's IP matter for external APIs?

```
# Step 1: can the pod resolve DNS?
```

```
kubectl exec mypod -- nslookup api.external.com
```

```
# Step 2: can the pod reach the IP?
```

```
kubectl exec mypod -- curl -v https://api.external.com --max-time 5
```

```
# Step 3: Check Security Group on the node
```

```
aws ec2 describe-security-groups --group-ids sg-node-xxx # allow 443 outbound?
```

```
# Step 4: Does NAT Gateway exist for the private subnet?
```

```
aws ec2 describe-route-tables --filters Name=association.subnet-id,Values=subnet-node
```

```
# Step 5: VPC Flow Logs REJECT on the node IP
```

```
# Step 6: If using VPC CNI, check pod IP in Security Group rules
```

EKS pod connectivity path: pod → node Security Group → route table → NAT Gateway (if private subnet) → internet. VPC CNI assigns pod IPs from the VPC CIDR — Security Group rules must allow from pod IPs, not just node IPs (use Security Groups for Pods feature for pod-level SG enforcement). Common fix: NAT Gateway not in the route table for the node subnet, or Security Group blocking outbound 443.

Official docs

- [Amazon EKS: User guide](#)

Q25 What is AWS Secrets Manager vs Parameter Store — when do you use each?

IAM & Security

 Spoken answer · ~79 s

Both store configuration and secrets, with different strengths. Secrets Manager is purpose-built for secrets: native automatic rotation, with managed rotation for RDS and others and Lambda-based rotation for custom secrets; cross-region replication; fine-grained resource policies; versioning with staging labels; and integration with RDS credentials. It costs per secret per month plus API calls. Parameter Store, part of Systems Manager, stores strings, string lists, and secure strings encrypted with KMS, organised hierarchically by path. Standard parameters are free, with advanced parameters costing more, and it has lower throughput limits. It has no built-in rotation. Use Secrets Manager for credentials that need rotation, such as database passwords and API keys, for secrets shared across accounts or regions, and where audit and lifecycle features are valuable. Use Parameter Store for general application configuration, feature flags, and non-sensitive values, or secrets that rarely change, where cost matters at scale. In EKS, either can be mounted via the Secrets Store CSI driver or fetched by the application with an IAM role, and external-secrets syncs them into Kubernetes. Whichever you choose, restrict access with IAM, encrypt with KMS, never log values, and audit access with CloudTrail.

Likely follow-ups

- What does Secrets Manager rotation do that Parameter Store doesn't?
- How do pods consume these secrets on EKS?

```
# Secrets Manager: full lifecycle management
```

```
aws secretsmanager create-secret --name prod/db/password --secret-string "mysecretpassword"
```

```
aws secretsmanager get-secret-value --secret-id prod/db/password
```

```
# Automatic rotation (Lambda function):
```

```
aws secretsmanager rotate-secret --secret-id prod/db/password --rotation-lambda-arn arn:xxx
```

```
# Parameter Store: config + secrets (cheaper)
```

```
aws ssm put-parameter --name /prod/db/password --value "mysecret" --type SecureString
```

```
aws ssm get-parameter --name /prod/db/password --with-decryption
```

Secrets Manager: auto-rotation (natively integrates with RDS, Redshift, ElastiCache), cross-account access, versioning, cost \$0.40/secret/month. Parameter Store: free tier (Standard), cheaper at scale, does not support auto-rotation. Use Secrets Manager for credentials requiring rotation (database passwords, API keys). Use Parameter Store for configuration values and non-sensitive parameters. External Secrets Operator syncs both into Kubernetes Secrets automatically — no application code changes needed.

Official docs

- [AWS Secrets Manager](#)
- [AWS Systems Manager Parameter Store](#)

Q26 What is AWS Cost Explorer and how do you use it to identify waste?

Cost Opt

Spoken answer · ~80 s

Cost Explorer is the AWS tool for visualising and analysing spend and usage over time. To find waste, I first set up cost allocation tags so spend maps to teams, services, and environments; without tagging, the data can't be acted on. Then I slice the data: group by service to see the biggest line items, by account, by usage type, and by tag; look at daily granularity to spot spikes; and use filters for specific services. Typical waste signals: growth in a service with no matching traffic growth; idle EC2 and RDS instances; large data transfer lines; unattached EBS volumes and old snapshots; NAT gateway data processing charges; unused load balancers and elastic IPs; and S3 storage in expensive classes. Cost Explorer's rightsizing recommendations, savings plan and reservation recommendations and coverage and utilisation reports show where commitments help. Combine it with Compute Optimizer, Trusted Advisor, and the Cost and Usage Report queried in Athena for deeper analysis. Set up AWS Budgets and Cost Anomaly Detection to alert on unexpected changes. Make it a recurring practice: a monthly review with owners, tracking unit costs, and recording savings. Fix the causes, not just the symptoms.

Likely follow-ups

- Why are cost allocation tags essential?
- What is Cost Anomaly Detection for?

```
aws ce get-cost-and-usage --time-period Start=2024-01-01,End=2024-01-31 \
```

```
--granularity MONTHLY --metrics AmortizedCost \
```

```
--group-by Type=DIMENSION,Key=SERVICE
```

```
# Rightsizing recommendations:
```

```
aws ce get-rightsizing-recommendation --service AmazonEC2
```

```
# Reserved Instance coverage report:
```

```
aws ce get-reservation-coverage --time-period Start=2024-01-01,End=2024-01-31
```

```
# Savings Plan recommendations:
```

```
aws ce get-savings-plans-purchase-recommendation --savings-plans-type COMPUTE_SP
```

Cost Explorer workflow: identify top 5 services by spend (usually EC2, EKS, RDS, data transfer, S3), drill into each by region and resource type. Rightsizing recommendations identify over-provisioned EC2/RDS instances. Reservation coverage shows what % of On-Demand is covered by commitments — under 60% coverage means overspending. Tag all resources with environment, team, and project to enable per-team cost allocation. Use AWS Budgets to alert when spend exceeds threshold.

 Official docs

- [AWS Cost Explorer](#)

Q27 What is AWS WAF — how do you configure it for EKS/ALB?

Networking

Spoken answer · ~85 s

AWS WAF is a web application firewall that inspects HTTP requests and allows, blocks, counts, or challenges them based on rules, in a web ACL attached to CloudFront, an ALB, API Gateway, AppSync, or Cognito. For an EKS application behind an ALB, you can attach the web ACL to the ALB by using an annotation on the Ingress, in the AWS Load Balancer Controller, specifying the web ACL ARN. Rule building blocks: AWS managed rule groups, such as the core rule set, known bad inputs, SQL injection, and the IP reputation list; rate-based rules to limit requests per IP; custom rules matching headers, URIs, geographies, or IP sets; and bot control and account takeover prevention for specialised needs. My practice: start with managed rules in count mode, review the logs for false positives, then switch to block; add rate limiting on sensitive endpoints like login; set the default action to allow with explicit blocks, or the reverse for strict applications; enable full logging to S3, CloudWatch, or Firehose and analyse blocked requests; and manage the web ACL as code. Remember WAF has limits: it doesn't replace secure coding, and rule capacity units limit complexity. Pair it with Shield for DDoS and monitor its metrics.

Likely follow-ups

- Why deploy managed rules in count mode first?
- How do you attach a WAF to an ALB created by the Load Balancer Controller?

```
# Create WAF Web ACL:
```

```
aws wafv2 create-web-acl --name prod-waf --scope REGIONAL \
  --default-action Allow={} --rules file://waf-rules.json
```

```
# Attach to ALB:
```

```
aws wafv2 associate-web-acl --web-acl-arn WAF_ARN --resource-arn ALB_ARN
```

```
# Key managed rule groups:
```

```
AWSManagedRulesCommonRuleSet    - OWASP Top 10 (SQLi, XSS)
```

```
AWSManagedRulesAmazonIpReputationList – block known bad IPs
```

```
AWSManagedRulesKnownBadInputsRuleSet – log4j, Spring4shell
```

```
# Rate limiting rule:
```

```
{"Type": "RATE_BASED", "Limit": 2000, "AggregateKeyType": "IP"}
```

AWS WAF protects ALBs, API Gateway, and CloudFront from common web attacks. Start with AWS Managed Rule Groups (no custom rules needed) — they cover OWASP Top 10 automatically. Rate limiting prevents DDoS and credential stuffing. WAF logs to S3/CloudWatch for security analysis. Shield Advanced (paid) adds DDoS protection, 24/7 AWS DDoS Response Team access, and attack cost protection. For EKS + ALB: attach WAF to the ALB ARN provisioned by the AWS Load Balancer Controller.

Official docs

- [Elastic Load Balancing](#)
- [Application Load Balancer](#)
- [AWS WAF](#)

Q28 How do you diagnose high latency in an AWS-hosted application?

Troubleshooting

Spoken answer · ~84 s

To diagnose high latency, I localise it before fixing anything. Start from the user's perspective: which requests, which users or regions, since when, and whether it correlates with a deployment or traffic change. Use ALB metrics, such as target response time versus total request time, to see whether the delay is at the load balancer, network, or application. Check percentiles, not averages. Use distributed tracing from X-Ray or OpenTelemetry to find which hop or dependency is slow: the database, a downstream service, an external API, or a cache miss. Then drill into the suspect. For the application: CPU, memory, garbage collection, thread pool exhaustion, connection pool waits, and cold starts. For databases: Performance Insights for slow queries, lock waits, connection saturation, IOPS limits, and replica lag. For the network: cross-zone or cross-region calls, NAT gateway saturation, DNS resolution delay, and packet loss. For compute: CPU credit exhaustion on burstable instances, noisy neighbours, throttled containers from CPU limits. For scaling: autoscaling lag during traffic spikes. Compare against baseline dashboards. Check recent changes with CloudTrail and deployment history. Once the bottleneck is identified, fix it, whether through query tuning, caching, scaling, or configuration, and confirm with before-and-after measurements. Add alarms on latency SLOs.

Likely follow-ups

- Why look at percentiles instead of average latency?
- How does tracing locate the slow hop?

```
# Step 1: Which component is slow? (ALB access logs)
```

```
aws logs filter-log-events --log-group-name alb-access-logs --filter-pattern "[request_processing_time > 1]"
```

Step 2: Is it the application or the database?

```
aws xray get-service-graph --start-time $(date -d "1 hour ago" +%s) --end-time $(date +%s)
```

Step 3: RDS Performance Insights

```
aws pi get-resource-metrics --service-type RDS --identifier db-xxx --metric-queries file://queries.json
```

Step 4: EKS: check CPU throttling

```
container_cpu_cfs_throttled_seconds_total rate
```

Step 5: Data transfer between AZs? (cross-AZ latency adds 1-3ms)

Latency diagnosis layers: (1) DNS resolution time (Route 53 latency dashboard), (2) TLS handshake time (ALB target response time), (3) Application processing time (X-Ray trace waterfall), (4) Database query time (RDS Performance Insights, slow query log), (5) Resource constraints (CPU throttling on pods, RDS CPU/IOPS limits). Cross-AZ data transfer adds latency AND cost — colocate application pods and RDS read replicas in the same AZ when possible.



Official docs

- [Amazon RDS User Guide](#)

 Spoken answer · ~85 s

EKS upgrades need planning because Kubernetes versions move quickly and deprecated APIs can break workloads. EKS only allows upgrading the control plane one minor version at a time, and each version has a limited support window. My process. Preparation: read the release notes, check for deprecated or removed APIs using tools like kubent or Pluto and EKS upgrade insights, verify that add-ons, ingress controllers, CNI, CSI drivers, and operators support the target version, and make sure PodDisruptionBudgets, replicas, and readiness probes are sensible. Test the upgrade in a non-production cluster that mirrors production. Take backups of critical state. Then upgrade the control plane, which AWS does with rolling replacement and no downtime for the API in normal cases. Next upgrade the add-ons, then the data plane: for managed node groups use a rolling update, or with Karpenter let drift replace nodes with the new AMI; for self-managed nodes create new groups and drain old ones. Nodes may lag the control plane within the supported skew. Monitor workload health during draining. Validate with smoke tests and check metrics. Have a rollback plan; control plane upgrades can't be undone, so staging validation is vital. Schedule upgrades regularly, rather than letting versions age, and document the runbook.

Likely follow-ups

- How do you find deprecated APIs before an upgrade?
- Why can't an EKS control plane upgrade be rolled back?

```
# Check current version:
```

```
aws eks describe-cluster --name prod --query cluster.version
```

```
# Upgrade control plane first:
```

```
aws eks update-cluster-version --name prod --kubernetes-version 1.30
```

```
aws eks wait cluster-active --name prod
```

```
# Upgrade add-ons:
```

```
aws eks update-addon --cluster-name prod --addon-name kube-proxy --addon-version v1.30.0-eksbuild.1
```

```
aws eks update-addon --cluster-name prod --addon-name vpc-cni
```

```
aws eks update-addon --cluster-name prod --addon-name coredns
```

```
# Upgrade node groups:
```

```
aws eks update-nodegroup-version --cluster-name prod --nodegroup-name workers
```

EKS upgrade sequence: always control plane first, then add-ons, then node groups. One minor version at a time (1.28 → 1.29, not 1.28 → 1.30). Test on non-production cluster first. Check deprecated API versions in workloads before upgrading (kubectl convert + pluto for deprecated API detection). EKS supports two minor versions behind current — upgrade before support ends to maintain managed node group patching. Use managed node groups for node upgrades — they drain nodes automatically.

Official docs

- [Kubernetes: Upgrading kubeadm clusters](#)
- [Amazon EKS: Updating a cluster](#)
- [Amazon EKS: User guide](#)

Q30 Walk through a production incident where an EKS service is returning 503s.

Troubleshooting

Spoken answer · ~85 s

For an EKS service returning 503s, I work from the edge inward while limiting impact. First, scope it: all requests or some, which endpoints, since when, and what changed recently, like a deployment or scaling event. If a recent deploy is the likely cause and the evidence fits, roll back immediately and investigate afterward. Check the ALB: target group health, the count of healthy targets, 5xx from the load balancer versus from targets. A 503 from the ALB itself often means no healthy targets are registered. Then in Kubernetes: look at pod status for crash loops, OOM kills, and failing readiness probes, check events, check whether the Service has endpoints, and whether labels and selectors match. Look at the pod logs and recent restarts. Check resource pressure: CPU throttling, memory limits, node capacity, and pending pods; check HPA behaviour. Check dependencies: database connections exhausted, downstream timeouts, DNS failures. Check networking: security groups between ALB and pods, network policies, and the load balancer controller's logs. Verify rollout settings such as maxUnavailable and readiness gates, and graceful shutdown during deployments. Mitigate by scaling out, restarting, rolling back, or shedding load. Afterwards, write a post-incident review with the timeline, the root cause, and the follow-up actions.

Likely follow-ups

- What does an ALB-generated 503 usually mean?
- How do readiness probes affect 503s during deployments?

```
# Step 1: Check ALB target health:
```

```
aws elbv2 describe-target-health --target-group-arn tg-xxx # targets healthy?
```

```
# Step 2: Check pods:
```

```
kubectl get pods -n prod -l app=myapp # are pods Running?
```

```
kubectl describe pod myapp-xxx -n prod    # events show issues
```

Step 3: Check endpoints:

```
kubectl get endpoints myapp -n prod      # any IPs registered?
```

Step 4: Check logs:

```
kubectl logs -l app=myapp -n prod --tail=50 | grep -i error
```

Step 5: Check HPA:

```
kubectl get hpa -n prod                  # is max replicas reached?
```

Step 6: Check node capacity:

```
kubectl get nodes                        # all Ready?
```

```
kubectl top nodes                        # CPU/memory pressure?
```

503 from ALB means no healthy targets. Systematic checks: (1) ALB target group health — are pod IPs registered? (2) Pod status — any CrashLoopBackOff or Pending? (3) Kubernetes Endpoints — are pod IPs listed? (4) Readiness probe — is it passing? (5) Service selector — does it match pod labels? (6) Node capacity — are pods pending due to insufficient node resources? The most common cause: readiness probe failure (the pod starts but fails health check, so ALB never routes to it).

Official docs

- [Amazon EKS: User guide](#)

Q31 **What is the IAM policy evaluation logic — how are Allow, Deny, and boundaries resolved?**

IAM & Security

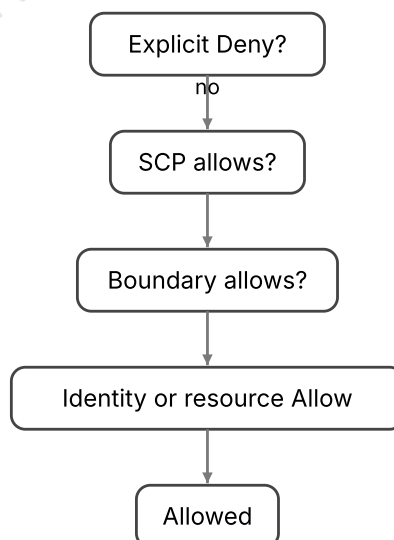
 Spoken answer · ~85 s

IAM evaluates every request with a defined logic. By default, everything is implicitly denied. The evaluation gathers all applicable policies: identity-based policies, resource-based policies, permission boundaries, service control policies, resource control policies, session policies, and VPC endpoint policies. First, any explicit Deny in any applicable policy ends the evaluation with a deny; an explicit deny always wins. Next, if an Organizations SCP applies, the action must be allowed by it, otherwise it is denied. Then, within the account, the request must be allowed by the identity-based policy or, in some cases, a resource-based policy. For same-account access, a resource-based policy that names the principal can grant access on its own. For cross-account access, both the identity-based policy in the caller's account and the resource policy in the target account must allow it. A permission boundary on a user or role sets the maximum permissions: the effective permissions are the intersection of the boundary and the identity-based policy. Session policies similarly limit assumed role sessions. So the effective permission is the intersection of all the guardrails, with allows needed in the right places. Conditions can refine each statement. When debugging, use the IAM policy simulator and CloudTrail error messages, and check for explicit denies first.

Likely follow-ups

- Why does an explicit deny override everything?
- How do permission boundaries interact with identity policies?

 IAM evaluation



Evaluation order (simplified):

1. Explicit DENY anywhere -> DENY (always wins)
2. SCP (Organizations) -> must ALLOW or implicit deny

3. Resource-based policy / Identity policy -> needs an ALLOW

4. Permission boundary -> caps the max; must also allow

5. Session policy (assume-role) -> further caps

No explicit allow anywhere = implicit DENY

The golden rule: an explicit Deny **always** overrides any Allow. A request is permitted only if every applicable layer allows it AND nothing denies it. SCPs and permission boundaries do not *grant* access — they set the maximum; you still need an identity/resource policy that allows the action. This is why a user can have AdministratorAccess yet be blocked by an SCP or boundary.

Debugging access-denied: Use the IAM Policy Simulator and CloudTrail to see which policy denied a call. Decode the encoded authorization message with `aws sts decode-authorization-message`. The usual culprit is an SCP or permission boundary the user forgot about — not the identity policy.

Official docs

- [AWS IAM: Policy evaluation logic](#)

Q32 How do you securely give applications on EC2/EKS access to AWS without static keys?

IAM & Security

Spoken answer · ~83 s

The principle is to use temporary credentials issued to the workload's identity, not static access keys stored in code or config. On EC2, attach an instance profile with an IAM role; the SDK retrieves temporary credentials from the instance metadata service automatically and refreshes them. Enforce IMDSv2, which requires session tokens and mitigates SSRF-based credential theft, and limit the hop count. On EKS, use IAM Roles for Service Accounts, where the cluster's OIDC provider lets a pod's Kubernetes service account assume an IAM role through web identity; or EKS Pod Identity, a simpler mechanism with an agent on each node. Both give per-pod least-privilege roles, rather than all pods inheriting the node role, which should be minimal. For ECS, use task roles; for Lambda, execution roles. For CI systems outside AWS, use OIDC federation: the provider issues a token, and AWS exchanges it for temporary credentials via a role whose trust policy checks the repository and branch. For on-premises workloads, IAM Roles Anywhere uses X.509 certificates. Also scan repositories for leaked keys, use SCPs to forbid creating IAM user keys where possible, and monitor with Access Analyzer and credential reports. The result is short-lived, auditable, automatically rotated credentials.

Likely follow-ups

- Why enforce IMDSv2?
- What is the advantage of IRSA over using the node role?

```
# EC2: instance profile (role) -> auto-rotated temp creds via IMDS:
```

```
aws iam create-instance-profile --instance-profile-name app-profile
```

```
# ALWAYS enforce IMDSv2 (token-required) to block SSRF credential theft:
```

```
aws ec2 modify-instance-metadata-options --instance-id i-xxx --http-tokens required --http-put-response-hop-limit 1
```

```
# EKS: IRSA (IAM Roles for Service Accounts) via OIDC:
```

```
eksctl create iamserviceaccount --cluster prod --name s3-reader --attach-policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess --approve
```

```
# Or EKS Pod Identity (newer, simpler than IRSA):
```

```
aws eks create-pod-identity-association --cluster-name prod --namespace app --service-account s3-reader --role-arn arn:...
```

Never bake long-lived access keys into apps or images. On EC2 use an instance profile; on EKS use IRSA or Pod Identity so each pod gets a scoped role with temporary, auto-rotated STS credentials. For CI use OIDC federation (GitHub Actions, GitLab) instead of stored keys.

IMDSv2 is critical: IMDSv1 let an SSRF bug fetch instance credentials with a single GET. IMDSv2 requires a session token (PUT then GET) and a hop limit of 1, which stops most SSRF-to-credential-theft attacks. Enforce it org-wide.

Official docs

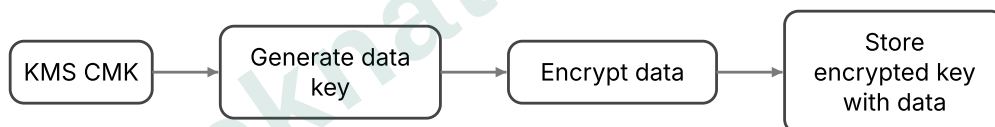
- [AWS IAM: Temporary credentials \(STS\)](#)
- [GitHub Actions: OIDC with AWS](#)
- [GitHub Actions: OpenID Connect](#)

Spoken answer · ~86 s

KMS, the Key Management Service, creates and manages cryptographic keys, with the key material protected in HSMs and never exposed in plaintext. KMS keys are used mainly to encrypt small data directly, up to four kilobytes, so for larger data it uses envelope encryption. In envelope encryption, the service asks KMS for a data key; KMS returns a plaintext data key and a copy encrypted under the KMS key. The service uses the plaintext data key to encrypt the data locally, quickly, then discards the plaintext key and stores the encrypted data key alongside the ciphertext. To decrypt, the service sends the encrypted data key to KMS, which decrypts it if the caller is authorised, and then decrypts the data locally. This keeps the master key within KMS, reduces latency and cost for large data, and allows many data keys per master key. Access to a KMS key is controlled by the key policy, which is the primary access control and must allow the account for IAM policies to apply, plus IAM policies and grants. Key policies can restrict usage to specific roles and services, and use conditions such as encryption context. Use customer-managed keys for control over policy and audit, and monitor usage in CloudTrail.

Likely follow-ups

- Why use envelope encryption instead of encrypting everything with KMS directly?
- What is the role of the key policy?

Envelope encryption

```
# Create a customer-managed key (CMK):
```

```
aws kms create-key --description "app data key"
```

```
# Envelope encryption: generate a data key, KMS returns plaintext + encrypted:
```

```
aws kms generate-data-key --key-id alias/app --key-spec AES_256
```

```
# Encrypt locally with the plaintext data key, store the ENCRYPTED data key.
```

```
# To decrypt: send encrypted data key back to KMS -> get plaintext key back.
```

```
# Enable automatic annual rotation:
```

```
aws kms enable-key-rotation --key-id key-id
```

Envelope encryption: KMS encrypts a small data key, not your large data. You encrypt the bulk data locally with the data key (fast, no size limit), then store the KMS-encrypted data key alongside it. KMS never sees your data. Access is controlled by the **key policy** (on the key itself) combined with IAM — both must allow. Grants give temporary, scoped key access.

Key policy vs IAM: Unlike most services, a KMS key has its own resource policy that is authoritative — if the key policy does not allow the principal (directly or by delegating to IAM), no IAM policy can grant access. Locking yourself out of a key is a classic mistake; always keep the account root or an admin role in the key policy.

Official docs

- [Kubernetes: Encrypting Secret data at rest](#)
- [AWS KMS](#)

Q34 How do you manage secrets in AWS? Compare Secrets Manager and SSM Parameter Store. IAM & Security

Spoken answer · ~83 s

Both store secrets securely with KMS encryption and IAM access control, but they serve different needs. Secrets Manager is designed for secrets: automatic rotation, using Lambda functions with built-in support for RDS, Redshift, and DocumentDB; native cross-region replication; resource policies for cross-account sharing; versioning with staging labels, which enables safe rotation; and a deletion recovery window. It charges per secret per month and per API call. Parameter Store is a hierarchical configuration store with SecureString parameters encrypted by KMS; standard-tier parameters are free, and it offers version history and parameter policies on the advanced tier, such as expiration notifications, but no native rotation. Throughput limits are lower. My guidance: use Secrets Manager for credentials that should rotate, like database passwords, third-party API keys, and anything where rotation and audit matter, or that needs replication across regions. Use Parameter Store for application configuration, feature toggles, endpoints, and low-sensitivity values, and for static secrets when cost is a major concern. Both work with IAM roles, so applications can fetch values at runtime without static credentials; in Kubernetes use the Secrets Store CSI driver or External Secrets Operator. Always limit access by least privilege, never log secret values, and monitor access with CloudTrail.

Likely follow-ups

- What is a staging label in Secrets Manager?
- When is Parameter Store the cheaper sensible choice?

```
# Secrets Manager: built-in rotation, JSON secrets, cross-region replication:
```

```
aws secretsmanager create-secret --name prod/db --secret-string
\ '{"user": "app", "pass": "..."} \'
```

```
aws secretsmanager get-secret-value --secret-id prod/db
```

```
# Rotation with a Lambda (e.g. RDS):
```

```
aws secretsmanager rotate-secret --secret-id prod/db --rotation-lambda-arn arn:...
```

```
# SSM Parameter Store (cheaper, SecureString via KMS):
```

```
aws ssm put-parameter --name /prod/api-key --value "..." --type SecureString
```

```
aws ssm get-parameter --name /prod/api-key --with-decryption
```

Secrets Manager: automatic rotation (native RDS/Redshift/DocumentDB integration), fine-grained resource policies, cross-region replication — but costs per secret + per API call. SSM Parameter Store: free standard tier, SecureString encrypted with KMS, hierarchical paths, great for config + simple secrets — but no built-in rotation. Use Parameter Store for config and low-churn secrets; Secrets Manager when you need automatic rotation or replication.

Decision rule: Need rotation or cross-region secret replication → Secrets Manager. Just need encrypted config/values cheaply → Parameter Store SecureString. Both integrate with EKS via the Secrets Store CSI driver so pods mount secrets without app code changes.

Official docs

- [AWS Secrets Manager](#)
- [AWS Systems Manager Parameter Store](#)
- [AWS Systems Manager Session Manager](#)

Q35 What are SCPs and how do you use AWS Organizations for multi-account governance?

IAM & Security

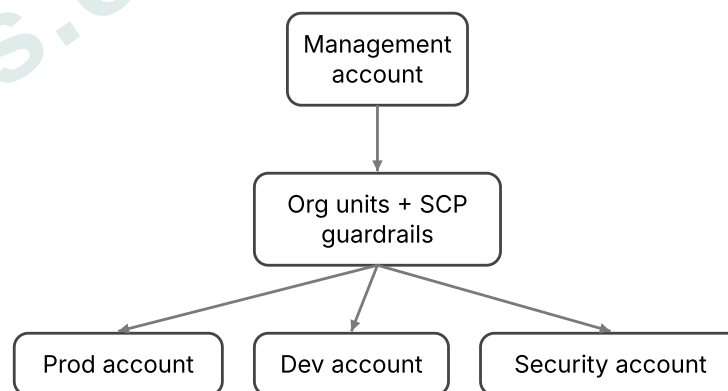
🗣 Spoken answer · ~86 s

Service control policies are organisation-level guardrails that set the maximum permissions for accounts in an AWS Organization. They are attached to the root, organisational units, or individual accounts, and are inherited down the hierarchy. An SCP never grants permission on its own; an action is permitted only when both the SCP and the identity or resource policy allow it, so SCPs filter the permissions available, and they apply to all users and roles in member accounts, including the account root user, but not to the management account. To use Organizations for governance, I design an OU structure by function and risk, for example security, infrastructure, production workloads, non-production, sandbox, and a suspended OU for closed accounts. Each account is an isolation boundary for blast radius, billing, and access. Typical SCPs: deny actions outside approved regions; deny disabling CloudTrail, Config, GuardDuty, or Security Hub; deny leaving the organisation; deny deleting log archives; deny creation of unencrypted volumes or public S3 access changes; deny use of root user; and restrict expensive or unapproved services in sandbox accounts. Prefer a deny-list approach for simplicity. Test SCPs in a sandbox OU before wide rollout, since mistakes can break many accounts. Combine with Control Tower, centralised logging, and delegated administrator accounts.

Likely follow-ups

- Why don't SCPs apply to the management account?
- How do you roll out an SCP safely?

🗺 Organizations and SCPs



Org structure: OUs (Organizational Units) group accounts by purpose:

Root -> Security OU, Prod OU, Dev OU, Sandbox OU

SCP example: deny leaving the org + restrict regions:

```
{"Effect": "Deny", "Action": "*", "Resource": "*", "Condition": {"StringNotEquals": {"aws:RequestedRegion": ["us-east-1", "eu-west-1"]}}}
```

```
aws organizations attach-policy --policy-id p-xxx --target-id ou-xxx
```

```
# Deny disabling CloudTrail / GuardDuty (guardrails):
```

```
"Action": ["cloudtrail:StopLogging", "guardduty:DeleteDetector"], "Effect": "Deny"
```

Multi-account is the AWS best practice for isolation: separate accounts per environment/team limit blast radius and simplify billing. SCPs are preventive guardrails applied to OUs/accounts — they set the maximum permissions even the account admin/root cannot exceed. Common SCPs: restrict regions, prevent disabling security services, block root usage, deny leaving the org. AWS Control Tower automates landing-zone setup with these guardrails.

SCPs do not grant: they only filter. They also do not apply to the management (payer) account. Test SCPs carefully — an overly broad Deny can lock out an entire OU. Use Control Tower / well-architected landing zones rather than hand-rolling.



Official docs

- [AWS Organizations: Service control policies](#)
- [AWS Organizations: What is AWS Organizations?](#)

Spoken answer · ~78 s

Detection and response in AWS rely on layered services that feed one workflow. CloudTrail records API activity, VPC flow logs record network traffic, and DNS logs and application logs add context. GuardDuty analyses those sources with threat intelligence and machine learning to detect suspicious behaviour: compromised credentials, unusual API calls, cryptomining, communication with known malicious IPs, and S3 or EKS-specific threats. Security Hub aggregates findings from GuardDuty, Inspector, Macie, Config, and others, normalises them, scores posture against standards like CIS and AWS Foundational Security Best Practices, and gives a single view across accounts. Config tracks resource configuration and flags drift and non-compliance. I enable these across the organisation with a delegated administrator account, and send findings to EventBridge. Response is partly automated: EventBridge rules trigger Lambda or Step Functions for containment, such as isolating an instance with a quarantine security group, disabling or revoking compromised credentials, snapshotting for forensics, and notifying the on-call team and ticketing system. For serious incidents, follow a runbook: contain, preserve evidence, eradicate, recover, and review. Use a dedicated security account and immutable logs. Practise with game days and tune findings to reduce noise.

Likely follow-ups

- How does EventBridge enable automated response?
- What does Security Hub add on top of GuardDuty?

```
# CloudTrail: record of every API call (the audit log – enable org-wide):
```

```
aws cloudtrail create-trail --name org-trail --s3-bucket-name audit-bucket --is-organization-trail
```

```
# GuardDuty: ML threat detection (crypto-mining, creds exfil, recon):
```

```
aws guardduty create-detector --enable
```

```
# Security Hub: aggregates findings + CIS/AWS best-practice checks:
```

```
aws securityhub enable-security-hub --enable-default-standards
```

```
# Auto-respond with EventBridge -> Lambda (e.g. isolate instance):
```

```
aws events put-rule --event-pattern '{"source":["aws.guardduty"]}'
```

Layered detection: CloudTrail is the foundational audit trail (who did what, when). GuardDuty analyzes CloudTrail, VPC Flow Logs, and DNS logs with ML to flag threats (compromised credentials, crypto-mining, unusual API calls). Security Hub centralizes findings across accounts and runs compliance standards (CIS,

PCI). Wire GuardDuty findings to EventBridge for automated response — quarantine an instance, revoke sessions, or alert.

Incident response in AWS: On a compromised credential alert, immediately revoke the session (attach a Deny-all policy / disable the key), snapshot for forensics, and rotate. CloudTrail tells you the blast radius (what the credential touched). Detective helps visualize the attack path.

Official docs

- [AWS CloudTrail](#)
- [AWS GuardDuty](#)
- [AWS Security Hub](#)

Q37 What is the difference between identity-based, resource-based, and ABAC policies?

IAM & Security

Spoken answer · ~85 s

These are three ways to define who can access what. Identity-based policies attach to a principal, such as a user, group, or role, and state what that principal may do on which resources. They're the most common, and work well when you manage permissions from the perspective of a person or workload. Resource-based policies attach to a resource, such as an S3 bucket, KMS key, SQS queue, or Lambda function, and state which principals may access it. They include a Principal element, and are essential for cross-account access, because they can grant access to a principal in another account directly. Attribute-based access control, ABAC, uses tags on principals and resources as the basis for decisions, with condition keys such as `aws:PrincipalTag` and `aws:ResourceTag`. For example, allow a user to access resources only where the project tag matches their own project tag. The advantage of ABAC is scale: instead of writing a policy per project, a single policy handles all projects, and new resources get access just by tagging them, so permissions grow without constant policy edits. The risks are that tag governance becomes security-critical, so you must control who can set tags. I often combine RBAC for coarse roles with ABAC for fine-grained, scalable access.

Likely follow-ups

- Why are resource-based policies essential for cross-account access?
- What must you control when using ABAC?

```
# Identity-based: attached to a user/group/role (what THEY can do):
```

```
{"Effect":"Allow","Action":"s3:GetObject","Resource":"arn:aws:s3:::bucket/*"}
```

```
# Resource-based: attached to the RESOURCE (who can access IT) - has Principal:
```

```
{"Principal":{"AWS":"arn:aws:iam::222:root"},"Action":"s3:GetObject","Resource":"..."}
```

```
# ABAC: allow based on matching TAGS instead of listing every resource:
```

```
"Condition":{"StringEquals":{"aws:ResourceTag/team":"${aws:PrincipalTag/team}"}}
```

Identity-based policies attach to principals and grant them actions. Resource-based policies attach to resources (S3 buckets, SQS, Lambda, KMS) and include a Principal — they enable cross-account access without role assumption. ABAC (attribute-based access control) grants access by comparing tags on the principal and resource, so one policy scales to many resources/teams without editing it as you add resources.

ABAC at scale: Tag resources and principals with team/project, then write a single policy that allows access only when the tags match. New projects need no policy changes — just consistent tagging. This is the modern approach for large multi-team orgs versus exploding lists of resource ARNs.



Official docs

- [Amazon S3 User Guide](#)

 Spoken answer · ~80 s

I enforce S3 security with layers of preventive controls at the organisation level, plus detection. Preventive: turn on S3 Block Public Access at the account level in every account, and at the organisation level through policy, so no bucket can be made public by ACL or policy. Use SCPs to deny actions that disable Block Public Access or change the bucket's public settings. Enforce encryption by setting default bucket encryption, and use bucket policies or SCPs to deny uploads that don't use the required encryption, for example requiring SSE-KMS with an approved key. Enforce TLS with a bucket policy denying requests where `aws:SecureTransport` is false. Disable ACLs by setting object ownership to bucket owner enforced. Restrict access to the organisation using the `aws:PrincipalOrgID` condition, and restrict network access to VPC endpoints where appropriate. Standardise buckets through infrastructure-as-code modules or service catalogue products that bake these settings in. Detective: AWS Config rules and Security Hub controls flag non-compliant buckets; Access Analyzer for S3 identifies external or public access; Macie discovers sensitive data; CloudTrail data events and server access logs provide audit. Alert and auto-remediate with EventBridge and Lambda or Config remediation. Review exceptions formally.

Likely follow-ups

- How do you stop anyone from disabling Block Public Access?
- Which services detect publicly exposed buckets?

```
# Account-level Block Public Access (the master switch):
```

```
aws s3control put-public-access-block --account-id 123 --public-access-block-configuration
BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=tru
e
```

```
# Default encryption (SSE-KMS) on a bucket:
```

```
aws s3api put-bucket-encryption --bucket b --server-side-encryption-configuration
\'{"Rules":[{"ApplyServerSideEncryptionByDefault":{"SSEAlgorithm":"aws:kms"}}]\'
```

```
# Bucket policy: deny any unencrypted PUT or non-TLS access:
```

```
{"Effect":"Deny","Action":"s3:PutObject","Condition":{"StringNotEquals":{"s3:x-amz-server-
side-encryption":"aws:kms"}}}
```

```
{"Effect":"Deny","Action":"s3:*","Condition":{"Bool":{"aws:SecureTransport":"false"}}}
```

Defense in depth for S3: enable account-level Block Public Access (overrides any bucket-level mistake), set default encryption, and add bucket policies that deny unencrypted uploads and non-TLS requests. Enforce org-wide with an SCP requiring Block Public Access and via IAM Access Analyzer to flag any bucket exposed externally.

The classic breach: a public S3 bucket leaking data. Account-level Block Public Access is the single most important control — it stops a misconfigured bucket policy or ACL from ever making data public, regardless of what someone sets later. Combine with Access Analyzer alerts.

Official docs

- [Amazon S3 User Guide](#)

Q39 How does AWS WAF and Shield protect applications? When do you use each?

IAM & Security

Spoken answer · ~83 s

WAF and Shield address different threat layers. AWS WAF works at layer seven, inspecting HTTP and HTTPS requests against rules to block application attacks like SQL injection, cross-site scripting, malicious bots, and abusive request rates. It attaches to CloudFront, ALBs, API Gateway, and similar resources, and you use managed rule groups, custom rules, rate-based rules, and bot or fraud controls. Use it for any internet-facing web application or API. AWS Shield protects against DDoS attacks. Shield Standard is automatic and free for all customers, protecting against common network and transport layer attacks on AWS edge services. Shield Advanced is a paid service that adds enhanced detection and mitigation for larger and more sophisticated attacks across layers three, four, and seven, access to the Shield Response Team, cost protection against scaling charges during an attack, and WAF integration at no extra fee for protected resources. Use Advanced for business-critical, high-visibility internet-facing applications where downtime would be costly. They complement each other: Shield absorbs volumetric and protocol attacks, and WAF handles application-layer rules, including rate limiting for HTTP floods. Design for resilience too: CloudFront in front, Route 53, auto scaling, and limiting origin exposure. Monitor metrics and test your response.

Likely follow-ups

- What extra does Shield Advanced provide over Standard?
- Which layer does each service protect?

```
# WAF web ACL on ALB / CloudFront / API Gateway:
```

```
aws wafv2 create-web-acl --name app-acl --scope REGIONAL --default-action Allow={}
```

```
# Managed rule groups (OWASP top 10, bad inputs, bot control):
```

```
AWSManagedRulesCommonRuleSet, AWSManagedRulesSQLiRuleSet, AWSManagedRulesBotControl
```

```
# Rate-based rule (block IPs over N req / 5 min):
```

```
{"RateBasedStatement":{"Limit":2000,"AggregateKeyType":"IP"}}
```

Shield Advanced (DDoS) + cost protection:

aws shield subscribe-to-proactive-engagement

WAF is a Layer 7 firewall: it inspects HTTP requests and blocks SQLi, XSS, bad bots, and abusive IPs via managed and custom rules, including rate limiting. Shield Standard (free, automatic) protects against common L3/L4 DDoS. Shield Advanced adds enhanced L3-L7 DDoS protection, 24/7 DDoS response team, and cost protection against scaling-driven bills during an attack. Place WAF on CloudFront for global edge filtering.

Layering: CloudFront (edge) + WAF (L7 rules, rate limiting) + Shield (DDoS) + Security Groups (L3/4) gives defense in depth. For L7 floods, WAF rate-based rules and Bot Control are the right tool; for volumetric L3/4 attacks, Shield. Put the WAF as close to the edge (CloudFront) as possible to drop bad traffic early.

Official docs

- [AWS WAF](#)

Q40 A developer accidentally committed AWS access keys to a public GitHub repo. What do you do?

IAM & Security

Spoken answer · ~85 s

Treat leaked keys as compromised immediately, because bots scan GitHub within minutes. Priorities in order. First, contain: deactivate the access key, then delete it, in IAM right away; don't just delete the commit, because the key remains in history, forks, and caches. If the key belongs to a role session, revoke active sessions. Second, assess the blast radius: use CloudTrail to examine everything the key did since it was exposed, filtering by access key ID, looking for suspicious API calls, new IAM users or keys, new instances, especially in unusual regions, data access, and changes to security settings. Check billing for unexpected charges. Third, eradicate: remove any attacker-created resources, backdoor users, roles, access keys, and unusual Lambda functions or instances, and rotate any other credentials the key could have read. Fourth, clean the repository: remove the secret from history with git filter-repo or BFG, force-push, and ask GitHub to purge caches if needed, although you should assume it was exposed regardless. Fifth, contact AWS Support if abuse occurred; AWS often notifies and applies a quarantine policy. Afterwards, prevent recurrence: use roles and OIDC instead of long-lived keys, add pre-commit and CI secret scanning, enable GitHub secret scanning and push protection, and run a blameless review.

Likely follow-ups

- Why is deleting the commit not enough?
- How do you find what the compromised key did?

1. IMMEDIATELY deactivate/delete the key (do not wait):

```
aws iam update-access-key --access-key-id AKIA... --status Inactive --user-name dev
```

```
aws iam delete-access-key --access-key-id AKIA... --user-name dev
```

2. Scope the blast radius via CloudTrail:

```
aws cloudtrail lookup-events --lookup-attributes  
AttributeKey=AccessKeyId,AttributeValue=AKIA...
```

3. Check for attacker persistence (new users/keys/roles, EC2 instances):

```
aws iam list-users; aws ec2 describe-instances --query  
"Reservations[].Instances[].InstanceId"
```

Act in order: (1) revoke the credential first — minutes matter, attackers scrape public repos within seconds; (2) investigate what the key did via CloudTrail (it logs every API call by that key); (3) hunt for persistence the attacker may have created (new IAM users/keys, unexpected instances often spun up for crypto-mining, modified trust policies); (4) rotate anything the key could have read; (5) remediate the root cause.

Prevent recurrence: Stop using long-lived keys — move to IAM roles, IRSA, and OIDC for CI. Add secret scanning (git-secrets, gitleaks, GitHub push protection) to block commits. AWS also auto-detects exposed keys and applies a quarantine policy, but never rely on that — assume compromise and respond fully.

Official docs

- [AWS IAM: Temporary credentials \(STS\)](#)

Q41 **How do Auto Scaling Groups work? Explain scaling policies, health checks, and lifecycle hooks.**

Compute

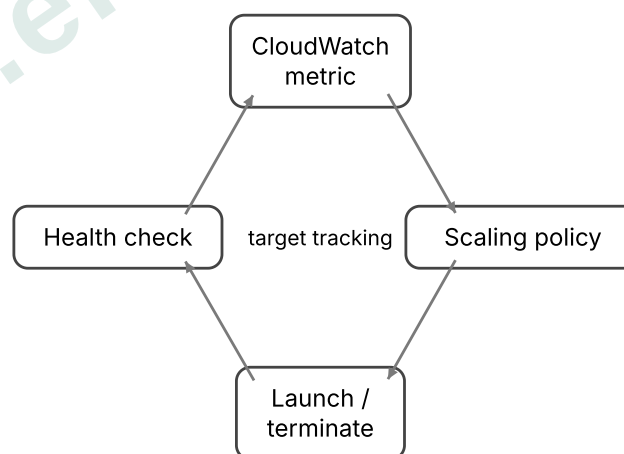
 Spoken answer · ~86 s

An Auto Scaling Group maintains a fleet of EC2 instances between a minimum and maximum size, launched from a launch template, spread across subnets in multiple zones. It replaces unhealthy instances and scales with demand. Scaling policies: target tracking is the simplest and usually best, for example keep average CPU at fifty percent or requests per target at a set value, and the group adjusts itself. Step scaling adds or removes capacity in steps based on alarm severity. Scheduled scaling handles predictable patterns like business hours, and predictive scaling uses forecasts. Cooldowns and instance warm-up prevent flapping. Health checks: EC2 status checks by default, plus ELB health checks when attached to a load balancer, so instances failing application checks are replaced. Set a health check grace period long enough for boot. Lifecycle hooks pause an instance in a pending or terminating state so you can run custom actions: on launch, finish configuration or register with a system; on termination, drain connections, upload logs, or deregister, before continuing or timing out. Other features: instance refresh for rolling AMI updates, warm pools to cut startup time, mixed instance policies with spot, and termination policies. I distribute across at least three zones and size the minimum for a zone failure.

Likely follow-ups

- Why use target tracking over step scaling?
- What would you do in a termination lifecycle hook?

 Auto Scaling



ASG with a launch template, spread across AZs:

```
aws autoscaling create-auto-scaling-group --auto-scaling-group-name web --launch-template
LaunchTemplateName=web-lt --min-size 2 --max-size 10 --desired-capacity 3 --vpc-zone-
identifier "subnet-a,subnet-b,subnet-c"
```

Target tracking (keep avg CPU at 50%):

```
aws autoscaling put-scaling-policy --policy-type TargetTrackingScaling --target-tracking-configuration \"{'PredefinedMetricSpecification': {'PredefinedMetricType': 'ASGAverageCPUUtilization'}, 'TargetValue': 50}\"
```

Use ELB health checks (not just EC2):

```
aws autoscaling update-auto-scaling-group --auto-scaling-group-name web --health-check-type ELB --health-check-grace-period 120
```

ASGs maintain desired capacity, replace unhealthy instances, and scale on policies. Target tracking (keep a metric at a target) is the simplest and preferred over step scaling for most cases. Set health-check-type to ELB so instances failing the load balancer health check (not just crashed) get replaced. Lifecycle hooks pause launch/terminate so you can run setup or drain connections before an instance is added/removed.

Production tips: Spread across 3 AZs for resilience; use a warm-up/grace period so new instances are not killed before the app starts; combine On-Demand + Spot via mixed instances policy for cost; and use instance refresh for rolling AMI updates without downtime.



Official docs

- [Amazon EC2 Auto Scaling](#)

Q42 **What is AWS Lambda — explain cold starts, concurrency, and when serverless fits.**

Compute

🗣 Spoken answer · ~82 s

Lambda runs code in response to events without managing servers; you pay per invocation and duration. Each execution environment is a container-like sandbox. A cold start happens when Lambda must create a new environment, download code, start the runtime, and run initialisation code, adding latency from tens of milliseconds to a few seconds, worse for large packages, VPC attachment in older setups, and heavy runtimes like the JVM. Mitigations: smaller packages, lazy loading, lighter runtimes, SnapStart for Java, and provisioned concurrency, which keeps environments warm at a cost. Concurrency is the number of simultaneous executions. There's an account-level limit, reserved concurrency guarantees and caps a function's share, and provisioned concurrency pre-warms. Bursts scale quickly but have rate limits, and downstream systems like databases can be overwhelmed, so cap concurrency and use RDS Proxy. Serverless fits event-driven, spiky, or unpredictable workloads, glue code, scheduled jobs, API backends with moderate traffic, and file or stream processing. It fits poorly for steady high-throughput workloads where containers are cheaper, very latency-sensitive paths intolerant of cold starts, long-running jobs beyond the fifteen-minute limit, and workloads needing specialised hardware or persistent connections. Design functions to be idempotent, handle retries, and use dead-letter queues.

Likely follow-ups

- How does provisioned concurrency differ from reserved concurrency?
- Why does Lambda scaling threaten downstream databases?

Concurrency controls:

```
aws lambda put-function-concurrency --function-name api --reserved-concurrent-executions 100
```

Provisioned concurrency to eliminate cold starts on hot paths:

```
aws lambda put-provisioned-concurrency-config --function-name api --qualifier prod --provisioned-concurrent-executions 20
```

Tune memory (also scales CPU) for price/performance:

```
aws lambda update-function-configuration --function-name api --memory-size 1024 --timeout 30
```

Lambda runs code without managing servers, scaling automatically per request and billing per ms. A **cold start** is the latency when a new execution environment spins up (init + your init code); it hits the first request and during scale-out. Mitigate with provisioned concurrency on latency-sensitive paths, smaller packages, and lightweight runtimes. Account concurrency is shared — use reserved concurrency to protect critical functions and cap others.

When NOT to use Lambda: long-running jobs (15-min cap), very high steady throughput (can be costlier than containers), workloads needing persistent connections, or heavy/low-latency cold-start-sensitive paths without provisioned concurrency. Watch out for downstream limits — a Lambda scaling to thousands of concurrent executions can overwhelm an RDS connection pool (use RDS Proxy).

Memory = CPU: Lambda CPU scales with memory, so raising memory can make a function finish faster AND cheaper. Use AWS Lambda Power Tuning to find the sweet spot.

Official docs

- [AWS Lambda: Concurrency](#)
- [AWS Lambda Developer Guide](#)

Q43 Compare ECS, EKS, and Fargate. How do you choose a container platform on AWS?

Compute

Spoken answer · ~81 s

The three overlap, so I separate orchestration from compute. ECS is AWS's own container orchestrator: simple, deeply integrated with IAM, load balancers, and CloudWatch, with a small learning curve and no control plane fee. EKS is managed Kubernetes: more powerful and portable, with a vast ecosystem, custom resources, and operators, but with more complexity and a control plane cost. Fargate is not an orchestrator but a serverless compute option that works with both ECS and EKS, removing node management. So the real choices are orchestrator, ECS or EKS, and compute, EC2 or Fargate. Choose ECS on Fargate for teams that want the simplest path to run containers on AWS with minimal operations. Choose EKS when you need Kubernetes' ecosystem, multi-cloud or hybrid portability, existing Kubernetes skills, advanced scheduling, service meshes, or operators. Choose EC2 compute when you need GPUs, daemonsets, special networking, or want cost efficiency at scale through bin-packing and spot. Fargate suits spiky or small workloads and teams wanting less operational burden. Consider team skills, hiring, workload complexity, cost, and portability. Avoid choosing Kubernetes for fashion; the operational cost is real. Many organisations use ECS for simple services and EKS for the platform.

Likely follow-ups

- Is Fargate an alternative to ECS or a compute option?
- When is ECS the better choice than EKS?

ECS / EKS / Fargate

ECS and EKS on EC2

Fargate

You manage nodes

Serverless tasks or pods

More control, lower unit cost

No node ops

```
# ECS on Fargate (no nodes to manage):
```

```
aws ecs create-service --cluster prod --service-name api --task-definition api:3 --desired-count 3 --launch-type FARGATE
```

```
# ECS on EC2 (you manage the instances, cheaper at scale):
```

```
aws ecs create-cluster --cluster-name prod # + EC2 capacity provider
```

```
# EKS = managed Kubernetes (portable, ecosystem):
```

```
eksctl create cluster --name prod --fargate
```

ECS is AWS-native, simpler, tightly integrated, no Kubernetes knowledge needed — great when you are all-in on AWS and want low operational overhead. EKS is managed Kubernetes — choose it for portability, the CNCF ecosystem (Helm, operators, service mesh), or existing K8s skills/workloads. Fargate is a *launch type* (works with both ECS and EKS): serverless containers, no node management, you pay per task vCPU/memory.

Decision guide: Want simplest + AWS-only → ECS. Need Kubernetes portability/ecosystem or multi-cloud → EKS. Do not want to manage nodes/patching → Fargate (with either). Steady high scale where node bin-packing saves money → EC2 launch type. Many teams run EKS + Fargate for spiky/isolated workloads and managed node groups for the rest.

Official docs

- [Amazon EKS: User guide](#)
- [Amazon ECS Developer Guide](#)

Q44 How do you build immutable, hardened machine images and keep an AMI pipeline current?

Compute

🗣 Spoken answer · ~85 s

Immutable, hardened images mean you build a new AMI for every change instead of patching running servers. A pipeline: start from a trusted base, such as the latest Amazon Linux or a CIS-hardened AMI; use Packer or EC2 Image Builder to launch a temporary instance, apply configuration through scripts or Ansible, install agents and dependencies, apply hardening such as CIS benchmarks, disable unneeded services, set secure SSH and kernel parameters, and remove secrets and temporary files. Then run tests and scans: Inspector for vulnerabilities, compliance checks, and smoke tests that boot the image. Tag the AMI with version, commit, and build date, and copy and share it across regions and accounts, with encryption. Keep it current by triggering rebuilds on a schedule, such as weekly, and when the base image updates or a critical CVE appears; publish the latest AMI ID to Parameter Store so launch templates and Terraform reference it without hardcoding. Roll out through Auto Scaling instance refresh, or by Karpenter drift on EKS, so the fleet turns over gradually with health checks. Deprecate and delete old AMIs with lifecycle policies. Benefits: consistency, faster boots, easy rollback, and no configuration drift. Restrict who can launch unapproved AMIs with SCPs or approved-AMI rules.

Likely follow-ups

- Why prefer immutable images to patching in place?
- How do launch templates pick up the newest AMI?

```
# EC2 Image Builder pipeline (build + test + distribute AMIs):
```

```
aws imagebuilder create-image-pipeline --name golden --image-recipe-arn arn:... --
infrastructure-configuration-arn arn:...
```

```
# Or Packer to bake an AMI from a base + provisioning:
```

```
packer build app.pkr.hcl # installs deps, agents, hardening, app
```

```
# Reference the AMI by SSM parameter (auto-updates launch templates):
```

```
aws ssm get-parameter --name /golden/ami/latest --query Parameter.Value
```

Immutable infrastructure: bake a versioned golden AMI with the OS hardened (CIS benchmark), patches, agents (CloudWatch, SSM, security), and the app, then deploy by replacing instances — never patch in place. EC2 Image Builder or Packer automate the bake; Image Builder can run on a schedule to pull the latest patched base, run tests, and distribute across regions/accounts.

Keeping current: Schedule the pipeline to rebuild from the newest patched base AMI, store the resulting AMI ID in an SSM parameter, and have launch templates reference that parameter so an instance refresh rolls out the new image. This gives a repeatable, auditable patch cadence without snowflake servers.

Q45 How do Spot Instances work and how do you run production workloads on them safely?

Compute

 Spoken answer · ~80 s

Spot instances are spare EC2 capacity at up to roughly ninety percent below on-demand, but AWS can reclaim them with a two-minute interruption notice. They suit fault-tolerant, flexible workloads. To run production safely, design for interruption. Diversify: use many instance types, sizes, and availability zones, with allocation strategies such as price-capacity-optimized, so a shortage in one pool doesn't remove everything. Use Auto Scaling groups with mixed instance policies, or Karpenter, which handles spot selection and interruption. Keep a baseline on on-demand or Savings Plans for critical capacity, and use spot for the burst portion; for example, a stateless web tier at seventy percent spot with an on-demand floor. Handle the interruption notice: watch instance metadata or EventBridge events, drain connections, checkpoint work, deregister from load balancers, and let Kubernetes reschedule pods, using the node termination handler. Make workloads stateless and idempotent, with state in external stores, and use queues so interrupted jobs are retried. Use PodDisruptionBudgets and multiple replicas spread across zones. Avoid spot for singleton stateful services, databases, or latency-critical single points. Monitor interruption rates and capacity-rebalance recommendations, and test by simulating interruptions. Done well, spot cuts cost dramatically without hurting availability.

Likely follow-ups

- How does diversification reduce spot interruption risk?
- What do you do in the two-minute interruption window?

```
# Mixed instances policy: baseline On-Demand + Spot for the rest:
```

```
aws autoscaling create-auto-scaling-group --mixed-instances-policy
\'{"InstancesDistribution":
{"OnDemandBaseCapacity":2,"OnDemandPercentageAboveBaseCapacity":20,"SpotAllocationStrategy"
:"price-capacity-optimized"},"LaunchTemplate":{"Overrides":[{"InstanceType":"m6i.large"},
{"InstanceType":"m5.large"}, {"InstanceType":"m5a.large"}]}}\'
```

```
# Handle the 2-minute interruption notice (IMDS or EventBridge):
```

```
curl http://169.254.169.254/latest/meta-data/spot/instance-action
```

Spot uses spare EC2 capacity at up to ~90% off but can be reclaimed with a 2-minute warning. Run them safely by: diversifying across many instance types and AZs (price-capacity-optimized strategy reduces interruptions), keeping workloads stateless/fault-tolerant, and handling the interruption notice to drain gracefully. Ideal for CI runners, batch, big-data, and stateless web tiers behind an ASG.

Do not run on Spot: stateful singletons, databases, or anything that cannot tolerate sudden termination. For EKS, use the AWS Node Termination Handler (or Karpenter, which natively handles Spot interruptions and consolidation) to cordon and drain nodes on the 2-minute notice.

Official docs

- [Amazon EC2 User Guide](#)

Q46 What is Karpenter and how does it improve EKS node autoscaling vs Cluster Autoscaler?

Compute

Spoken answer · ~80 s

Karpenter is a node provisioner for Kubernetes that creates EC2 instances directly in response to pending pods. The Cluster Autoscaler works by adjusting the desired size of pre-defined Auto Scaling groups, so you must predefine node groups per instance type and shape, scaling is slower because it goes through the ASG, and bin-packing is limited by the available groups. Karpenter reads pod requirements, such as CPU, memory, architecture, zone, and taints, then calls the EC2 fleet API to launch the best-fit instance from a wide choice of types, including spot, in seconds to minutes. You configure NodePools with constraints and EC2NodeClasses with AMI, subnets, and security groups. Benefits: faster scaling, less configuration, higher utilisation through right-sized nodes, lower cost by choosing cheaper instance types and spot, and consolidation that removes underused nodes or replaces them with cheaper ones. Drift detection and node expiry roll nodes to new AMIs, helping with patching. Operationally, set accurate pod requests, use PodDisruptionBudgets to control consolidation disruption, set NodePool limits to cap spend, and keep Karpenter's own pods on stable capacity. It works best for diverse, bursty workloads; for very steady uniform workloads, the advantage narrows.

Likely follow-ups

- How does Karpenter choose an instance type?
- What is drift in Karpenter?

```
# Karpenter provisioner/NodePool: pick instances just-in-time by pod needs:
```

```
apiVersion: karpenter.sh/v1 kind: NodePool
```

```
  requirements: [karpenter.k8s.aws/instance-category In [c,m,r], karpenter.sh/capacity-type
In [spot,on-demand]]
```

```
  disruption: consolidationPolicy: WhenEmptyOrUnderutilized
```

```
# Karpenter watches unschedulable pods and launches the cheapest fitting node.
```

Cluster Autoscaler scales fixed node groups (one instance type each) by adjusting ASG desired count — slower and coarse. Karpenter watches pending pods and provisions *right-sized* nodes directly from EC2 in seconds, choosing instance type, size, and Spot/On-Demand to fit the actual pod requirements. It also

consolidates — repacking pods and terminating underutilized nodes to cut cost.

Why it wins: faster scale-up (no pre-defined node groups), better bin-packing and cost via diverse instance selection + Spot, and automatic consolidation. The trade-off is more dynamic node churn, so set Pod Disruption Budgets and use do-not-disrupt annotations for sensitive workloads. Karpenter is now the recommended autoscaler for most EKS clusters.

Official docs

- [Kubernetes Autoscaler: Cluster Autoscaler](#)
- [Amazon EKS best practices guide](#)
- [Karpenter documentation](#)

Q47 How do you design a highly available, multi-AZ architecture for a stateless web app?

Compute

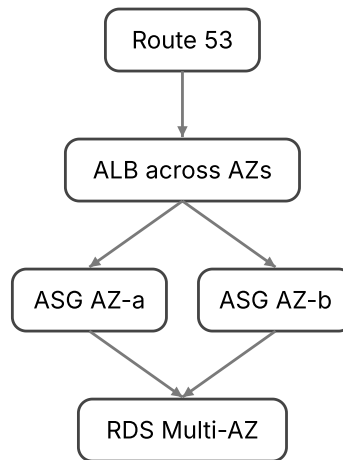
Spoken answer · ~85 s

For a stateless web app, high availability means no single point of failure at any layer. Run the application across at least two, preferably three, availability zones. Put an Application Load Balancer in front, spanning public subnets in each zone, with cross-zone load balancing and health checks. Run instances or containers in private subnets, managed by an Auto Scaling group, ECS service, or Kubernetes deployment that spreads replicas across zones, with a minimum that survives the loss of one zone and still carries peak load. Because the application is stateless, any instance can serve any request: store sessions in ElastiCache or DynamoDB, uploads in S3, and configuration externally. Use a multi-AZ database, such as RDS Multi-AZ or Aurora, with automatic failover, and a replicated cache. Use one NAT gateway per zone so outbound traffic survives a zone loss. Put CloudFront and Route 53 at the edge, with health checks. Add autoscaling for demand, graceful shutdown, readiness probes, and connection draining for smooth deploys. Define timeouts, retries, and circuit breakers for dependencies. Test failure: terminate instances, simulate a zone outage with chaos experiments, and confirm capacity. Monitor SLOs, and codify everything in infrastructure as code. Consider multi-region only if the business requirement demands it.

Likely follow-ups

- Why keep the application stateless for HA?
- How do you size capacity to survive a zone loss?

Multi-AZ design



ALB across 3 AZs -> ASG instances in private subnets in each AZ:

Route 53 -> CloudFront -> ALB (3 AZs) -> ASG (min spread per AZ) -> RDS Multi-AZ

Health checks at every layer:

```
aws elbv2 modify-target-group --target-group-arn tg --health-check-path /healthz
```

Stateless: session/state in ElastiCache or DynamoDB, not on the instance

HA pattern: front with Route 53 + CloudFront, an ALB spanning at least 3 AZs, instances in an ASG spread across those AZs (so losing one AZ keeps you serving), and a Multi-AZ data tier (RDS Multi-AZ / DynamoDB). Keep the app tier stateless — store session and shared state in ElastiCache or DynamoDB so any instance can serve any request and instances are freely replaceable.

Surviving an AZ failure: design so the remaining AZs have enough headroom to absorb the load (do not run at 100% across 3 AZs, or losing one overloads the other two). Health checks at each layer + ASG self-healing + Multi-AZ data give automatic recovery. This is the canonical AWS Well-Architected reliability design.

Official docs

- [Amazon RDS: Multi-AZ deployments](#)

Q48 **What deployment strategies does AWS support (blue/green, canary, rolling) and how?**

Compute

 Spoken answer · ~82 s

AWS supports several deployment strategies, and the right one depends on risk tolerance and architecture. Rolling replaces instances or pods gradually, a few at a time, keeping the service up; it's simple and built into ECS, Kubernetes, and Auto Scaling instance refresh, but both versions run together briefly and rollback is slower. Blue/green runs two full environments; you deploy to the idle one, test it, then shift traffic, and roll back by switching back. CodeDeploy supports blue/green for EC2, ECS, and Lambda, using load balancer target groups; it costs double capacity temporarily but gives fast rollback. Canary sends a small percentage of traffic, say five or ten percent, to the new version first, watches metrics, and then proceeds: CodeDeploy offers canary and linear configurations for Lambda and ECS, and weighted target groups or Route 53 weighted records work too, as do Kubernetes tools like Argo Rollouts. Feature flags decouple deployment from release. Whichever is used, add automated validation with alarms that trigger automatic rollback, make database changes backward compatible using expand-contract, keep graceful shutdown and connection draining, and track deployment metrics. For high-risk changes, canary with automated analysis is my preference; for simple services, rolling is enough.

Likely follow-ups

- How does a canary deployment limit risk?
- Why must database changes be backward compatible during deployments?

```
# CodeDeploy blue/green for ECS / Lambda / EC2:
```

```
aws deploy create-deployment --deployment-group-name bg --deployment-config-name  
CodeDeployDefault.ECSCanary10Percent5Minutes
```

```
# Lambda weighted alias (canary 10% -> 100%):
```

```
aws lambda update-alias --function-name api --name prod --function-version 5 --routing-  
config AdditionalVersionWeights={"4":0.9}
```

```
# ALB weighted target groups for traffic shifting:
```

```
aws elbv2 modify-listener --default-actions ...ForwardConfig with weights
```

Rolling: replace instances/tasks in batches (built into ASG/ECS) — simple, no extra cost, but mixed versions briefly run together. Blue/green: stand up a full new (green) environment, shift traffic, keep blue for instant rollback — zero-downtime and safe, at the cost of double resources during cutover. Canary: shift a small % of traffic to the new version, watch metrics, then ramp — catches issues with minimal blast radius.

Choosing: Use canary (CodeDeploy/Lambda weighted aliases) for risky changes where you want metric-gated rollout and automatic rollback on CloudWatch alarms. Use blue/green when you need instant rollback and can

afford duplicate capacity. Rolling is fine for low-risk, backward-compatible changes. Always pair with health checks and automatic rollback alarms.

Official docs

- [Kubernetes: Deployments](#)
- [Google SRE Workbook: Canarying Releases](#)
- [Martin Fowler: CanaryRelease](#)

Q49 How does EC2 instance metadata (IMDS) work and why is IMDSv2 a security requirement?

Compute

Spoken answer · ~86 s

The instance metadata service is an endpoint, 169.254.169.254, reachable only from within the instance, that provides information about it: instance ID, network details, and, importantly, temporary IAM role credentials for the instance profile. Applications and the SDKs call it to get credentials automatically. IMDSv1 is a simple request-response with no authentication, so any process that can make an HTTP request from the instance, including through a server-side request forgery vulnerability in an application, can read the credentials. This was the pattern in major breaches. IMDSv2 requires a session-oriented flow: the client first makes a PUT request to obtain a token, with a time-to-live, then includes the token in a header on later requests. SSRF attacks usually can't issue PUT requests with custom headers, and the response by default has a hop limit of one, so proxied or forwarded requests from containers fail. So IMDSv2 mitigates credential theft. Best practice: require IMDSv2 on all instances by setting `HttpTokens` to `required` in launch templates, with an account-level default; use an SCP or Config rule to enforce it; set the hop limit to one, or two where containers need access; and for EKS prefer pod-level identity so pods don't need the node's role. Also minimise the permissions on instance roles.

Likely follow-ups

- Why does the PUT token step defeat most SSRF attacks?
- What does the hop limit protect?

IMDS versions

IMDSv1	IMDSv2
Plain GET, SSRF-prone	Session token via PUT
Credentials can leak	Blocks simple SSRF

IMDSv1 (insecure): a single GET returns role credentials:

```
curl http://169.254.169.254/latest/meta-data/iam/security-credentials/role-name
```

```
# IMDSv2: must first PUT to get a session token, then use it:
```

```
TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 300")
```

```
curl -H "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest/meta-data/
```

```
# Enforce v2 + hop limit 1 (blocks containers/SSRF reaching IMDS):
```

```
aws ec2 modify-instance-metadata-options --instance-id i-xxx --http-tokens required --http-put-response-hop-limit 1
```

IMDS exposes instance data and, crucially, the IAM role's temporary credentials at 169.254.169.254. IMDSv1's simple GET means any SSRF vulnerability (tricking the app into making a request to that URL) can exfiltrate role credentials. IMDSv2 requires a PUT-issued, short-lived session token and a low hop limit, which defeats most SSRF and prevents containers from casually reaching the host's credentials.

This was the Capital One breach vector: an SSRF flaw fetched IMDSv1 credentials and accessed S3. Enforce IMDSv2 (`http-tokens required`) and a hop limit of 1 across the fleet via SCP/launch templates. It is one of the highest-impact, lowest-effort hardening steps in AWS.

Official docs

- [Amazon EC2 User Guide](#)

Q50 How do you patch and manage a fleet of EC2 instances at scale with Systems Manager?

Compute

 Spoken answer · ~83 s

Systems Manager provides fleet management through the SSM agent, which runs on EC2 instances and on-premises servers, communicating outbound only, so no inbound ports are needed. Instances need an instance profile with the managed policy, plus network access to SSM endpoints, perhaps through VPC endpoints. Patch Manager handles patching: define patch baselines that specify which classifications and severities to approve, with auto-approval delays; group instances using tags into patch groups; and schedule patching through maintenance windows with rate controls, such as patching a percentage at a time, with error thresholds that halt the rollout. It scans for compliance and reports missing patches. In production I patch in waves: dev first, then staging, then production zones, with health checks between. Run Command executes ad hoc commands across the fleet safely, without SSH. State Manager enforces desired configuration. Inventory collects software and configuration data. Session Manager gives audited shell access without keys or bastions. Automation runbooks orchestrate multi-step operations, such as snapshot before patching, patch, validate, and roll back on failure. For immutable fleets, the preferred approach is baking patched AMIs and replacing instances; SSM patching suits stateful or long-lived hosts. Report compliance, alert on failures, and track patch latency.

Likely follow-ups

- How does Session Manager improve on SSH bastions?
- How do you control the blast radius of a patching run?

```
# SSM needs the agent + an instance role (no SSH/keys required):
```

```
aws ssm describe-instance-information # which instances are managed
```

```
# Patch Manager: baseline + maintenance window:
```

```
aws ssm create-patch-baseline --name prod-baseline --approval-rules ...
```

```
aws ssm create-association --name AWS-RunPatchBaseline --schedule-expression "cron(0 3 ? * SUN *)"
```

```
# Run a command across the fleet (no SSH):
```

```
aws ssm send-command --document-name AWS-RunShellScript --targets Key=tag:env,Values=prod -  
-parameters commands="yum update -y"
```

```
# Session Manager: shell access without open SSH ports:
```

```
aws ssm start-session --target i-xxx
```

Systems Manager manages fleets without SSH: Patch Manager applies patch baselines on maintenance windows, Run Command executes scripts across tagged instances, State Manager enforces desired config, and Session Manager gives audited shell access through IAM (no bastion, no open port 22, full logging to CloudWatch/S3). Inventory and Compliance report what is installed and which instances are out of patch compliance.

Why SSM over SSH: no inbound ports, no key management, every action is IAM-controlled and logged in CloudTrail/Session logs. Combine with immutable AMIs for the base and SSM for emergency fixes/compliance. Tag-based targeting lets you patch `'env=prod'` fleet-wide on a schedule with automatic rollback reporting.

Official docs

- [Argo CD: ApplicationSet](#)
- [Amazon EC2 User Guide](#)

Q51 How do you connect VPCs and on-prem networks? Compare peering, Transit Gateway, VPN, Direct Connect.

Networking

Spoken answer · ~83 s

The options differ in scale, cost, and what they connect. VPC peering links two VPCs directly: cheap, simple, low latency, no bandwidth bottleneck, but it's one-to-one, non-transitive, CIDRs can't overlap, and routes must be added on both sides, so it gets unwieldy with many VPCs. Transit Gateway is a regional hub-and-spoke router: you attach VPCs, VPNs, and Direct Connect gateways, and manage routing centrally with route tables, enabling transitive routing, segmentation between environments, and inter-region peering. It scales operationally at a cost per attachment and per gigabyte. Site-to-Site VPN creates encrypted IPsec tunnels over the internet to on-premises; it's quick to set up and inexpensive, with bandwidth limited per tunnel and variable latency, and it's often used as a backup. Direct Connect provides a dedicated private fibre connection to AWS: consistent low latency, high bandwidth up to hundreds of gigabits, and lower data transfer cost, but with weeks of lead time and a higher fixed cost; for resilience use two connections at different locations. A common production design: Transit Gateway as the hub, Direct Connect as the primary link to on-premises with a VPN as backup, and PrivateLink for specific service exposure. Plan non-overlapping CIDRs from the start.

Likely follow-ups

- Why is peering non-transitive and what replaces it at scale?
- How do you make hybrid connectivity resilient?

```
# VPC Peering (1:1, non-transitive):
```

```
aws ec2 create-vpc-peering-connection --vpc-id vpc-a --peer-vpc-id vpc-b
```

```
# Transit Gateway (hub-and-spoke, transitive, scales to hundreds of VPCs):
```

```
aws ec2 create-transit-gateway; aws ec2 create-transit-gateway-vpc-attachment --transit-gateway-id tgw --vpc-id vpc-a
```

Site-to-Site VPN (encrypted over internet):

```
aws ec2 create-vpn-connection --type ipsec.1 --customer-gateway-id cgw --transit-gateway-id tgw
```

Direct Connect = dedicated private fiber (consistent low latency)

VPC Peering is simple and cheap but non-transitive (A-B and B-C does not give A-C) and becomes a mesh nightmare beyond a few VPCs. Transit Gateway is a central hub that connects many VPCs and on-prem transitively — the standard for multi-VPC/multi-account at scale. Site-to-Site VPN gives encrypted connectivity over the internet (quick, variable latency). Direct Connect provides a dedicated physical link for consistent low latency and high throughput.

Choosing: 2-3 VPCs → peering. Many VPCs/accounts → Transit Gateway. On-prem with cost sensitivity / quick setup → VPN; on-prem needing predictable performance/bandwidth → Direct Connect (often with a VPN backup for resilience). PrivateLink is the right answer when you only need to expose a single service, not whole-network routing.

Official docs

- [Cloudflare Learning: What is a VPN?](#)
- [AWS Transit Gateway](#)
- [AWS Direct Connect](#)

Spoken answer · ~83 s

PrivateLink lets you expose or consume a specific service privately across VPCs or accounts without exposing the whole network. The provider puts the service behind a Network Load Balancer and creates an endpoint service; consumers create an interface endpoint, which places elastic network interfaces with private IPs in their subnets. Traffic flows over the AWS network, never the public internet, and the consumer only reaches that one service. Versus VPC peering: peering connects two entire networks, with full routing between them, requires non-overlapping CIDRs, and is bidirectional; PrivateLink is one-directional and service-specific, works even with overlapping CIDRs, and can serve thousands of consumers, with the provider controlling which accounts may connect. Peering is better when two environments need broad two-way connectivity. Versus NAT: a NAT gateway lets private resources initiate outbound connections to the internet, translating addresses, and you pay for data processing; PrivateLink provides private access to a particular service without internet routing. AWS services themselves are reachable via interface endpoints, and S3 and DynamoDB via gateway endpoints, which are free and avoid NAT costs. Use PrivateLink for SaaS integrations, shared internal services across accounts, and security-sensitive access paths. Mind the per-endpoint hourly cost and per-gigabyte charges.

Likely follow-ups

- When would overlapping CIDRs force you to choose PrivateLink?
- How do gateway endpoints reduce NAT costs?

```
# Interface VPC endpoint (PrivateLink) to an AWS service or partner service:
```

```
aws ec2 create-vpc-endpoint --vpc-endpoint-type Interface --service-name com.amazonaws.us-east-1.secretsmanager --subnet-ids subnet-a --security-group-ids sg-x
```

```
# Gateway endpoint (S3/DynamoDB only, free, route-table based):
```

```
aws ec2 create-vpc-endpoint --vpc-endpoint-type Gateway --service-name com.amazonaws.us-east-1.s3 --route-table-ids rtb-x
```

```
# Expose YOUR service to other VPCs via an endpoint service (NLB-backed):
```

```
aws ec2 create-vpc-endpoint-service-configuration --network-load-balancer-arns arn:...
```

PrivateLink exposes a specific service (an AWS service, or your own NLB-fronted service) as a private endpoint inside your VPC — traffic never traverses the internet, and there is no network-level route between the two VPCs (unlike peering, which joins whole address spaces). It avoids NAT Gateway data-processing costs for AWS API calls and prevents CIDR-overlap problems since only the one service is reachable.

Endpoints save money + secure traffic: Use Gateway endpoints (free) for S3/DynamoDB, Interface endpoints for other AWS APIs so private subnets reach AWS services without a NAT Gateway. Use a PrivateLink endpoint

service to offer your SaaS to customers privately. The key contrast: peering = whole-VPC routing; PrivateLink = one-service exposure, no overlap concerns.

Official docs

- [RFC 3022: Traditional NAT](#)
- [Amazon VPC: VPC peering](#)
- [AWS PrivateLink](#)

Q53 How does Route 53 work? Explain routing policies and health-check based failover.

Networking

Spoken answer · ~85 s

Route 53 is AWS's DNS service with hosted zones, health checks, and traffic routing. Records can be simple, or use routing policies. Weighted splits traffic by percentage, good for canaries and migrations. Latency-based routes users to the region with the lowest latency. Failover has a primary and secondary: Route 53 serves the primary while its health check passes, and switches to the secondary when it fails. Geolocation routes by user country or continent, useful for compliance; geoproximity uses distance and bias; multivalue answer returns up to eight healthy records. Health checks probe endpoints over HTTP, HTTPS, or TCP from multiple global locations, with a configurable interval and failure threshold; they can also monitor other health checks or CloudWatch alarms, which lets private resources participate. For failover I use alias records pointing to load balancers with evaluate target health, set TTLs low, such as sixty seconds, so clients pick up changes quickly, though some resolvers cache longer. Design for the failure mode: a secondary region needs capacity and data ready, or failover just moves the outage. Combine policies, for example latency to regions with failover inside each. Test failover regularly, use private hosted zones for internal names, and monitor health check status with alarms.

Likely follow-ups

- What does evaluate target health do on an alias record?
- Why can failover take longer than the TTL suggests?

Routing policies:

Simple – one record, no logic

Weighted – split traffic by % (canary, A/B, gradual migration)

Latency – route to the lowest-latency region for the user

Geolocation/Geoproximity – route by user location

Failover – primary/secondary with health checks

```
Multivalue      - return multiple healthy IPs (simple client-side LB)
```

```
# Health check driving failover:
```

```
aws route53 create-health-check --health-check-config  
Type=HTTPS,ResourcePath=/healthz,FullyQualifiedDomainName=api.example.com
```

Route 53 is DNS plus traffic management. Weighted records enable canary releases and blue/green at the DNS layer. Latency-based routing sends users to the nearest/fastest region. Failover routing pairs a primary and secondary with health checks so DNS stops returning the unhealthy endpoint. Alias records point to AWS resources (ALB, CloudFront, S3) for free and can sit at the zone apex (where CNAMEs cannot).

Multi-region active-passive: Failover routing + health checks gives automatic regional failover; combine with latency routing for active-active. Remember DNS failover is bounded by TTL and client caching, so it is minutes-scale — pair with in-region (ALB/ASG) health for fast local recovery. Use ALIAS (not CNAME) at the apex for ALB/CloudFront.

Official docs

- [Amazon Route 53](#)
- [Route 53: Routing policies](#)

Q54 What is CloudFront and how do you use it for performance and security?

Networking

Spoken answer · ~84 s

CloudFront is AWS's content delivery network, caching and serving content from edge locations close to users. For performance: it caches static assets and cacheable API responses at the edge, reducing latency and origin load; it keeps persistent, optimised connections to the origin; supports HTTP/2 and HTTP/3, compression, and TLS termination close to the user; and can serve dynamic content faster through the AWS backbone. Configure cache behaviours per path, set cache policies using TTLs and which headers, cookies, and query strings form the cache key, since too many key variations lower hit rates, and use origin shield to consolidate requests to the origin. Invalidate or version asset filenames when content changes. For security: integrate AWS WAF and Shield for application and DDoS protection; enforce HTTPS with ACM certificates and modern TLS; use Origin Access Control so S3 buckets accept requests only from CloudFront; restrict ALB origins by custom headers or managed prefix lists; use signed URLs or cookies for private content; geo-restriction; and field-level encryption. Edge compute through CloudFront Functions and Lambda@Edge handles redirects, header manipulation, authentication, and A/B testing. Monitor cache hit ratio, error rates, and origin latency, and enable logging. It also reduces data transfer cost from the origin.

Likely follow-ups

- What makes a cache key too broad?
- How do you stop users bypassing CloudFront to reach the origin?

```
# Distribution with an S3/ALB origin + OAC (no public bucket):
```

```
aws cloudfront create-distribution --origin-domain-name bucket.s3.amazonaws.com
```

```
# Cache behaviors by path (static long-cache, /api no-cache):
```

```
/static/* -> cache 1y; /api/* -> forward to ALB, cache disabled
```

```
# Invalidate after a deploy:
```

```
aws cloudfront create-invalidation --distribution-id E123 --paths "/*"
```

```
# Edge logic with CloudFront Functions / Lambda@Edge (auth, redirects, headers)
```

CloudFront is a global CDN: it caches content at edge locations near users (lower latency, less origin load) and terminates TLS at the edge. Use Origin Access Control so the S3 origin is private and only CloudFront can read it. Per-path cache behaviors let you cache static assets aggressively while bypassing cache for dynamic APIs. It is also the best place to attach WAF and Shield for edge security and to add headers/redirects/auth via CloudFront Functions or Lambda@Edge.

Performance + security combo: CloudFront (edge cache + TLS) + WAF (L7 rules) + Shield (DDoS) + OAC (private origin) is the standard edge stack. Set proper Cache-Control headers and use versioned asset filenames so you rarely need invalidations (which cost money and take time).



Official docs

- [Amazon CloudFront](#)

 Spoken answer · ~87 s

The three load balancers operate at different layers. The Application Load Balancer works at layer seven for HTTP, HTTPS, and gRPC: it routes on host, path, headers, and query strings, supports redirects and fixed responses, WebSockets, authentication through OIDC or Cognito, WAF integration, and sends to instances, IPs, or Lambda. Use it for web applications, microservices, and Kubernetes ingress. The Network Load Balancer works at layer four for TCP, UDP, and TLS: extremely high throughput and low latency, preserves the client source IP, provides static IPs and elastic IPs per zone, and handles millions of requests per second. Use it for non-HTTP protocols, ultra-low latency needs, static IP requirements, PrivateLink services, and fronting other targets. The Gateway Load Balancer works at layer three and combines a transparent network gateway with load balancing, using GENEVE encapsulation to send traffic to fleets of virtual appliances, such as firewalls, intrusion detection, and deep packet inspection, scaling them horizontally and maintaining flow symmetry. Use it when inserting third-party security appliances in the traffic path. A quick selection: HTTP routing logic, choose ALB; raw performance, TCP/UDP, or fixed IPs, choose NLB; inline inspection appliances, choose GWLB. They can also be chained, such as NLB in front of ALB for static IPs with HTTP routing.

Likely follow-ups

- Why would you put an NLB in front of an ALB?
- What problem does Gateway Load Balancer solve?

 Load balancers

ALB (L7) / NLB (L4)	Choose by
HTTP routing / static IP, TCP	Protocol and features
Gateway LB: appliances	Latency, source IP, inspection

```
# ALB (L7 - HTTP/HTTPS, host/path routing, WebSockets, gRPC):
```

```
aws elbv2 create-load-balancer --type application --subnets subnet-a subnet-b
```

```
# NLB (L4 - TCP/UDP, ultra-low latency, static IP, millions of req):
```

```
aws elbv2 create-load-balancer --type network --subnets subnet-a subnet-b
```

```
# GWLB (insert appliances - firewalls/IDS - transparently):
```

```
aws elbv2 create-load-balancer --type gateway
```

ALB operates at Layer 7: routes by host/path/header, supports HTTPS termination, WebSockets, gRPC, redirects, and integrates with WAF and Cognito auth — the default for HTTP apps and Kubernetes Ingress. NLB operates at Layer 4: extreme performance/low latency, handles TCP/UDP, preserves source IP, provides static/Elastic IPs, and is used for non-HTTP protocols or when you need a fixed IP. Gateway Load Balancer transparently inserts third-party network appliances (firewalls, IDS/IPS) into the traffic path.

Quick rule: HTTP(S) with routing/auth/WAF → ALB. Non-HTTP, need static IP, or millions of connections at lowest latency → NLB. Need to run inline security appliances → GWLB. For EKS, the AWS Load Balancer Controller provisions ALBs for Ingress and NLBs for L4 Services.

Official docs

- [Cloudflare Learning: Load balancing](#)
- [Elastic Load Balancing](#)
- [Application Load Balancer](#)

Q56 A NAT Gateway is generating huge data-processing costs. How do you diagnose and reduce it?

Networking

Spoken answer · ~87 s

NAT gateway data-processing charges accrue per gigabyte for every byte passing through, on top of the hourly fee and data transfer, so heavy traffic gets expensive quickly. To diagnose, find who is generating the traffic and where it's going. Enable VPC flow logs on the NAT gateway's network interface, or on the private subnets, and query with Athena or CloudWatch Logs Insights to rank source IPs by bytes and destination IPs by bytes. Cost Explorer shows NatGateway-Bytes usage by account and zone; CloudWatch metrics on the gateway show bytes in and out. Typical culprits: traffic to S3 or DynamoDB, which can use free gateway endpoints; heavy ECR image pulls from EKS nodes, fixable with ECR interface endpoints; traffic to other AWS services such as CloudWatch, STS, and Secrets Manager, which interface endpoints can serve privately; big downloads of packages or updates; log shipping to external SaaS; and cross-AZ traffic through a NAT in another zone. Reduce cost with gateway endpoints for S3 and DynamoDB first, since they are free; interface endpoints where the volume justifies their hourly cost; a NAT gateway per zone to avoid cross-zone charges; caching images and packages in internal mirrors; compressing data; and moving heavy workloads to public subnets where appropriate. Validate savings afterwards.

Likely follow-ups

- Why are S3 gateway endpoints the first fix?
- How do flow logs identify the heaviest NAT users?

Find what is going through NAT with VPC Flow Logs:

```
aws ec2 create-flow-logs --resource-type Subnet --resource-ids subnet-priv --traffic-type ALL --log-destination-type cloud-watch-logs
```

```
# Query: top talkers / destinations through the NAT ENI
```

```
# Replace AWS API traffic with VPC endpoints (no NAT charges):
```

```
aws ec2 create-vpc-endpoint --vpc-endpoint-type Gateway --service-name ...s3 --route-table-ids rtb # S3/DynamoDB FREE
```

```
aws ec2 create-vpc-endpoint --vpc-endpoint-type Interface --service-name ...ecr.dkr # ECR, etc.
```

NAT Gateways charge both hourly and per-GB processed, so high egress from private subnets gets expensive. Diagnose with VPC Flow Logs to find the top destinations. The biggest win is usually adding VPC endpoints: S3/DynamoDB via free Gateway endpoints, and ECR/Secrets Manager/etc. via Interface endpoints, so that traffic to AWS services bypasses the NAT entirely.

Common hidden cost: pulling container images from ECR or writing to S3 through the NAT Gateway. Gateway endpoints for S3/DynamoDB are free and eliminate that data-processing charge immediately. Also check cross-AZ NAT (place a NAT per AZ to avoid cross-AZ data charges) and chatty third-party calls that could be cached or batched.

Official docs

- [RFC 3022: Traditional NAT](#)
- [Amazon VPC: NAT gateways](#)

Q57 **How do you debug why an EC2 instance in a private subnet cannot reach the internet?**

Networking

 Spoken answer · ~86 s

I trace the outbound path in order. An instance in a private subnet reaches the internet through a NAT gateway, so check each hop. First, the instance: does it have the right security group outbound rules, allowing the destination, and does the host firewall allow it? Then DNS: can it resolve names, and is the VPC's DNS resolution enabled? Second, the subnet's route table: there must be a default route, zero-zero-zero-zero slash zero, pointing to a NAT gateway, not directly to an internet gateway. Third, the NAT gateway: it must be in a public subnet, be in the available state, and have an elastic IP. Fourth, the public subnet's route table must have a default route to an internet gateway, and the internet gateway must be attached to the VPC. Fifth, NACLs on both subnets must allow outbound traffic and the return traffic on ephemeral ports, since they're stateless. Sixth, check the zone: a NAT in a different zone could be down. Also consider whether a proxy is required, or an egress firewall or Network Firewall rule blocks the traffic. Use VPC Reachability Analyzer to find the blocking component, and flow logs to see rejected traffic. Finally verify with curl from the instance, using Session Manager.

Likely follow-ups

- Which route does a private subnet need for internet access?
- Why check both subnets' route tables?

Walk the path methodically:

1. Route table: does the private subnet have 0.0.0.0/0 -> nat-gateway-id?

```
aws ec2 describe-route-tables --filters Name=association.subnet-id,Values=subnet-priv
```

2. NAT Gateway: is it in a PUBLIC subnet with an IGW route + an EIP?

3. Security Group: outbound allowed? (default allows all egress)

4. NACL: stateless - allow outbound AND inbound ephemeral ports (1024-65535)?

5. DNS: enableDnsSupport/enableDnsHostnames on the VPC?

Systematic checklist: the private subnet route table must send 0.0.0.0/0 to the NAT Gateway; the NAT Gateway itself must live in a public subnet that routes to the Internet Gateway and has an Elastic IP; the Security Group must allow the outbound traffic; and because NACLs are stateless, the outbound NACL must allow the request and the inbound NACL must allow the ephemeral return ports. Finally, confirm VPC DNS settings if name resolution fails.

Most common cause: the NAT Gateway placed in a private subnet by mistake (it must be in a public subnet with an IGW route), or a missing 0.0.0.0/0 route in the private route table. The stateless-NACL ephemeral-port trap is the second most common — Security Groups handle return traffic automatically, NACLs do not.

Q58 How do you design VPC CIDR and IP addressing to avoid running out of IPs or conflicts?

Networking

 Spoken answer · ~85 s

IP planning is hard to change later, so I plan deliberately. First, define the whole address space: a company-wide plan that allocates non-overlapping ranges to every VPC, account, region, and on-premises network, recorded in an IP address management tool, such as AWS VPC IPAM, so conflicts are avoided; overlapping CIDRs block peering, Transit Gateway routing, and VPN connectivity. Choose RFC 1918 ranges and size generously: a VPC can be from slash sixteen to slash twenty-eight, and a slash sixteen gives sixty-five thousand addresses, which is usually a safe default for production; small VPCs run out. Subdivide into subnets per tier and per zone, remembering AWS reserves five addresses per subnet, and leave room for growth rather than allocating everything. Size subnets with workloads in mind: EKS with the VPC CNI assigns pod IPs from the subnets, so it consumes addresses fast; use larger subnets for node and pod ranges, enable prefix delegation, or use custom networking with secondary CIDRs from the 100.64.0.0/10 range to extend capacity. A VPC can have secondary CIDR blocks added later as a mitigation. Reserve ranges for future regions and environments. Document the plan, validate with automated checks in infrastructure as code, and monitor subnet IP utilisation with alarms.

Likely follow-ups

- Why does EKS pod networking consume so many IPs?
- How does IPAM prevent overlapping CIDRs?

```
# Plan non-overlapping CIDRs (critical for peering/TGW/VPN):
```

```
Prod VPC: 10.0.0.0/16    Dev VPC: 10.1.0.0/16    On-prem: 10.100.0.0/16
```

```
# Reserve space; subnets sized for growth (each AZ gets a /20, etc.)
```

```
# EKS eats IPs fast (VPC CNI assigns a VPC IP per pod):
```

```
# mitigate with prefix delegation or a secondary CIDR for pods:
```

```
aws ec2 associate-vpc-cidr-block --vpc-id vpc-x --cidr-block 100.64.0.0/16
```

Choose non-overlapping RFC1918 ranges across all VPCs and on-prem up front — overlaps make peering, Transit Gateway, and VPN routing impossible to fix later without re-IP. Size subnets for growth (AWS reserves 5 IPs per subnet). EKS with the VPC CNI assigns a real VPC IP to every pod, so clusters exhaust subnets quickly; use prefix delegation, larger subnets, or a secondary (e.g. 100.64.0.0/16 CGNAT) CIDR dedicated to pods.

EKS IP exhaustion is a frequent production problem: pods Pending with `'insufficient IP addresses'`. Fixes: enable VPC CNI prefix delegation (assigns /28 blocks, far more pods per node), add a secondary CIDR for pod subnets, or use custom networking. Use IPAM (AWS VPC IP Address Manager) to plan and track allocation org-wide.

Official docs

- [RFC 4632: CIDR](#)
- [Amazon VPC: What is Amazon VPC?](#)

Q59 What are VPC Flow Logs and how do you use them for security and troubleshooting?

Networking

Spoken answer · ~81 s

VPC Flow Logs capture metadata about IP traffic going to and from network interfaces in a VPC, subnet, or individual interface. Each record includes source and destination addresses and ports, protocol, packets, bytes, the action, accept or reject, and timestamps; with custom fields, you can add details such as VPC ID, instance ID, and TCP flags. They do not capture packet payloads. Logs can be published to CloudWatch Logs, S3, or Firehose; I prefer S3 in Parquet format queried with Athena for cost and scale. For troubleshooting, flow logs show whether traffic reached the interface and whether security groups or NACLs rejected it; a reject record on the destination points to filtering, while no record suggests a routing or upstream problem. Asymmetric patterns, such as accepted inbound and rejected return traffic, indicate NACL issues. For security, they help identify unexpected outbound connections, port scans, large data exfiltration, traffic to known bad IPs, and lateral movement; GuardDuty analyses flow logs automatically. For cost analysis, they reveal top talkers and cross-zone traffic. Practicalities: choose the aggregation interval, enable on the right scope, filter to reduce cost, set retention, and remember there's a delay before records appear.

Likely follow-ups

- What does a REJECT record tell you?
- Why store flow logs in Parquet in S3?

```
# Enable on a VPC/subnet/ENI, to CloudWatch or S3:
```

```
aws ec2 create-flow-logs --resource-type VPC --resource-ids vpc-x --traffic-type ALL --log-destination-type s3 --log-destination arn:aws:s3:::flowlogs
```

```
# Fields include srcaddr, dstaddr, srcport, dstport, action (ACCEPT/REJECT), bytes
```

```
# Analyze at scale with Athena over the S3 logs:
```

```
SELECT srcaddr, dstaddr, sum(bytes) FROM flowlogs WHERE action='REJECT' GROUP BY 1,2 ORDER BY 3 DESC
```

Flow Logs capture metadata about IP traffic (not payloads): source/dest IP and port, protocol, bytes, and whether the Security Group/NACL ACCEPTed or REJECTed it. Use them to troubleshoot connectivity (REJECT entries show what a firewall blocked), find top talkers driving NAT/data costs, detect exfiltration or port scans, and feed GuardDuty. Query large volumes cheaply with Athena over S3.

Troubleshooting trick: when a connection mysteriously fails, Flow Logs tell you whether packets even arrived and whether a SG/NACL rejected them — instantly separating 'firewall blocked it' from 'it never got there' from 'app problem'. For security, alerting on unexpected REJECT/ACCEPT patterns and large outbound transfers catches scans and data exfiltration.

Official docs

- [Amazon VPC: Flow Logs](#)

Q60 How do you architect a multi-region application on AWS? What are the hard parts?

Networking

Spoken answer · ~83 s

Multi-region architecture starts with the requirement: what RTO and RPO, what latency, and whether active-active is truly needed. Patterns range from backup and restore, through pilot light and warm standby, to active-active. The building blocks: global traffic routing with Route 53 or Global Accelerator and health checks; infrastructure as code deployed identically to each region; replicated artefacts, such as ECR images and AMLs; and replicated data. The hard part is data. For S3, use cross-region replication. For databases, Aurora Global Database provides low-lag replication with a managed failover, though only one writer region; DynamoDB global tables provide multi-region multi-writer with last-writer-wins conflict resolution; RDS cross-region read replicas can be promoted. True multi-region writes force conflict-resolution and consistency decisions. Other hard parts: failover is rarely tested and fails when needed; hidden dependencies on a single region, such as IAM or DNS configurations, secrets, and third-party services; configuration and secrets replication; deployments that must be coordinated; cost, as you run duplicate capacity; and operational complexity, with observability across regions. Avoid split-brain by designing a clear failover and fallback process. Practise game days to prove the recovery time. Many workloads are better served by multi-AZ plus a well-tested warm standby than by active-active.

Likely follow-ups

- Why is multi-region data replication the hardest part?
- What does Aurora Global Database give you?

```
# Global routing: Route 53 latency/failover or Global Accelerator (anycast):
```

```
aws globalaccelerator create-accelerator --name app # 2 static anycast IPs, fast failover
```

```
# Data replication is the hard part:
```

DynamoDB Global Tables (multi-region active-active)

Aurora Global Database (1 primary write region, fast read replicas + failover)

S3 Cross-Region Replication; ECR cross-region replication for images

Compute is the easy part — replicate stacks per region behind Route 53 (latency or failover routing) or Global Accelerator for anycast IPs and sub-minute failover. The hard part is **data**: you must choose between active-active (DynamoDB Global Tables — handles multi-region writes with last-writer-wins) and active-passive (Aurora Global Database — single write region, fast cross-region replicas and promotion). You also must handle replication lag, conflict resolution, and consistency.

Be honest about why: multi-region adds large cost and complexity; most needs are met by multi-AZ. Justify it by RTO/RPO (true regional DR or active-active latency). Decide write topology first (active-active vs active-passive), since that dictates the database choice and the whole architecture. Replication lag and split-brain are the recurring pain points interviewers probe.

Official docs

- [AWS Well-Architected: Reliability pillar](#)
- [Google Cloud: Multi-region deployment archetypes](#)

Q61 **Explain S3 storage classes and lifecycle policies. How do you optimize storage cost?**

Storage

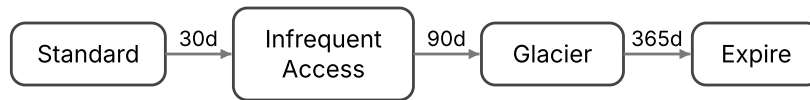
Spoken answer · ~83 s

S3 storage classes trade cost against access speed and retrieval fees. Standard is for frequently accessed data. Intelligent-Tiering automatically moves objects between access tiers based on usage, with a small monitoring fee, ideal when patterns are unknown or change. Standard-IA and One Zone-IA are cheaper per gigabyte for infrequently accessed data, with retrieval charges and a thirty-day minimum; One Zone-IA stores in a single zone, so it's for re-creatable data. Glacier Instant Retrieval offers archive pricing with millisecond access; Glacier Flexible Retrieval takes minutes to hours; Glacier Deep Archive is the cheapest, with retrieval in hours, for long-term retention. Lifecycle policies automate transitions and expiry: for example, move logs to Standard-IA after thirty days, to Glacier after ninety, and delete after a year; also expire old noncurrent versions and abort incomplete multipart uploads. To optimise cost: use S3 Storage Lens and Storage Class Analysis to understand access patterns; apply Intelligent-Tiering to unpredictable data; avoid many tiny objects, which have minimum size charges in IA classes; compress and aggregate data; watch request and retrieval costs, which can exceed storage savings if data is accessed more than expected; and delete unused data and old versions. Remember transition requests also cost money.

Likely follow-ups

- When does Intelligent-Tiering beat manual lifecycle rules?
- What hidden costs can erase savings from cold storage classes?

🚧 S3 lifecycle



Classes by access pattern:

Standard – frequent access

Intelligent-Tiering – auto-moves objects between tiers (unknown/changing access)

Standard-IA / One Zone-IA – infrequent access, lower storage cost + retrieval fee

Glacier Instant / Flexible / Deep Archive – archival, cheapest, retrieval latency

Lifecycle rule: transition + expire:

```
aws s3api put-bucket-lifecycle-configuration --bucket b --lifecycle-configuration \
'{"Rules":[{"Status":"Enabled","Transitions":[{"Days":30,"StorageClass":"STANDARD_IA"},
{"Days":90,"StorageClass":"GLACIER"}],"Expiration":{"Days":365}}]}'
```

Match class to access pattern: Standard for hot data, Standard-IA for infrequent, Glacier tiers for archives, and Intelligent-Tiering when access is unpredictable (it auto-tiers with no retrieval fees on the frequent/infrequent tiers). Lifecycle policies automate transitions to cheaper classes over time and expire old data. Also expire incomplete multipart uploads and old versions, which silently accumulate cost.

Cost wins: Intelligent-Tiering is the safe default for unknown patterns (no retrieval fees, small monitoring charge). Use S3 Storage Lens to find buckets to optimize. Watch for the IA minimum object size/duration charges (small short-lived objects can cost more in IA than Standard). Always add a lifecycle rule to abort incomplete multipart uploads.

📖 Official docs

- [Kubernetes: StorageClasses](#)
- [Amazon S3: Lifecycle management](#)
- [Amazon S3 User Guide](#)

Q62 **Compare EBS, EFS, and S3. When do you use each, and which EBS volume types?**

Storage

 Spoken answer · ~85 s

EBS is block storage for a single instance in one zone; EFS is shared network file storage across zones; S3 is object storage accessed by API. Use EBS for boot disks, databases, and low-latency block workloads; EFS for shared POSIX file systems, content management, and ReadWriteMany needs in Kubernetes; S3 for backups, logs, static assets, and data lakes. EBS volume types: gp3 is the general-purpose SSD default, with baseline three thousand IOPS and a hundred and twenty-five megabytes per second, and you can provision more IOPS and throughput independently, cheaper than gp2; gp2 ties performance to size with burst credits and is mostly legacy. io2 and io2 Block Express provide provisioned, high IOPS with high durability for critical databases and latency-sensitive workloads. st1 is throughput-optimised HDD for large sequential workloads like big data and logs, and sc1 is cold HDD for infrequent access. HDD types can't be boot volumes. Choose by IOPS, throughput, and latency needs, and right-size over time since volumes can be modified online. Use snapshots, incremental and stored in S3, for backup, with encryption and automated lifecycle through Data Lifecycle Manager or AWS Backup. Remember EBS is zonal, so for resilience replicate at the application layer or restore snapshots elsewhere.

Likely follow-ups

- Why is gp3 usually preferred over gp2?
- When would you provision io2?

```
# EBS = block storage, single-AZ, attaches to one EC2 (mostly):
```

```
gp3 – general SSD (baseline 3000 IOPS, tune IOPS/throughput independently)
```

```
io2 Block Express – high IOPS/durability (databases)
```

```
st1/sc1 – throughput HDD / cold HDD (big sequential, logs, cheap)
```

```
# EFS = managed NFS, multi-AZ, many instances share it:
```

```
aws efs create-file-system --performance-mode generalPurpose --throughput-mode elastic
```

```
# S3 = object storage over HTTP, infinite scale
```

EBS is a network block device for a single instance (like a virtual disk) — use for boot volumes and databases; gp3 is the modern default (cheaper than gp2, decoupled IOPS/throughput), io2 Block Express for high-performance databases. EFS is managed NFS that many instances across AZs can mount simultaneously — use for shared filesystems and lift-and-shift apps needing POSIX. S3 is object storage accessed via API — use for backups, media, data lakes, and static assets.

Quick rule: one instance, low-latency block + filesystem → EBS (gp3 default, io2 for DBs). Shared across many instances/AZs → EFS. Web-scale, durable, accessed by URL/API → S3. Migrate gp2 to gp3 for an easy ~20% cost cut. Use EBS snapshots (incremental) for backups and DLM to automate them.

Official docs

- [Amazon EBS](#)
- [Amazon EFS](#)
- [Amazon S3 User Guide](#)

Q63 How do S3 consistency, versioning, and Object Lock work? How do you protect against deletion?

Storage

Spoken answer · ~82 s

S3 provides strong read-after-write consistency for all operations: after a successful write or delete, any subsequent read or list reflects it. That removed the old eventual consistency caveats. Versioning keeps multiple versions of an object under the same key; overwrites create new versions, and deletes add a delete marker rather than removing data, so you can recover earlier versions. It's essential protection against accidental overwrite or deletion, but storage grows, so use lifecycle rules to expire noncurrent versions. Object Lock provides write-once-read-many protection: in governance mode, privileged users can override retention; in compliance mode, no one, even the root user, can delete or alter the object until retention ends; legal holds apply indefinitely. It requires versioning. To protect against deletion: enable versioning, add MFA Delete for versioned buckets where appropriate, use Object Lock for regulatory immutability, restrict delete permissions through least privilege and bucket policies, deny delete actions via SCPs for critical buckets, replicate to another bucket or account, ideally with a different owner so a compromised account can't remove both, and back up with AWS Backup. Monitor with CloudTrail data events and alerts. Test recovery procedures so you know how to restore a previous version quickly.

Likely follow-ups

- What is the difference between governance and compliance mode?
- Why replicate to a separate account?

```
# Versioning (keeps every version; deletes add a delete marker):
```

```
aws s3api put-bucket-versioning --bucket b --versioning-configuration Status=Enabled
```

```
# MFA Delete (require MFA to permanently delete versions):
```

```
aws s3api put-bucket-versioning --bucket b --versioning-configuration  
Status=Enabled,MFADelete=Enabled --mfa "serial code"
```

```
# Object Lock (WORM - compliance, immutable for a retention period):
```

```
aws s3api put-object-retention --bucket b --key k --retention
Mode=COMPLIANCE,RetainUntilDate=2027-01-01
```

S3 now provides strong read-after-write consistency for all operations. Versioning keeps every version of an object so overwrites and deletes are recoverable (a delete just adds a delete marker). Object Lock enforces WORM (write-once-read-many): in Governance mode privileged users can override, in Compliance mode **no one** (not even root) can delete until the retention expires — the strongest defense against ransomware and malicious deletion.

Ransomware/insider defense: versioning + MFA Delete + Object Lock (Compliance) makes backups immutable so an attacker with credentials still cannot erase them. Pair with Cross-Region Replication to a separate account for an air-gapped copy. This is the S3 equivalent of the 3-2-1-1-0 immutable-backup rule.

Official docs

- [Amazon S3: Versioning](#)
- [Amazon S3 User Guide](#)

Q64 How do you back up and recover AWS resources? Explain snapshots, AWS Backup, and DR strategies.

Storage

Spoken answer · ~80 s

Backup and recovery should start from the business's RTO and RPO per workload. Building blocks: snapshots are point-in-time, incremental copies of EBS volumes and RDS databases, stored durably, which can be copied across regions and accounts. AWS Backup centralises policy-based backups across services, including EC2, EBS, RDS, DynamoDB, EFS, and S3, with schedules, retention, lifecycle to cold storage, cross-region and cross-account copies, vault lock for immutability, and compliance reporting. Combine with native features such as RDS point-in-time recovery and S3 versioning. Test restores regularly, because an untested backup is a hope, not a plan. DR strategies in order of increasing cost and decreasing recovery time: backup and restore, with hours of recovery; pilot light, where the core data is replicated and minimal infrastructure runs; warm standby, a scaled-down full environment; and multi-site active-active, with near-zero recovery. Choose by tier. Implement infrastructure as code so environments can be recreated, replicate AMIs, images, and secrets to the recovery region, and automate failover where possible. Protect the backups against ransomware or account compromise through cross-account copies and vault lock. Document runbooks, run DR drills, measure actual recovery time, and monitor backup job success with alerts.

Likely follow-ups

- Why copy backups to a separate account?
- What distinguishes pilot light from warm standby?

EBS snapshots are incremental; automate with Data Lifecycle Manager or AWS Backup:

```
aws backup create-backup-plan --backup-plan '{"BackupPlanName":"daily","Rules":
[{"RuleName":"daily","TargetBackupVaultName":"vault","ScheduleExpression":"cron(0 5 * * ?
*)","Lifecycle":{"DeleteAfterDays":35}}]}'
```

Cross-region/cross-account copy for DR + ransomware isolation:

```
aws backup ... CopyActions to a vault in another region/account (locked vault)
```

AWS Backup centralizes backups across EBS, RDS, DynamoDB, EFS, S3, and more with policy-based plans, retention, and cross-region/cross-account copies. EBS snapshots are incremental (only changed blocks) and stored in S3. For DR, pick a strategy by RTO/RPO: Backup & Restore (cheapest, hours), Pilot Light (core running, scale up on failover), Warm Standby (scaled-down full stack), or Multi-Site Active-Active (lowest RTO, highest cost).

Test restores + isolate copies: an untested backup is not a backup — run restore drills and measure actual RTO. Copy backups to a separate region and a separate account with a locked Backup Vault (Vault Lock = WORM) so a compromised production account cannot delete your DR copies. Choose the DR tier by business RTO/RPO, not by default.

Official docs

- [AWS Backup](#)

Spoken answer · ~81 s

S3 scales to very high request rates, but you have to design for it. It supports at least thirty-five hundred write and fifty-five hundred read requests per second per prefix, so spreading keys across many prefixes multiplies throughput; avoid hot single prefixes. Use parallelism: many concurrent connections and requests, which is how the SDKs and tools like the AWS CRT-based clients maximise throughput. For large objects, use multipart upload to upload parts in parallel, retry failed parts independently, and resume; and use byte-range GETs to download large objects in parallel or fetch only needed parts. Use Transfer Acceleration or CloudFront for geographically distant clients. Keep compute in the same region as the bucket and use VPC gateway endpoints to avoid NAT bottlenecks. Choose appropriate object sizes: very small objects incur request overhead, so batch or aggregate; sizes of a few megabytes to hundreds of megabytes work well. For analytics, use columnar formats like Parquet with partitioning, so queries read only relevant data. S3 Express One Zone gives single-digit-millisecond latency for latency-sensitive workloads. Implement retries with exponential backoff to handle throttling responses. Monitor with CloudWatch request metrics and tune based on observed error rates and latency.

Likely follow-ups

- Why does the prefix matter for request throughput?
- How does multipart upload help with large files?

```
# Multipart upload for large objects (parallel parts, resumable):
```

```
aws s3 cp big.bin s3://b/ --expected-size 5368709120 # CLI auto-multipart
```

```
# S3 scales to 3,500 PUT and 5,500 GET per second PER PREFIX -> spread keys
```

```
# S3 Transfer Acceleration (upload via CloudFront edge):
```

```
aws s3api put-bucket-accelerate-configuration --bucket b --accelerate-configuration  
Status=Enabled
```

```
# Byte-range GETs to parallelize large downloads
```

S3 scales horizontally per key prefix (3,500 writes / 5,500 reads per second each), so high-throughput workloads should distribute object keys across many prefixes rather than a sequential hot prefix. Use multipart upload for large objects (parallelism + resume on failure) and byte-range fetches to parallelize downloads. Transfer Acceleration routes uploads through the nearest CloudFront edge for far-flung clients. For analytics, columnar formats + S3 Select / Athena reduce data scanned.

Prefix design matters: the old advice to randomize key prefixes is less critical now (S3 auto-partitions), but a single sequential hot prefix can still bottleneck — spread high-rate keys. For data lakes, partition by query

dimensions (date/region) and use Parquet so engines scan less. CRT-based clients/awscli get much higher throughput.

Official docs

- [Amazon S3 User Guide](#)

Q66 What is S3 event-driven processing and how do you build it reliably?

Storage

Spoken answer · ~85 s

Event-driven processing on S3 means object operations, like creation, trigger downstream work. S3 can send event notifications to Lambda, SQS, SNS, or EventBridge; EventBridge offers richer filtering, multiple destinations, and archive and replay. A typical flow: a file lands in a bucket, an event goes to an SQS queue, and consumers, Lambda or containers, process it and write results elsewhere. To build it reliably: assume at-least-once delivery, so duplicates can occur and processing must be idempotent, for example by keying on bucket, key, and version ID or ETag. Use SQS between S3 and the consumer to buffer spikes, control concurrency, and enable retries, with a dead-letter queue for messages that repeatedly fail, plus alarms on queue depth and DLQ messages. Set visibility timeouts longer than processing time. Handle ordering issues: events aren't guaranteed in order, so versioning or sequencers help. Filter by prefix and suffix to avoid recursive triggers, such as a function writing back to the same bucket in a way that triggers itself, which can cause costly loops; use separate prefixes or buckets. Make processing resumable for large files, and verify results. For complex multi-step flows, orchestrate with Step Functions. Monitor failures, latency, and throughput, and test replay of missed events.

Likely follow-ups

- Why must S3 event consumers be idempotent?
- How do you avoid a recursive trigger loop?

```
# S3 event notification -> SQS / SNS / Lambda / EventBridge:
```

```
aws s3api put-bucket-notification-configuration --bucket b --notification-configuration \
'{"QueueConfigurations":[{"QueueArn":"arn:...sqs","Events":["s3:ObjectCreated:*"]}]}\'
```

```
# Prefer SQS (or EventBridge) in front of Lambda for buffering + retries + DLQ:
```

```
S3 -> SQS -> Lambda (batch) , with a Dead Letter Queue for failures
```

S3 can emit events on object create/delete to SQS, SNS, Lambda, or EventBridge — the basis of serverless pipelines (image processing, ingestion, ETL triggers). For reliability, put SQS or EventBridge between S3 and the consumer: it buffers spikes, enables batching and retries, and provides a dead-letter queue for poison messages, instead of invoking Lambda directly where bursts can throttle and failures vanish.

Reliability patterns: use SQS for decoupling + DLQ; make consumers idempotent (events can be delivered more than once); for ordering or fan-out, use SNS-to-SQS or EventBridge. EventBridge adds rich filtering and many

targets. Always have a DLQ and alarming on it so failed objects are not silently dropped.

Official docs

- [Amazon S3 User Guide](#)

Q67 How do you securely share data across AWS accounts (S3, KMS, cross-account access)?

Storage

Spoken answer · ~83 s

Cross-account sharing needs permission on both sides. For S3, there are two main patterns. A bucket policy in the owning account grants access to a principal in another account, and the other account must also allow the action in an IAM policy for that principal; both are needed. Alternatively, the other account assumes a role in the owning account, through a trust policy, and acts with that role's permissions, keeping everything controlled in one account. Newer options include S3 Access Points with their own policies, simplifying management for many consumers. For objects uploaded across accounts, set object ownership to bucket owner enforced to avoid ACL ownership problems. For encrypted data, KMS adds another gate: the key policy must allow the other account, and the principal's IAM policy must permit `kms:Decrypt` and related actions; with SSE-S3 there's no such step, but with customer-managed keys you must share the key explicitly. Use conditions like `aws:PrincipalOrgID` to restrict to your organisation instead of listing accounts. For broader data sharing, use Lake Formation, or Resource Access Manager for supported resources. Apply least privilege, monitor with CloudTrail and Access Analyzer, which flags external access, and document the sharing so it can be reviewed and revoked.

Likely follow-ups

- Why do both the bucket policy and the caller's IAM policy need to allow access?
- What extra step does a customer-managed KMS key add?

```
# Bucket policy granting another account (resource-based):
```

```
{"Principal":{"AWS":"arn:aws:iam::ACCOUNT-B:root"},"Action":["s3:GetObject"],"Resource":["arn:aws:s3::b/*"]}
```

```
# If SSE-KMS, the KMS key policy must ALSO allow Account B:
```

```
aws kms put-key-policy ... # grant kms:Decrypt to ACCOUNT-B principal
```

```
# Modern: enforce bucket-owner ownership so cross-account writes are owned by you:
```

```
aws s3api put-bucket-ownership-controls --bucket b --ownership-controls Rules=[{"ObjectOwnership": "BucketOwnerEnforced"}]
```

Cross-account S3 sharing needs both a resource-based bucket policy allowing the other account/role and, if the data is SSE-KMS encrypted, a KMS key policy that also grants that principal kms:Decrypt — forgetting the KMS half is the most common failure. Use BucketOwnerEnforced ownership so objects written by another account are owned by the bucket owner (ACLs are disabled). For richer governance, use AWS RAM or Lake Formation.

The classic cross-account gotcha: the bucket policy allows access but downloads fail with AccessDenied because the KMS key policy was not updated. Both the bucket policy AND the KMS key policy must permit the external principal. Also prefer assuming a role over root-account principals, and scope to specific prefixes.

Official docs

- [AWS IAM: Roles](#)
- [AWS KMS](#)
- [Amazon S3 User Guide](#)

Q68 What is a data lake on AWS and how do the core services fit together?

Storage

Spoken answer · ~83 s

A data lake is a central repository storing structured and unstructured data at scale in its raw or lightly processed form, so many analytics tools can use it. On AWS, S3 is the storage foundation, organised into zones such as raw, curated, and consumption layers, using columnar formats like Parquet and partitioning for efficiency. Ingestion: Kinesis or MSK for streaming, Firehose to land data in S3, DMS for database replication, Glue jobs or AppFlow for batch and SaaS sources. The AWS Glue Data Catalog holds metadata, schemas, and partitions, shared across services; Glue crawlers discover schemas, and Glue ETL or EMR transform data. Query and analysis: Athena runs serverless SQL directly on S3, Redshift Spectrum or Redshift serves warehouse workloads, EMR runs Spark and Hadoop, and QuickSight visualises. Governance and security: Lake Formation provides central permissions down to table, column, and row level, plus data sharing across accounts; KMS encryption, CloudTrail, and Macie for sensitive data discovery. Operate it with data quality checks, lifecycle policies, cost controls, and clear ownership, otherwise it turns into a data swamp. Table formats like Iceberg add ACID transactions and time travel, creating a lakehouse. Start with clear use cases rather than building infrastructure first.

Likely follow-ups

- What role does the Glue Data Catalog play?
- How does Lake Formation simplify permissions?

```
# S3 = storage layer (raw / cleaned / curated zones):
```

```
s3://lake/raw/  s3://lake/processed/  s3://lake/curated/
```

```
# Glue = catalog + ETL (crawlers infer schema into the Data Catalog):
```

```
aws glue start-crawler --name lake-crawler
```

```
# Athena = serverless SQL over S3; Lake Formation = governance/permissions:
```

```
SELECT * FROM curated.orders WHERE dt='2026-06-01' -- pay per TB scanned
```

An AWS data lake centers on S3 as cheap, durable storage organized into zones (raw, processed, curated). AWS Glue provides the Data Catalog (a metastore of table schemas) and serverless ETL; crawlers infer schemas. Athena runs serverless SQL directly over S3 (priced per data scanned), and Lake Formation centralizes fine-grained permissions. Use columnar Parquet + partitioning to cut cost and speed queries; Redshift Spectrum or EMR handle heavier analytics.

Cost + performance: Athena bills per TB scanned, so store data as partitioned Parquet/ORC and only scan needed partitions/columns — this can cut query cost by 10-100x versus raw JSON/CSV. Lake Formation gives table/column/row-level access control across the lake, which raw IAM cannot easily express.

Official docs

- [Amazon S3 User Guide](#)

Q69 **An application reports 'AccessDenied' reading from an encrypted S3 bucket. How do you debug?**

Storage

 Spoken answer · ~87 s

AccessDenied on an encrypted S3 bucket has several layers, so I check each systematically. First, read the exact error and identify the caller using CloudTrail or the error details: which principal, which action, and which resource. Then check the identity-based IAM policy: does it allow s3:GetObject on the bucket and object ARNs, both the bucket and the key path? Check for explicit denies, permission boundaries, session policies, and SCPs that restrict it. Check the bucket policy, including deny statements with conditions, such as requiring TLS, a specific VPC endpoint, or source IP, or restricting to an organisation ID. Check Block Public Access and ACL or ownership settings if cross-account. If it's cross-account, both accounts must allow. Then the encryption layer: if the bucket uses SSE-KMS, the principal also needs kms:Decrypt, and perhaps kms:GenerateDataKey, on the key; the key policy must permit the principal or delegate to IAM; and the key must be in the right region, not disabled or pending deletion. A missing KMS permission produces a confusing AccessDenied. Also check VPC endpoint policies if the request goes through an endpoint. Use the IAM policy simulator and Access Analyzer's policy validation, and check the object's existence, since missing objects also return AccessDenied without list permission. Fix the narrowest cause.

Likely follow-ups

- Why does a missing KMS permission show up as an S3 AccessDenied?
- How can a missing object produce AccessDenied?

```
# Check each layer that can deny:
```

1. Identity policy: does the role allow `s3:GetObject` on this bucket/prefix?

2. Bucket policy: any explicit Deny? Does it require a VPC endpoint / TLS / KMS?

3. KMS key policy: does the role have `kms:Decrypt` on the bucket key?

```
aws s3api get-bucket-policy --bucket b ; aws kms get-key-policy --key-id k --policy-name default
```

4. Block Public Access / SCP / VPC endpoint policy restricting access?

```
aws cloudtrail lookup-events ... # find the exact denying call + decode message
```

S3 AccessDenied is a layered problem: the identity policy, the bucket policy (which may add explicit Denies for non-TLS, wrong KMS key, or non-endpoint access), the KMS key policy (for SSE-KMS data you need `kms:Decrypt`), and any SCP or VPC endpoint policy can each block. Work top-down, and use CloudTrail plus `aws sts decode-authorization-message` to see precisely which policy denied the request.

For encrypted buckets, suspect KMS first: the object is readable per S3 policy but the role lacks `kms:Decrypt` on the key, so the download fails. An explicit Deny in the bucket policy (e.g. requiring `SecureTransport` or a specific endpoint) is the second usual cause. Remember: any explicit Deny anywhere overrides all Allows.



Official docs

- [Amazon S3 User Guide](#)

Spoken answer · ~84 s

The choice follows the access pattern, consistency needs, and scale. RDS is managed relational databases, such as PostgreSQL, MySQL, and others, for workloads needing SQL, joins, transactions, and relational integrity, at moderate scale; it's the default for many business applications. Aurora is AWS's cloud-native MySQL- and PostgreSQL-compatible engine with a distributed storage layer, up to fifteen read replicas, faster failover, and better scaling and throughput; use it when you need higher performance, availability, or read scaling than standard RDS, at higher cost. DynamoDB is a serverless key-value and document NoSQL database with single-digit-millisecond latency at nearly unlimited scale; use it for known access patterns, very high throughput, session stores, carts, and IoT, accepting limited ad hoc querying and the need to model carefully. ElastiCache is an in-memory cache and store, Redis or Memcached, for sub-millisecond reads, caching, sessions, leaderboards, and rate limiting; it complements a database rather than replacing it. Decision cues: relational queries and transactions, RDS or Aurora; massive scale with predictable access, DynamoDB; speed in front of another store, ElastiCache. Also consider operational burden, cost, team skills, and consistency needs. Often a system uses several: Aurora as the system of record, ElastiCache for hot reads, DynamoDB for specific high-scale features.

Likely follow-ups

- When does Aurora justify its higher price over RDS?
- Why can't ElastiCache replace the primary database?

RDS — managed relational (MySQL/Postgres/etc.), Multi-AZ, read replicas

Aurora — AWS cloud-native relational, 5x MySQL throughput, storage auto-grows

DynamoDB — serverless key-value/document, single-digit-ms at any scale

ElastiCache — in-memory (Redis/Memcached), caching + sessions + leaderboards

Pick by data model and scale needs. RDS for standard relational workloads where you want a managed engine. Aurora when you want relational + higher performance, faster failover, and storage that scales automatically (and Serverless v2 for variable load). DynamoDB for massive scale, predictable low latency, and serverless ops with a key-value/document access pattern designed around your queries. ElastiCache to offload reads, store sessions, and accelerate hot data in front of a database.

Access pattern drives the choice: known relational queries + joins → RDS/Aurora. Hyper-scale with well-defined key-based access → DynamoDB (model the table from the access patterns, not normalized like SQL). Reduce DB load / sub-ms reads → ElastiCache in front. A common production stack is Aurora + ElastiCache + DynamoDB for different workloads.

Official docs

- [Amazon ElastiCache](#)
- [Amazon RDS User Guide](#)
- [Amazon Aurora User Guide](#)

Q71 **How does Aurora differ from standard RDS architecturally? Explain the storage layer and failover.**

Databases

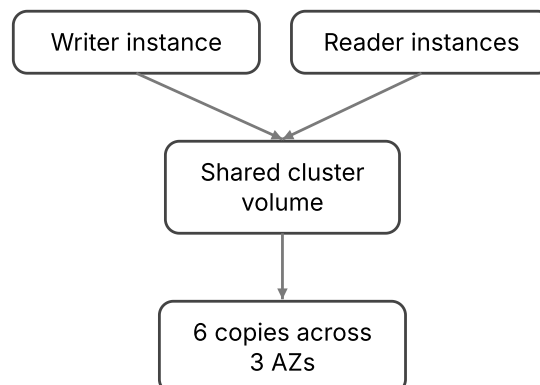
Spoken answer · ~85 s

Standard RDS uses a database engine running on an instance with attached EBS storage; Multi-AZ replicates synchronously to a standby with its own volume, and failover takes about a minute or two. Aurora separates compute from storage. The storage layer is a distributed, log-structured cluster volume replicated six ways across three availability zones, automatically growing up to a hundred and twenty-eight terabytes; the database instances send only redo log records to storage, not whole pages, which reduces network I/O, and storage nodes materialise pages. Writes need a quorum of four of six copies, and reads need three, so it tolerates losing a whole zone for writes and more for reads without data loss, with self-healing storage. Because all instances share the same storage volume, read replicas, up to fifteen, read the same data with very low replica lag, typically tens of milliseconds. Failover promotes a replica, and takes around thirty seconds or less, since no data has to be copied; the cluster endpoint, reader endpoint, and custom endpoints provide routing. Backups are continuous and incremental to S3 with point-in-time restore; cloning is fast through copy-on-write. Aurora also offers Global Database and Serverless v2. The trade-offs: higher cost, and some engine differences from community versions.

Likely follow-ups

- Why is Aurora failover faster than standard RDS Multi-AZ?
- What does the quorum design tolerate?

Aurora storage



Aurora cluster: writer + up to 15 readers share ONE distributed storage volume:

```
aws rds create-db-cluster --engine aurora-postgresql --db-cluster-identifier prod
```

```
aws rds create-db-instance --db-cluster-identifier prod --db-instance-class db.r6g.large --  
engine aurora-postgresql # writer
```

```
# Add readers (also failover targets):
```

```
aws rds create-db-instance --db-cluster-identifier prod ... # reader
```

```
# Cluster endpoint = writer; reader endpoint load-balances reads
```

Standard RDS replicates by streaming to separate full copies (Multi-AZ standby, async read replicas). Aurora decouples compute from storage: a shared, distributed, auto-healing storage volume replicated 6 ways across 3 AZs, with up to 15 reader instances reading the same storage. Failover just promotes a reader (no data copy needed) so it is typically under 30 seconds, and storage grows automatically up to 128TB.

Why it matters: Aurora gives faster failover (promote a reader, not rebuild), cheaper read scaling (readers share storage, no replication overhead), and continuous backup to S3. Use the reader endpoint for read scaling, the cluster endpoint for writes. Aurora Serverless v2 scales capacity in fine-grained increments for spiky workloads. Global Database extends it across regions.

Official docs

- [Amazon RDS User Guide](#)
- [Amazon Aurora User Guide](#)

Spoken answer · ~84 s

Scaling a relational database is staged, from cheapest to most complex. First, optimise: fix slow queries, add the right indexes, reduce N+1 patterns, and tune parameters, since many scaling problems are query problems. Second, scale up: a larger instance class, more memory, and faster storage, which is simple and often effective. Third, add caching: ElastiCache absorbs repeated reads. Fourth, offload reads with read replicas: the application or a proxy sends read-only queries to replicas; be aware of replication lag and that read-your-writes consistency needs care, such as routing a user's reads to the primary shortly after a write. Aurora makes replicas cheap and low-lag. Fifth, manage connections: relational databases handle limited concurrent connections, so use RDS Proxy or PgBouncer for pooling, especially with Lambda or many containers. Sixth, reduce data size by archiving or purging old data, and using native partitioning to improve query performance and maintenance. Seventh, for write scaling beyond a single primary, shard by a key such as tenant ID, either in the application or using a distributed SQL system, or move suitable workloads to DynamoDB; this carries significant complexity for cross-shard queries and transactions. Measure first, with Performance Insights, to choose the right stage, and load test changes.

Likely follow-ups

- Why exhaust query tuning before scaling hardware?
- What does RDS Proxy do for connection scaling?

```
# Read replicas to offload reads:
```

```
aws rds create-db-instance-read-replica --db-instance-identifier replical --source-db-instance-identifier prod
```

```
# RDS Proxy to pool connections (critical for Lambda/serverless):
```

```
aws rds create-db-proxy --db-proxy-name prod-proxy --engine-family POSTGRESQL --auth ... --role-arn ...
```

```
# Vertical scale (bigger instance) is the simplest first step
```

Scale reads with read replicas (route SELECTs to the reader endpoint), but writes still go to one primary — so the write path scales vertically (bigger instance) until you must shard/partition the data. Connection management is a frequent bottleneck: each connection costs memory, and serverless/many-instance apps exhaust the DB. RDS Proxy pools and reuses connections, smooths failovers, and is essential when Lambda fan-out would otherwise open thousands of connections.

Connection exhaustion is the classic serverless-meets-RDS failure: Lambda scales to thousands of concurrent executions, each opening a DB connection, and the database runs out. RDS Proxy fixes this by multiplexing.

Caching (ElastiCache) and read replicas relieve read pressure; eventually, sharding or moving hot tables to DynamoDB addresses write scaling.

Official docs

- [Amazon RDS: Read replicas](#)
- [PgBouncer](#)

Q73 How do you model data in DynamoDB? Explain partition keys, GSIs, and single-table design.

Databases

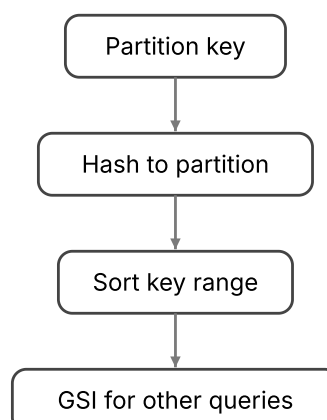
Spoken answer · ~88 s

DynamoDB modelling begins with access patterns, not entities. List every query the application needs, then design the keys to serve them. The partition key determines distribution: choose a high-cardinality attribute with even access, because a hot key throttles. The optional sort key orders items within a partition and enables range queries, begins-with, and sorting. Together they form the primary key. Global secondary indexes provide alternative key schemas for additional access patterns, with their own throughput, eventual consistency, and cost; local secondary indexes share the partition key but offer an alternative sort key and must be created with the table. Single-table design stores multiple entity types in one table, using generic key names like PK and SK and prefixes, so related items sit in the same partition and can be fetched in one query, avoiding the lack of joins. For example, a customer item and their orders share a partition key, with order items distinguished by sort key. Benefits: fewer round trips and lower latency; costs: complexity, harder ad hoc queries and migrations, and less readability. Use overloaded GSIs for additional patterns. It's not mandatory; for simple domains, multiple tables are fine. Use techniques like write sharding for hot keys, and keep item sizes small. Design iteratively with the access patterns documented.

Likely follow-ups

- Why start from access patterns rather than entities?
- What are the trade-offs of single-table design?

DynamoDB access



```
# Partition key (PK) + optional sort key (SK) define the primary key:
```

```
aws dynamodb create-table --table-name app --attribute-definitions  
AttributeName=PK,AttributeType=S AttributeName=SK,AttributeType=S --key-schema  
AttributeName=PK,KeyType=HASH AttributeName=SK,KeyType=RANGE --billing-mode PAY_PER_REQUEST
```

```
# Query a partition + sort range (efficient):
```

```
aws dynamodb query --key-condition-expression "PK = :u AND begins_with(SK, :p)"
```

```
# GSI for an alternate access pattern (different PK/SK)
```

DynamoDB performance depends on key design. The partition key determines data distribution — choose a high-cardinality key to spread load and avoid hot partitions. The sort key enables range queries within a partition. Global Secondary Indexes provide additional access patterns with different keys. Single-table design stores multiple entity types in one table (using composite PK/SK like USER#123 / ORDER#456) so related items are co-located and retrievable in one query.

Model from access patterns, not entities: unlike SQL, you design DynamoDB around the exact queries the app makes (no ad-hoc joins). List every access pattern first, then design PK/SK and GSIs to satisfy them. Avoid Scans (full table reads). Hot partitions and unbounded item collections are the main pitfalls; use write sharding for very hot keys.

Official docs

- [Amazon DynamoDB Developer Guide](#)
- [Apache Kafka documentation](#)

Spoken answer · ~83 s

DynamoDB offers two capacity modes. On-demand charges per request and scales automatically, with no capacity planning: ideal for new, unpredictable, or spiky workloads and for teams wanting simplicity, though the per-request price is higher at steady high volume. Provisioned mode lets you set read and write capacity units, optionally with auto scaling, and is cheaper for predictable, steady traffic, especially with reserved capacity. Throttling occurs when requests exceed capacity or a partition's limits, returning `ProvisionedThroughputExceeded` errors. Causes: insufficient provisioned capacity, a hot partition from skewed keys, bursts faster than auto scaling reacts, or a GSI running out of capacity, which throttles writes to the base table. Fixes: choose better partition keys or add write sharding, increase capacity or move to on-demand, adjust auto scaling targets, use exponential backoff with jitter in clients, and cache hot reads with DAX. DynamoDB adaptive capacity helps with uneven access but isn't a cure for a bad key. For cost: choose the capacity mode by traffic profile, use eventually consistent reads where acceptable, which cost half, keep items small, project only needed attributes into GSIs, use TTL to expire old data, choose Standard-IA table class for rarely accessed data, and monitor consumed versus provisioned capacity.

Likely follow-ups

- What causes a hot partition and how do you fix it?
- When is provisioned mode cheaper than on-demand?

```
# On-demand: pay per request, instant scaling, no capacity planning:
```

```
--billing-mode PAY_PER_REQUEST
```

```
# Provisioned + auto scaling: cheaper for steady predictable load:
```

```
aws dynamodb update-table --table app --billing-mode PROVISIONED --provisioned-throughput  
ReadCapacityUnits=100,WriteCapacityUnits=50
```

```
aws application-autoscaling register-scalable-target ... # auto-scale RCU/WCU
```

```
# DAX for microsecond cached reads:
```

```
aws dax create-cluster --cluster-name app-dax --node-type dax.r5.large
```

On-demand mode auto-scales per request with no planning — ideal for spiky or unknown traffic and new apps, but pricier per request at steady high volume. Provisioned mode with auto scaling is cheaper for predictable load. Throttling occurs when you exceed capacity or hit a hot partition; the SDK retries with backoff, but persistent throttling means under-provisioning or poor key distribution. DAX provides a microsecond in-memory cache for read-heavy workloads.

Avoid throttling + control cost: distribute keys to prevent hot partitions, use on-demand for spiky/unpredictable and provisioned+autoscaling for steady baseline. Watch ThrottledRequests and ConsumedCapacity in CloudWatch. Use DAX for hot reads, TTL to auto-expire old items (free deletes), and reserved capacity for committed savings.

Official docs

- [Microsoft: Cloud design pattern – Throttling](#)
- [Amazon DynamoDB Developer Guide](#)

Q75 How do you perform a zero-downtime database migration or major version upgrade on RDS/Aurora?

Databases

Spoken answer · ~86 s

The aim is to change the database with minimal or no downtime, using replication and a controlled cutover. For major version upgrades on RDS or Aurora, the blue/green deployment feature creates a synchronised staging copy, the green environment, running the new version and kept in sync by logical replication; you test it, then switch over, which typically takes under a minute, with the option to roll back to blue. For migrating between engines or to a new instance or region, use AWS Database Migration Service, with a full load plus change data capture to keep the target in sync, and the Schema Conversion Tool for heterogeneous engines. Process: assess compatibility and run pre-upgrade checks; build the target; replicate and validate data with row counts and checksums; test the application against the target, including performance under load; rehearse the cutover in staging; then in the cutover window, pause or drain writes, wait for replication lag to reach zero, switch the application's connection, such as by updating DNS or the endpoint, and verify. Keep the old database for rollback, perhaps with reverse replication. Reduce risk with backups and snapshots beforehand, DNS with low TTLs, connection retry handling in the application, and monitoring after cutover. Set go and no-go criteria.

Likely follow-ups

- How does RDS blue/green deployment work?
- What does DMS change data capture do?

```
# DMS for migration with ongoing replication (CDC) until cutover:
```

```
aws dms create-replication-task --migration-type full-load-and-cdc --source-endpoint-arn ... --target-endpoint-arn ...
```

```
# Blue/green deployment for RDS/Aurora upgrades (managed, with sync):
```

```
aws rds create-blue-green-deployment --source arn:::cluster:prod --target-engine-version 15.4
```

```
# Switch over when ready (fast, with safeguards):
```

```
aws rds switchover-blue-green-deployment --blue-green-deployment-identifier bgd-xxx
```

For engine/version upgrades, RDS/Aurora Blue/Green Deployments create a synchronized green copy at the new version, keep it replicating from blue, let you test it, then switch over in about a minute with safeguards (and blue retained for rollback). For cross-engine or cross-platform migrations, AWS DMS does a full load plus change-data-capture so the source stays live until you cut over, minimizing downtime.

Minimize risk: Blue/Green for in-place version upgrades (test the green, fast reversible switchover). DMS + Schema Conversion Tool for heterogeneous migrations (e.g. Oracle → Aurora PostgreSQL) with CDC so the app keeps writing to the source until cutover. Always rehearse the cutover, keep rollback ready, and validate data parity before switching.

Official docs

- [Kubernetes: Upgrading kubeadm clusters](#)
- [Amazon RDS User Guide](#)
- [Amazon Aurora User Guide](#)

Q76 How do you secure a database on AWS? (network isolation, encryption, IAM auth, rotation)

Databases

Spoken answer · ~82 s

Database security is defence in depth. Network isolation: put the database in private subnets with no public access; allow connections only from the application's security group; use VPC endpoints and private connectivity for admin access, with Session Manager and bastion-free patterns. Encryption: enable encryption at rest with KMS when creating the instance, since it can't be turned on later without a snapshot copy; enforce TLS in transit with parameter settings such as requiring SSL, and verify certificates in clients. Authentication and authorisation: avoid shared static passwords; use IAM database authentication to obtain short-lived tokens, or store credentials in Secrets Manager with automatic rotation; create least-privilege database users per application, separating admin, migration, and application roles; disable default accounts. Auditing and monitoring: enable database audit logging, such as the pgaudit extension, or Aurora advanced auditing, and export logs to CloudWatch; track API actions in CloudTrail; use GuardDuty RDS protection to detect anomalous logins; and Config rules to catch public exposure or unencrypted instances. Operations: patch regularly with maintenance windows, enable deletion protection, take encrypted backups and snapshots, restrict snapshot sharing, and apply SCPs to prevent public snapshots. Mask or tokenise sensitive data where possible, and review access periodically.

Likely follow-ups

- Why can't you enable encryption on an existing unencrypted RDS instance in place?
- What does IAM database authentication give you?

```
# Put DB in private/isolated subnets, no public access:
```

```
aws rds modify-db-instance --db-instance-identifier prod --no-publicly-accessible
```

```
# SG: allow only the app tier SG on the DB port:
```

```
aws ec2 authorize-security-group-ingress --group-id db-sg --protocol tcp --port 5432 --source-group app-sg
```

```
# Encryption at rest (KMS) + in transit (TLS); IAM database auth:
```

```
aws rds create-db-instance ... --storage-encrypted --kms-key-id k --enable-iam-database-authentication
```

Layer the controls: place the database in private/isolated subnets with no public accessibility, and restrict the Security Group to allow only the application tier's Security Group on the DB port (not CIDR ranges). Enable encryption at rest with KMS and enforce TLS in transit. Use IAM database authentication (token-based, no static DB passwords) or Secrets Manager with automatic rotation for credentials.

Best practices stack: private subnets + SG-referencing-SG (not 0.0.0.0/0) + KMS encryption + TLS enforced + IAM auth or Secrets Manager rotation + audit logging (e.g. pgAudit, enhanced monitoring) shipped to CloudWatch. Never make a production DB publicly accessible. Use RDS Proxy to also centralize Secrets-Manager-backed credentials.

Official docs

- [AWS KMS](#)

Q77 **What is ElastiCache used for, and how do you design caching to avoid stampedes and staleness?**

Databases

 Spoken answer · ~82 s

ElastiCache provides managed Redis, Valkey, or Memcached for low-latency access to hot data. Common uses: caching database query results and computed objects, session storage, rate limiting, leaderboards using sorted sets, pub/sub, and short-lived queues. Design caching around patterns: cache-aside, where the application checks the cache, loads from the database on a miss, and populates the cache, is the common default; write-through updates the cache on writes, reducing staleness but adding write latency. Staleness is controlled with TTLs matched to how fresh data must be, explicit invalidation on updates, and versioned keys. Stampedes happen when a popular key expires and many requests rebuild it at once, slamming the database. Prevent them with jittered TTLs so keys don't expire together, request coalescing or locks so only one caller rebuilds, serving stale data while refreshing in the background, and pre-warming. Also protect against cache penetration for missing keys by caching negative results briefly. Handle failure: the system must degrade gracefully if the cache is down, with timeouts and circuit breakers so the database isn't overwhelmed. Use Multi-AZ replication and cluster mode for availability and scale, size memory with an appropriate eviction policy, and monitor hit rate, evictions, latency, and memory.

Likely follow-ups

- How does jittered TTL prevent stampedes?
- What is cache-aside versus write-through?

```
# Redis cluster (replication + Multi-AZ failover):
```

```
aws elasticache create-replication-group --replication-group-id app --engine redis --num-node-groups 3 --replicas-per-node-group 2 --automatic-failover-enabled --multi-az-enabled
```

```
# Patterns: cache-aside (lazy), write-through, TTL on every key
```

```
# Prevent stampede: add jitter to TTLs, use a lock/single-flight on miss
```

ElastiCache (Redis/Valkey or Memcached) accelerates reads, stores sessions, rate-limit counters, leaderboards, and pub/sub. The common pattern is cache-aside: read from cache, on miss read the DB and populate the cache with a TTL. Guard against a **cache stampede** (many concurrent misses hammering the DB when a hot key expires) by adding random jitter to TTLs and using a lock or single-flight so only one request rebuilds the value. Manage staleness with appropriate TTLs or explicit invalidation on write.

Production caching concerns: stampede/thundering-herd on expiry (jittered TTL + single-flight), staleness (TTL vs write-through vs invalidation trade-off), and what happens if the cache is down (the DB must absorb the load — size for it or use a circuit breaker). Use Redis cluster mode + Multi-AZ for HA, and never store the only copy of data in a cache.

Official docs

- [Amazon ElastiCache](#)
- [AWS Builders' Library: Caching challenges and strategies](#)

Q78 How do you monitor and tune RDS/Aurora performance? (Performance Insights, slow queries)

Databases

Spoken answer · ~84 s

I monitor and tune database performance with a loop of measuring, finding the bottleneck, and fixing it. Start with CloudWatch metrics: CPU, freeable memory, database connections, read and write IOPS and latency, queue depth, replica lag, and free storage. Enhanced Monitoring adds operating-system metrics at fine granularity. Performance Insights is the key tool: it shows database load as average active sessions, broken down by wait events, SQL statements, users, and hosts, so you see what the database is waiting on and which queries consume the load. Enable slow query logging and, for PostgreSQL, `pg_stat_statements`, and export logs to CloudWatch. Then tune. For slow queries: examine execution plans with `EXPLAIN ANALYZE`, add or fix indexes, remove unused indexes, rewrite inefficient queries, and avoid N+1 patterns. For waits on locks, shorten transactions and fix contention. For connection pressure, use pooling or RDS Proxy. For IO bottlenecks, increase provisioned IOPS or move to `gp3` or `io2`, or reduce IO through indexing and caching. For memory, adjust buffer parameters via parameter groups. Update statistics and manage vacuum and bloat in PostgreSQL. Add alarms on CPU, connections, replica lag, and storage; DevOps Guru for RDS can detect anomalies. Load test changes, and review regularly as data grows.

Likely follow-ups

- What does Performance Insights show that CloudWatch does not?
- How do you find the slow queries causing load?

```
# Enable Performance Insights (top SQL, wait events, load):
```

```
aws rds modify-db-instance --db-instance-identifier prod --enable-performance-insights
```

```
# Enhanced Monitoring (OS-level metrics) + CloudWatch:
```

```
aws rds modify-db-instance --db-instance-identifier prod --monitoring-interval 60 --monitoring-role-arn arn:...
```

```
# Slow query log to CloudWatch Logs:
```

```
aws rds modify-db-instance --cloudwatch-logs-export-configuration EnableLogTypes=postgresql
```

Performance Insights is the primary tool: it visualizes database load by wait event and shows the top SQL statements driving load, so you can see whether you are CPU-bound, I/O-bound, lock-bound, or waiting on a specific query. Enhanced Monitoring adds OS-level metrics (process list, memory, I/O). Export slow query

logs to CloudWatch, and watch key CloudWatch metrics: CPU, FreeableMemory, Read/WriteLatency, DiskQueueDepth, and DatabaseConnections.

Tuning workflow: use Performance Insights to find the top wait + top SQL, then fix at the source — add an index, rewrite the query, add a read replica or cache for read-heavy load, or scale up if genuinely resource-bound. High connections + low throughput suggests connection pooling (RDS Proxy). Lock waits point to long transactions. Always tie a CloudWatch alarm to replica lag and connection count.

Official docs

- [PostgreSQL: Using EXPLAIN](#)
- [Amazon RDS User Guide](#)
- [Amazon Aurora User Guide](#)

Q79 How do you achieve database high availability and disaster recovery on AWS?

Databases

Spoken answer · ~84 s

Database HA and DR address different failures. For high availability inside a region, use Multi-AZ: RDS Multi-AZ keeps a synchronous standby in another zone and fails over automatically in a minute or two; Multi-AZ clusters with two readable standbys fail over faster; Aurora spreads storage across three zones and promotes a replica in about thirty seconds. Use the cluster or instance endpoint with DNS-based failover, and make clients reconnect and retry, with short DNS caching. For disaster recovery across regions: RDS cross-region read replicas can be promoted, with asynchronous replication and potential data loss measured by lag; Aurora Global Database replicates storage-level with typically under a second of lag and allows managed failover with an RPO in seconds and an RTO of about a minute; DynamoDB global tables are multi-region, active-active. Backups provide the last line: automated backups with point-in-time recovery, manual snapshots copied across regions and accounts, and AWS Backup for policy-based protection, with vault lock for ransomware resistance. Define RTO and RPO for each database and choose accordingly. Document and automate the failover runbook, including promoting, repointing applications, and failing back. Test failover and restores regularly, monitor replica lag, and protect against human error with deletion protection and restricted permissions.

Likely follow-ups

- What determines the RPO of a cross-region replica?
- Why test restores and failovers regularly?

```
# Multi-AZ (synchronous standby, automatic failover - HA within a region):
```

```
aws rds modify-db-instance --db-instance-identifier prod --multi-az
```

```
# Cross-region read replica / Aurora Global Database (DR across regions):
```

```
aws rds create-db-cluster --global-cluster-identifier global --source-region us-east-1 ...
```

Automated + manual snapshots, cross-region snapshot copy

Distinguish HA from DR. HA within a region: RDS Multi-AZ keeps a synchronous standby in another AZ with automatic failover (no data loss, ~60s); Aurora replicates storage across 3 AZs and promotes a reader on failure. DR across regions: cross-region read replicas or Aurora Global Database (sub-second replication, fast regional promotion) protect against a regional outage. Automated backups + cross-region snapshot copies provide point-in-time recovery.

Match the mechanism to RTO/RPO: Multi-AZ for in-region HA (automatic, near-zero RPO). For regional DR, Aurora Global Database gives the lowest RTO/RPO (typically <1s lag, <1min promotion); cross-region replicas/snapshots are cheaper with higher RTO. Test failover and document the promotion runbook — automatic in-AZ, often manual cross-region.

Official docs

- [AWS Well-Architected: Reliability pillar](#)
- [AWS Disaster Recovery of workloads \(whitepaper\)](#)

Q80 How does CloudWatch work? Explain metrics, alarms, dashboards, and custom metrics.

Monitoring

Spoken answer · ~79 s

CloudWatch is the core AWS monitoring service. Metrics are time-ordered data points identified by namespace, name, and dimensions. AWS services publish standard metrics, typically at one-minute or five-minute granularity; detailed monitoring and high-resolution custom metrics reach one-second resolution. Custom metrics are published by applications through the PutMetricData API, the CloudWatch agent, or the embedded metric format in logs, which turns structured log lines into metrics efficiently. Be careful with high-cardinality dimensions, which increase cost and make aggregation hard. Alarms evaluate a metric, or a math expression, against a threshold over a number of periods and move between OK, ALARM, and INSUFFICIENT_DATA; they can notify through SNS, trigger Auto Scaling actions, or EC2 actions. Choose statistics like percentiles for latency, set how missing data is treated, and use anomaly detection for dynamic baselines and composite alarms to combine signals and cut noise. Dashboards are customisable views of metrics, logs, and alarms, shareable across accounts and regions. Related features: Logs and Logs Insights for querying, Container Insights, synthetic canaries, and cross-account observability. Good practice: alarm on user-impacting symptoms and saturation, attach runbooks, define dashboards and alarms as code, and review noise regularly.

Likely follow-ups

- Why is high-cardinality dimension use a problem for custom metrics?
- How do composite alarms reduce alert noise?

Alarm on a metric (e.g. high CPU) -> SNS / Auto Scaling action:

```
aws cloudwatch put-metric-alarm --alarm-name high-cpu --metric-name CPUUtilization --  
namespace AWS/EC2 --statistic Average --period 300 --threshold 80 --comparison-operator  
GreaterThanThreshold --evaluation-periods 2 --alarm-actions arn:aws:sns:...
```

```
# Publish a custom (business) metric:
```

```
aws cloudwatch put-metric-data --namespace MyApp --metric-name OrdersPlaced --value 1 --  
unit Count
```

```
# Composite alarm to reduce noise (alarm only if A AND B)
```

CloudWatch collects metrics (AWS service metrics free; custom metrics you publish), evaluates alarms against thresholds over evaluation periods, and triggers actions (SNS notification, Auto Scaling, EC2 recovery). Dashboards visualize them. Custom metrics let you alarm on business/app signals (orders/sec, queue depth). Composite alarms combine conditions to cut noise, and anomaly-detection alarms learn normal bands instead of fixed thresholds.

Good alerting: alert on symptoms users feel (latency, error rate, saturation) not every low-level metric; use evaluation periods and M-of-N to avoid flapping; composite alarms to reduce noise; and anomaly detection for metrics without an obvious fixed threshold. Tie alarms to runbooks. Embedded Metric Format lets you emit metrics from logs cheaply.

Official docs

- [Prometheus: Instrumentation best practices](#)
- [Amazon CloudWatch User Guide](#)
- [Grafana documentation](#)

Q81 How do you implement centralized logging on AWS for EC2, containers, and Lambda?

Monitoring

 Spoken answer · ~85 s

Centralised logging means every source ships logs to a durable, searchable, access-controlled place. For EC2, install the CloudWatch agent, or Fluent Bit, to collect system and application logs and send them to CloudWatch Logs. For containers, on ECS use the awslogs or FireLens driver, which routes through Fluent Bit to multiple destinations; on EKS run Fluent Bit as a DaemonSet, enriched with Kubernetes metadata, sending to CloudWatch Logs, OpenSearch, or S3. Lambda sends logs to CloudWatch automatically. Use structured JSON logging with correlation IDs, so logs are queryable and traceable. For a multi-account organisation, centralise into a dedicated log-archive account: use subscription filters on log groups to stream to Kinesis Firehose and deliver to S3, plus OpenSearch for search, or cross-account observability for querying. Organisation-wide sources like CloudTrail, VPC flow logs, and ALB access logs go directly to S3. Manage costs and compliance: set retention per log group, filter noisy debug logs, store long-term data in S3 with lifecycle rules, and sample verbose logs. Protect logs: encrypt with KMS, restrict access, enable immutability for audit logs, and mask sensitive data before shipping. Query with Logs Insights or Athena, build metric filters and alerts from logs, and make log groups part of infrastructure as code.

Likely follow-ups

- Why use structured logging with correlation IDs?
- How do you aggregate logs from many accounts?

```
# CloudWatch Logs is the hub; agents/drivers ship logs:
```

```
EC2:    CloudWatch agent -> log groups
```

```
EKS:    Fluent Bit DaemonSet -> CloudWatch Logs / OpenSearch
```

```
Lambda: automatic -> /aws/lambda/fn log group
```

```
# Query across logs with Logs Insights:
```

```
fields @timestamp, @message | filter @message like /ERROR/ | stats count() by bin(5m)
```

```
# Metric filter -> turn a log pattern into a metric + alarm
```

Centralize on CloudWatch Logs (or stream to OpenSearch for full-text search/Kibana, or S3 for cheap retention + Athena). On EC2 use the CloudWatch agent; on EKS run Fluent Bit as a DaemonSet to ship container logs; Lambda logs automatically. Use CloudWatch Logs Insights for ad-hoc queries, metric filters to convert log patterns (e.g. count of ERROR) into alarmable metrics, and subscription filters to stream logs to other systems in real time.

Make logs useful + affordable: emit structured JSON logs (queryable fields), set retention per log group (logs default to never-expire and quietly cost money), and archive to S3 for long-term/compliance with lifecycle to Glacier. Use metric filters to alert on log-derived signals, and correlate logs with traces via a request/trace ID.

Official docs

- [AWS Lambda Developer Guide](#)

Q82 What is distributed tracing on AWS (X-Ray / OpenTelemetry) and why does it matter?

Monitoring

Spoken answer · ~83 s

Distributed tracing follows a single request as it passes through multiple services, recording each step as a span with timing and metadata, linked by a trace ID propagated in headers. In microservices a single user action may touch ten services, and metrics and logs alone can't show where time or errors originate. Traces give the call graph, per-hop latency, error locations, and dependency maps. On AWS, X-Ray is the managed backend, with a service map and trace analytics; the recommended instrumentation is OpenTelemetry through the AWS Distro for OpenTelemetry, which is vendor-neutral and can export to X-Ray or other backends. Components: instrument the application with the SDK or auto-instrumentation, run a collector as a sidecar, DaemonSet, or layer, grant IAM permissions, and ensure context propagation across HTTP, queues, and Lambda. Use sampling to control cost, retaining all errors and slow traces while sampling healthy traffic. Add useful attributes, such as customer tier or deployment version, and link traces to logs through the trace ID, so you can pivot between them. Tracing matters for finding bottlenecks, understanding dependencies, debugging intermittent failures, validating deployments, and measuring SLO contributors. Its limits: instrumentation coverage gaps, and cost and overhead, which need managing.

Likely follow-ups

- What does trace context propagation involve?
- How do sampling strategies preserve the traces that matter?

```
# Instrument with the AWS Distro for OpenTelemetry (ADOT) or X-Ray SDK:
```

```
# Each request gets a trace ID propagated across services -> a service map:
```

```
API GW -> Lambda -> DynamoDB / RDS, with timing on each hop
```

```
# Enable X-Ray on Lambda:
```

```
aws lambda update-function-configuration --function-name api --tracing-config Mode=Active
```

Distributed tracing follows a single request as it traverses multiple services, attaching a trace ID and recording timing at each hop. X-Ray (and the vendor-neutral OpenTelemetry via ADOT) builds a service map and breaks down latency per segment, so in a microservices/serverless system you can see exactly which downstream call is slow or failing — something logs and per-service metrics alone cannot show.

Why it matters: in distributed systems, 'the API is slow' is useless without knowing which of ten services caused it. Tracing localizes the slow/failing hop, reveals fan-out and N+1 calls, and ties together logs/metrics by trace ID (the three pillars of observability). Prefer OpenTelemetry (ADOT) for portability across vendors. Sample to control cost while keeping error/slow traces.

Official docs

- [AWS X-Ray](#)
- [OpenTelemetry: Signals \(traces, metrics, logs\)](#)
- [OpenTelemetry documentation](#)

Q83 How do you define and monitor SLOs, SLIs, and error budgets on AWS?

Monitoring

Spoken answer · ~86 s

I begin with the user journeys that matter and define SLIs, measurable indicators of good service, usually the ratio of good events to valid events: availability as successful requests over total, and latency as requests faster than a threshold over total. Then set SLOs, the targets, such as ninety-nine point nine percent of requests succeed over thirty days, and ninety-five percent complete within three hundred milliseconds. The error budget is the allowable unreliability: one minus the SLO, which becomes the permitted failure that gates risk-taking. On AWS, SLIs come from ALB metrics (request counts, 5xx, target response time), API Gateway, CloudFront, or application metrics from CloudWatch; for precision, measure close to the user with synthetic canaries and real user monitoring. CloudWatch Application Signals provides SLO definitions and burn-rate tracking natively; alternatives are Prometheus with tools like Sloth, or Grafana. Alert on burn rate rather than raw thresholds: multi-window, multi-burn-rate alerts page when the budget is being consumed fast, for example fourteen times the sustainable rate over an hour, and ticket on slow burns. Dashboards show remaining budget. Use the budget in decisions: when plenty remains, ship faster; when exhausted, prioritise reliability work. Review SLOs with product, adjust them over time, and ensure dependencies' SLOs fit within yours.

Likely follow-ups

- What is a burn-rate alert?
- How does the error budget guide release decisions?

SLI: a measured signal (e.g. % of requests < 300ms AND 2xx/3xx)

SLO: the target (e.g. 99.9% over 30 days). Error budget = 0.1% = ~43m/month

Compute from ALB/CloudWatch metrics or Application Signals:

```
good = sum(2xx+3xx under latency); total = sum(all); availability = good/total
```

```
# CloudWatch Application Signals can track SLOs natively now
```

SLIs are the measured indicators users care about (availability, latency, error rate); an SLO is the target for an SLI over a window (e.g. 99.9% of requests succeed under 300ms over 30 days); the error budget is the allowed failure (100% minus the SLO). On AWS, derive SLIs from ALB/CloudWatch metrics or X-Ray, alert on **budget burn rate** (fast burn = page now, slow burn = ticket), and use the budget to balance reliability work against feature velocity.

Alert on burn rate, not raw thresholds: multi-window burn-rate alerts (e.g. 2% of budget in 1h) catch real user impact while ignoring harmless blips, reducing pager fatigue. The error budget creates a shared, objective rule: if the budget is spent, prioritize reliability over features. CloudWatch Application Signals can now track service SLOs directly.

Official docs

- [Google SRE Book: Service Level Objectives](#)
- [Google SRE Book: Embracing Risk \(error budgets\)](#)

Q84 How do you monitor and observe an EKS cluster end to end?

Monitoring

Spoken answer · ~81 s

End-to-end EKS observability covers the control plane, nodes, workloads, and the application, across metrics, logs, and traces. Metrics: Container Insights with the CloudWatch agent, or Prometheus with Grafana, either self-managed or via Amazon Managed Service for Prometheus and Managed Grafana. Collect kube-state-metrics for object state, node exporter or cAdvisor for node and container resources, and application metrics. Control plane logs, such as API server, audit, and authenticator, can be enabled and sent to CloudWatch. Logs: Fluent Bit as a DaemonSet shipping container logs with Kubernetes metadata to CloudWatch, OpenSearch, or S3. Traces: OpenTelemetry instrumentation with the ADOT collector exporting to X-Ray or another backend. Events: capture Kubernetes events as signals. Key signals to watch: pod restarts, crash loops, OOM kills, pending pods, node readiness and pressure conditions, CPU throttling, HPA behaviour, PersistentVolume usage, API server latency, and ingress or ALB error rates and latency. Alerts should focus on user-facing SLOs and actionable conditions, with runbooks, with lower-level resource alerts as diagnostics. Build dashboards at cluster, namespace, and service levels. Also monitor networking issues like CoreDNS errors, and cost per namespace using labels. Manage the observability stack as code and watch its own health and cost.

Likely follow-ups

- Which Kubernetes signals most often reveal problems?
- Why enable control plane audit logs?

```
# Managed Prometheus + Grafana, or Container Insights:
```

```
aws amp create-workspace --alias prod # Amazon Managed Prometheus
```

```
# Container Insights (CloudWatch) for cluster/pod metrics + logs:
```

```
aws eks ... + CloudWatch agent / ADOT collector as a DaemonSet
```

```
# Key signals: node CPU/mem, pod restarts, pending pods, HPA, control-plane logs
```

```
aws eks update-cluster-config --name prod --logging '{"clusterLogging":[{"types":["api","audit","authenticator"],"enabled":true}]}'
```

Cover all layers: control plane (enable EKS control-plane logging — api/audit/authenticator to CloudWatch), nodes (CPU/memory/disk, NotReady), and workloads (pod restarts/CrashLoopBackOff, pending pods, HPA behavior, container CPU/memory vs limits). Use Amazon Managed Prometheus + Managed Grafana (or CloudWatch Container Insights) for metrics, Fluent Bit for logs, and ADOT/X-Ray for traces. kube-state-metrics and node-exporter feed the standard dashboards/alerts.

EKS-specific alerts that matter: pods Pending (out of IPs or node capacity), CrashLoopBackOff, OOMKilled containers (limits too low), nodes NotReady, HPA at max replicas, and PersistentVolume issues. Enable audit logging for security/forensics. Container Insights gives a fast managed start; Managed Prometheus/Grafana gives the full Kubernetes-native experience.

Official docs

- [Amazon EKS: User guide](#)

Q85 What is EventBridge and how do you use it for event-driven architecture and automation?

Monitoring

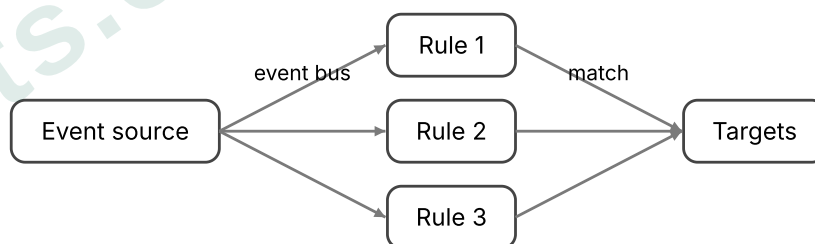
Spoken answer · ~83 s

EventBridge is a serverless event bus that routes events from sources to targets based on rules. Sources include AWS services emitting events, such as EC2 state changes, S3 events, and GuardDuty findings; your own applications via PutEvents; and SaaS partners. Rules match event patterns by content, filtering on fields, and route to targets like Lambda, SQS, SNS, Step Functions, API destinations, or other buses, with input transformation. Schedules via EventBridge Scheduler run cron-like or one-time tasks. Other features: schema registry for discovering event structures, archive and replay to reprocess past events, cross-account and cross-region buses, and Pipes to connect sources like SQS or DynamoDB streams to targets with filtering and enrichment. For event-driven architecture, services publish domain events, such as order placed, without knowing who consumes them; consumers subscribe with rules, so you add behaviour without changing producers, giving loose coupling. For automation, react to operational events: auto-remediate a non-compliant resource flagged by Config, quarantine an instance after a GuardDuty finding, or notify on deployment state changes. Reliability practices: delivery is at-least-once, so make consumers idempotent; configure retry policies and dead-letter queues per target; version event schemas; and monitor failed invocations. Compared with SNS, it offers richer filtering and integrations.

Likely follow-ups

- How is EventBridge different from SNS?
- Why configure a dead-letter queue on targets?

EventBridge



Rule matching AWS or custom events -> targets (Lambda, SQS, Step Functions...):

```
aws events put-rule --name ec2-stop --event-pattern \'{"source":["aws.ec2"],"detail-type":["EC2 Instance State-change Notification"],"detail":{"state":["stopped"]}}\'
```

```
aws events put-targets --rule ec2-stop --targets Id=1,Arn=arn:aws:lambda:...
```

Scheduler for cron-like jobs; custom event bus for app events

EventBridge is a serverless event bus that routes events from AWS services, SaaS partners, and your own apps to targets (Lambda, SQS, SNS, Step Functions, etc.) based on content-filtering rules. It powers two big patterns: operational automation (react to AWS events — auto-remediate, tag, notify) and application event-driven architecture (decouple microservices via a custom event bus). EventBridge Scheduler replaces cron for scheduled invocations with retries and dead-letter support.

EventBridge vs SNS/SQS: use EventBridge when you need rich content-based routing, many AWS/SaaS sources, schema registry, and fan-out to many target types; SNS for simple pub/sub fan-out; SQS for buffered point-to-point work queues. Add a DLQ to rules so unmatched/failed deliveries are not lost, and use the schema registry to keep producers/consumers in sync.

Official docs

- [Amazon EventBridge](#)

Q86 How do you do automated remediation and self-healing on AWS?

Monitoring

Spoken answer · ~79 s

Self-healing means detecting a known failure and fixing it automatically, within safe limits. Layers. Infrastructure level: Auto Scaling groups replace instances failing health checks; ECS and EKS restart failed containers and reschedule pods; load balancers remove unhealthy targets; Multi-AZ databases fail over automatically; Route 53 health checks trigger DNS failover. Event-driven remediation: CloudWatch alarms or EventBridge rules trigger Lambda functions, Step Functions, or Systems Manager Automation runbooks to act, for example restarting a service, increasing capacity, clearing a disk, rotating a credential, isolating an instance flagged by GuardDuty, or reverting a non-compliant security group detected by AWS Config, using Config's automatic remediation. Systems Manager State Manager keeps desired configuration enforced. At the application level, use health checks, circuit breakers, retries with backoff, and graceful degradation. Safeguards matter: automation can cause outages if it misfires, so build in rate limits, blast-radius controls, approval for risky actions, idempotency, dry-run modes, audit logging, and notifications that tell humans what happened. Start by automating the most frequent, well-understood runbook steps, and measure effectiveness. Always investigate root causes, because auto-healing can mask recurring issues; track how often it triggers. Test remediations through game days.

Likely follow-ups

- What safeguards keep automated remediation from causing outages?
- Why can self-healing hide underlying problems?

```
# EC2 auto-recovery alarm (recover impaired instance):
```

```
aws cloudwatch put-metric-alarm --alarm-name recover --metric-name StatusCheckFailed_System  
--namespace AWS/EC2 --alarm-actions arn:aws:automate:us-east-1:ec2:recover ...
```

```
# GuardDuty finding -> EventBridge -> Lambda/SSM Automation (isolate, snapshot):
```

```
aws ssm start-automation-execution --document-name AWS-StopEC2Instance ...
```

```
# Config rule + auto-remediation (e.g. re-enable encryption / fix SG):
```

```
aws configservice put-remediation-configurations ...
```

Self-healing combines detection with automated action. Built-in: ASG replaces unhealthy instances, EC2 alarm-based auto-recovery, ECS/EKS restart failed tasks/pods. Event-driven: GuardDuty/Config/CloudWatch → EventBridge → Lambda or SSM Automation documents to remediate (isolate a compromised instance, revert an open Security Group, re-enable encryption, restart a service). AWS Config rules can auto-remediate drift from desired state.

Guardrails for automation: make remediations idempotent and safe, log every action (CloudTrail/SSM), and add a human-approval step for destructive actions. Common wins: auto-close public Security Groups, auto-tag untagged resources, auto-recover impaired instances, and quarantine GuardDuty-flagged instances. Start with notify-only, then promote trusted remediations to automatic.

Official docs

- [Argo CD documentation](#)

Q87 How do you debug a latency spike in a production AWS application?

Monitoring

Spoken answer · ~83 s

I treat a latency spike as a localisation exercise. First, scope it: which endpoints, users, regions, and when it began; check for correlation with deployments, traffic changes, or dependencies. Use percentiles to see if it's the tail or the whole distribution. At the edge, compare CloudFront and ALB metrics, with target response time versus overall time, to see if the delay is before the application, in it, or after it. Use X-Ray or OpenTelemetry traces to identify the slow span: application code, a database call, a downstream service, or an external API. Then investigate the suspect resource. Application: CPU, memory pressure and garbage collection, thread or connection pool exhaustion, throttling from container CPU limits, and cold starts. Database: Performance Insights for slow queries, lock waits, connection saturation, IOPS limits, or replica lag. Network: cross-zone or cross-region hops, NAT gateway saturation, DNS delay, and packet loss. Scaling: autoscaling lag during a traffic surge, or insufficient capacity. Dependencies: throttling by AWS APIs or third parties. Check recent changes via CloudTrail and deployment history; if a release correlates, roll back first. Mitigate with scaling, cache, or load shedding, then fix the root cause, and add alerts on latency SLOs and a post-incident review.

Likely follow-ups

- How do you tell whether the delay is in the application or the network?
- What is your first step if a deployment correlates with the spike?

```
# 1. Confirm + localize with the service map (X-Ray/traces):
```

which segment got slow? API GW? a downstream service? the DB?

2. ALB/Target metrics: TargetResponseTime, 5xx, RejectedConnectionCount

3. Resource saturation: CPU, memory, DB connections, queue depth

4. DB: Performance Insights top SQL + waits; replica lag

5. Correlate with recent deploys/config changes/traffic

Work outside-in. Confirm user impact (ALB TargetResponseTime/p99, error rate), then use traces to localize which hop slowed down — front end, a specific microservice, or the database. Check saturation on that component (CPU, memory, connection pool, queue depth, Lambda concurrency/throttles). For DB-driven spikes, Performance Insights shows the top SQL and wait events. Always ask 'what changed?' — most spikes follow a deploy, config change, or traffic surge.

Common AWS latency causes: a slow/locking DB query or replica lag, connection-pool exhaustion (add RDS Proxy), Lambda cold starts or concurrency throttling, an undersized/over-scaled tier, a NAT or DNS bottleneck, or a downstream third-party call. Tracing + Performance Insights together usually pinpoint it fast. Mitigate first (scale out, roll back), then root-cause.



Start here (official docs)

- [AWS documentation](#)

Q88 How do you set up effective alerting that avoids both missed incidents and pager fatigue?

Monitoring

🗣 Spoken answer · ~88 s

Effective alerting starts from the principle that a page should mean a human must act now on a user-impacting problem. I base paging alerts on symptoms against SLOs: error rate and latency for key journeys, using multi-window burn-rate alerts that fire fast for severe burns and slowly for gradual ones, with few false alarms. Cause-based signals, such as CPU or memory, usually go to dashboards or tickets unless they predict imminent impact, such as disk about to fill. To avoid blind spots, cover the golden signals for every service, add synthetic canaries for critical user paths, and monitor the monitoring: heartbeat alerts that fire if metrics stop arriving, and treat missing data deliberately. Reduce noise: group related alarms with composite alarms, deduplicate, add suitable evaluation periods so brief blips don't page, use anomaly detection where baselines vary, and route by severity, paging for critical, chat or tickets for warnings. Every alert needs an owner, a runbook link, and clear context. Run a regular alert review: track pages per shift, the percentage that were actionable, and time to acknowledge; delete or tune alerts that aren't actionable. After each incident, ask whether we were alerted before customers noticed, and add detection where not. Protect on-call health with rotation limits and fair load.

Likely follow-ups

- Why alert on symptoms instead of causes?
- How do you measure the quality of your alerts?

```
# Alert on symptoms (SLO/golden signals), route by severity:
```

```
Page:    user-facing SLO burn (high error rate, latency breach)
```

```
Ticket: slow-burn budget, capacity trending, non-urgent
```

```
# Reduce noise: composite alarms, anomaly detection, M-of-N evaluation
```

```
aws cloudwatch put-composite-alarm --alarm-rule "ALARM(errors) AND ALARM(latency)"
```

```
# SNS -> on-call tool (PagerDuty/Opsgenie) with escalation
```

Effective alerting pages humans only for things that need immediate action and affect users. Alert on symptoms (the four golden signals / SLO burn rate), not every underlying metric. Tier by severity: page for fast SLO burn, ticket for slow burn or capacity warnings, dashboard for the rest. Cut noise with composite alarms (require multiple conditions), anomaly detection, and M-of-N evaluation periods to avoid flapping. Every page should map to a runbook.

The balance: too-sensitive alerts cause pager fatigue (people ignore real ones); too-loose miss incidents. Use multi-window burn-rate alerts so you page on genuine user impact and ticket on slow degradation. Continuously prune alerts that never lead to action, and review every page in retro: was it actionable? If not, fix or delete it.

Official docs

- [Google SRE Book: Practical Alerting](#)
- [Google SRE Book: Being On-Call](#)

Q89 How does ECS networking work? Explain awsvpc mode, task vs service, and ECS vs EKS trade-offs.

Containers

Spoken answer · ~85 s

ECS runs containers as tasks, grouped into services. A task definition describes the containers, CPU, memory, roles, and networking; a task is a running instance of it, suited to batch or one-off jobs; a service keeps a desired number of tasks running, replaces failed ones, integrates with load balancers, and handles rolling or blue/green deployments. Networking modes: awsvpc is the standard, giving every task its own elastic network interface with a VPC IP address and its own security groups, so tasks are first-class network citizens, with fine-grained security group control, and it's required for Fargate. Bridge mode uses Docker networking with port mapping on the host, and host mode shares the host's network. With awsvpc, watch ENI limits per instance, which restrict task density on EC2, although ENI trunking helps. Load balancers target task IPs directly. Compared with EKS: ECS is simpler, AWS-native, has no control plane charge, deeper integration with IAM and CloudWatch, and a lower learning curve; EKS offers Kubernetes' ecosystem, portability, richer scheduling and extensibility, operators, and service meshes, at the cost of more complexity and a control plane fee. Choose ECS when the team wants straightforward container orchestration on AWS; choose EKS for Kubernetes skills, portability, or advanced needs.

Likely follow-ups

- What does awsvpc mode give each task?
- When would you pick ECS over EKS?

```
# awsvpc mode – each task gets its own ENI + private IP + security group:
```

```
"networkMode": "awsvpc"    # task-level isolation, SGs per task (recommended)
```

```
# bridge/host modes = older, share the host network (port conflicts)
```

```
# Task definition (the blueprint) vs Service (keeps N tasks running):
```

```
aws ecs register-task-definition --cli-input-json file://task.json
```

```
aws ecs create-service --cluster prod --task-definition app:3 \\  
    --desired-count 4 --load-balancers targetGroupArn=tg-  
arn,containerName=app,containerPort=8080
```

A **task definition** is the immutable blueprint (image, CPU/memory, env, IAM task role); a **task** is a running instance of it; a **service** maintains a desired count, registers with a load balancer, and replaces failed tasks. In **awsvpc** mode each task gets its own ENI, private IP, and security group — first-class VPC networking and per-task SG isolation (the recommended mode, required on Fargate). **ECS vs EKS**: ECS is AWS-native, simpler, no control-plane to learn, deeply integrated; EKS gives portable Kubernetes and its ecosystem at the cost of operational complexity.

Interview tip: Distinguish the three nouns clearly (task def = blueprint, task = running copy, service = desired-count manager + LB). Recommend **awsvpc** mode for per-task SGs and clean IP-per-task (and note it consumes ENIs, which has density limits). On ECS vs EKS, the mature take: ECS if you're all-in on AWS and want simplicity; EKS if you need Kubernetes portability/ecosystem/skills. Mention the IAM **task role** (per-task least privilege, like IRSA).

Official docs

- [Amazon EKS: User guide](#)
- [Amazon ECS Developer Guide](#)

Q90 How do you autoscale on EKS? Compare Cluster Autoscaler, Karpenter, and HPA/KEDA.

Containers

Spoken answer · ~81 s

Autoscaling on EKS has two levels: scaling pods and scaling nodes. The Horizontal Pod Autoscaler adjusts the number of replicas for a deployment based on metrics, typically CPU or memory utilisation, or custom metrics through an adapter. KEDA extends this with event-driven scaling from many sources, such as SQS queue depth, Kafka lag, or Prometheus queries, including scale to zero. The Vertical Pod Autoscaler recommends or adjusts resource requests. For nodes: Cluster Autoscaler watches for pending pods and increases the desired size of pre-defined node groups, and removes underused nodes; it's mature but limited by the groups you define, and slower. Karpenter provisions nodes directly through EC2 fleet APIs based on pending pod requirements, picking optimal instance types, including spot, and consolidating underutilised nodes, so it usually scales faster, packs better, and costs less. They work together: HPA or KEDA decides how many pods are needed; when pods can't schedule, the node autoscaler adds capacity. Best practices: set accurate resource requests, since both depend on them; configure PodDisruptionBudgets; use topology spread for zone balance; avoid HPA and VPA fighting over the same metric; set sensible minimums and scale-down stabilisation; and test scaling behaviour under load.

Likely follow-ups

- How does KEDA differ from HPA?
- Why do accurate resource requests matter for autoscaling?

```
# HPA — scale PODS on metrics:
```

```
kubectl autoscale deployment api --cpu-percent=60 --min=3 --max=20
```

```
# Cluster Autoscaler – scale NODES via ASGs (node-group bound):
```

```
# adds nodes when pods are Pending; tied to predefined node groups
```

```
# Karpenter – provisions right-sized nodes directly, fast, no node groups:
```

```
kubectl apply -f nodepool.yaml # picks instance types/Spot to fit pending pods
```

```
# KEDA – event-driven pod scaling (SQS depth, Kafka lag, scale-to-zero):
```

```
kubectl apply -f scaledobject.yaml
```

Two dimensions: scale **pods** and scale **nodes**. **HPA** scales pod replicas on CPU/memory/custom metrics; **KEDA** extends that to event sources (queue depth, Kafka lag) and enables scale-to-zero. For nodes, **Cluster Autoscaler** adds/removes nodes within predefined ASG node groups when pods can't schedule. **Karpenter** is the modern replacement: it provisions right-sized nodes directly from the EC2 fleet (choosing instance types and Spot) to exactly fit pending pods, scales faster, and consolidates underutilized nodes — better bin-packing and cost.

Interview tip: Separate pod scaling (HPA/KEDA) from node scaling (Cluster Autoscaler/Karpenter) — interviewers test whether you conflate them. The current best-practice pairing is **HPA/KEDA + Karpenter**: Karpenter's just-in-time right-sized provisioning and consolidation beat Cluster Autoscaler's fixed node groups on speed and cost. KEDA's scale-to-zero and queue-based scaling is the answer for spiky event workloads.

Official docs

- [Kubernetes: Horizontal Pod Autoscaling](#)
- [Kubernetes Autoscaler: Cluster Autoscaler](#)
- [Amazon EKS best practices guide](#)

Spoken answer · ~82 s

Container security spans the build, the platform, and the runtime. Build: use minimal, trusted base images, such as distroless, pin versions, and run as non-root; scan images for vulnerabilities in CI and continuously in ECR using Amazon Inspector or ECR scanning; sign images and verify signatures at admission; and avoid embedding secrets. Registry: use private ECR repositories with least-privilege policies, immutable tags, and lifecycle rules. Platform: give workloads their own IAM identity, with ECS task roles or EKS IAM Roles for Service Accounts or Pod Identity, never relying on instance roles; grant least privilege. Isolate networks using security groups, security groups for pods, and Kubernetes network policies. On EKS, enforce Pod Security Standards or policy engines like Kyverno or OPA Gatekeeper to block privileged containers, host mounts, and unapproved images; use private cluster endpoints, restrict API access, enable audit logs, and apply RBAC. Secrets come from Secrets Manager or Parameter Store, not environment variables baked into images. Runtime: use GuardDuty runtime monitoring for ECS and EKS to detect suspicious behaviour, read-only root filesystems, dropped Linux capabilities, seccomp profiles, and resource limits. Keep nodes patched using hardened, regularly rebuilt AMIs. Log and monitor, and rehearse incident response.

Likely follow-ups

- Why are task roles better than instance roles for containers?
- What does admission control enforce on EKS?

```
# Scan images in ECR (on push) – block deploys on criticals:
```

```
aws ecr put-image-scanning-configuration --repository-name app --image-scanning-configuration scanOnPush=true
```

```
aws ecr start-image-scan --repository-name app --image-id imageTag=v1 # enhanced = Inspector
```

```
# Least-privilege per pod/task (not the node):
```

```
eksctl create iamserviceaccount ... # IRSA / EKS Pod Identity
```

```
# Pod hardening: non-root, read-only rootfs, drop caps, seccomp:
```

```
securityContext: { runAsNonRoot: true, readOnlyRootFilesystem: true,
```

```
allowPrivilegeEscalation: false, capabilities: { drop: ["ALL"]} }
```

Defense in depth across the lifecycle: **scan images** in ECR (scan-on-push, or Inspector enhanced scanning) and fail builds on critical CVEs; use minimal/distroless base images. Give each **pod/task its own IAM role** (IRSA/Pod Identity, ECS task role) so a compromised container has minimal blast radius — never the node

role. Harden the runtime: non-root user, read-only root filesystem, drop all Linux capabilities, NoNewPrivileges, seccomp/AppArmor profiles. Enforce policy with admission controllers (OPA/Kyverno) and detect at runtime with GuardDuty EKS Runtime Monitoring or Falco. Restrict node IMDS (hop-limit) so pods can't steal node creds.

Interview tip: Hit the lifecycle: build (scan + distroless), deploy (admission policy, per-pod IAM, hardened securityContext), runtime (GuardDuty EKS/Falco). The two highest-value AWS-specific points: per-pod IAM (IRSA/Pod Identity, not the node role) to contain blast radius, and restricting IMDS hop-limit so a compromised pod can't reach the node's credentials. Tie image scanning to a CI gate, not just 'enabled'.

Official docs

- [Trivy](#)
- [Falco documentation](#)

Q92 How does ECR work, and what's a good image strategy (tagging, lifecycle, multi-arch)?

Containers

Spoken answer · ~82 s

ECR is AWS's managed container registry, storing images and OCI artefacts in private or public repositories, integrated with IAM for access and encrypted at rest. Authentication uses temporary tokens obtained with the AWS CLI. A good image strategy covers several things. Tagging: use immutable tags so a tag can't be overwritten, preventing surprise changes; tag with the git commit SHA or a semantic version for traceability, and avoid relying on latest in production; deploy by digest where strictness matters. Lifecycle policies automatically expire old or untagged images based on age or count, controlling storage cost, while keeping recent releases and rollback candidates. Security: enable scan on push or continuous scanning with Inspector, use repository policies, and consider image signing. Replication: cross-region and cross-account replication supports multi-region deployments and disaster recovery, and pull-through cache rules cache upstream public images, avoiding rate limits and improving reliability. Multi-architecture: build multi-arch manifests, linux/amd64 and linux/arm64, with Buildx so the same tag works on x86 and Graviton nodes. Performance: keep images small with multi-stage builds, order layers for caching, and use VPC endpoints for ECR to avoid NAT costs and speed up pulls. Organise repositories per service, and manage them as code.

Likely follow-ups

- Why use immutable tags and commit SHA tags?
- What does a pull-through cache give you?

```
# Auth + push:
```

```
aws ecr get-login-password | docker login --username AWS --password-stdin
ACCT.dkr.ecr.REGION.amazonaws.com
```

```
# Immutable tags so a tag can never be overwritten (supply-chain safety):
```

```
aws ecr put-image-tag-mutability --repository-name app --image-tag-mutability IMMUTABLE
```

```
# Lifecycle policy to expire old/untagged images (cost + clutter):
```

```
aws ecr put-lifecycle-policy --repository-name app --lifecycle-policy-text  
file://lifecycle.json
```

```
# Multi-arch (amd64 + arm64/Graviton) with buildx:
```

```
docker buildx build --platform linux/amd64,linux/arm64 -t REPO:v1 --push .
```

ECR is the managed private registry, IAM-controlled and KMS-encrypted, with scan-on-push and cross-region/cross-account replication. A solid strategy: **immutable tags** (a deployed tag can never change underneath you — pin to the digest for true reproducibility), tag with the git SHA (not just latest), **lifecycle policies** to expire untagged/old images so storage and risk don't pile up, and **multi-arch manifests** so the same tag runs on x86 and Graviton (arm64) for cost/perf. Replicate to other regions for DR and pull-latency.

Interview tip: The standout practices: immutable tags + deploy by digest (kills the 'latest moved under me' and tag-overwrite supply-chain risks), git-SHA tags for traceability, and lifecycle policies to control cost/clutter. Multi-arch images for Graviton savings is a current cost-optimization signal. Mention scan-on-push and cross-region replication for DR. Avoid latest in production deploys.

Official docs

- [Amazon ECR](#)

Q93 **What is a service mesh on EKS (App Mesh / Istio), and when is it worth the complexity?**

Containers

 Spoken answer · ~81 s

A service mesh adds a dedicated infrastructure layer for service-to-service communication, typically through sidecar proxies, or newer sidecarless designs, managed by a control plane. It provides mutual TLS, traffic management such as retries, timeouts, canary routing and fault injection, and uniform observability of requests. On AWS, App Mesh was the managed option using Envoy, but AWS has announced its discontinuation, so options now are Istio, Linkerd, Cilium service mesh, or VPC Lattice, an AWS managed service-to-service networking layer that works across VPCs and compute types without sidecars. Whether it's worth it depends on need. It's valuable when you have many services and need consistent mTLS for zero-trust or compliance, sophisticated traffic shaping, or uniform telemetry across polyglot services without changing code. It's costly because of added latency and resource use from proxies, a complex control plane to operate and upgrade, more complicated debugging, and a learning curve. For fewer services, simpler solutions often suffice: shared libraries for retries, ingress and network policies, cert-manager, and ALB features. My approach is to define the problems first, pilot on a few services, measure overhead, prefer lighter options, and adopt when the benefits clearly exceed the operational cost.

Likely follow-ups

- What alternatives to a sidecar mesh exist on AWS?
- What overheads should you measure in a pilot?

A mesh adds a sidecar proxy (Envoy) per pod to handle service-to-service traffic:

- mTLS between services (encryption + identity) without app changes
- traffic shaping: canary/weighted routing, retries, timeouts, circuit breaking
- observability: per-call metrics/traces at the proxy

Lighter alternatives before a full mesh:

- VPC Lattice (AWS-native service-to-service connectivity/auth)
- Cilium/eBPF for mTLS + network policy without sidecars

A **service mesh** injects a sidecar proxy alongside each pod and moves cross-service concerns out of application code: **mTLS** (mutual auth + encryption), fine-grained **traffic management** (weighted/canary routing, retries, timeouts, circuit breaking, fault injection), and uniform **observability** (golden-signal metrics and traces per hop). It's worth it when you have many services, need zero-trust mTLS everywhere, or want advanced rollout/resilience controls centrally. It's **not** worth it for a handful of services — the operational complexity, latency, and resource overhead of sidecars outweigh the benefit.

Interview tip: Show judgment, not enthusiasm: a mesh's value (mTLS, canary routing, retries/circuit-breaking, per-hop telemetry) scales with the number of services, but sidecars add latency, memory, and a steep operational learning curve. For small fleets, native retries + ALB + IAM/TLS suffice. Mention sidecar-less options (Cilium eBPF, Ambient mesh, AWS VPC Lattice) as the modern trend reducing that overhead. 'Don't add a mesh for 5 services' is the line that lands.

Official docs

- [Istio documentation](#)
- [Amazon EKS: User guide](#)

Q94 How do you handle persistent storage and stateful workloads in containers on AWS?

Containers

Spoken answer · ~84 s

Stateful workloads in containers need durable storage that outlives the container. On EKS, use PersistentVolumes through the EBS CSI driver for block storage, which is zonal and single-attach, suited to databases and single-writer workloads; the EFS CSI driver for shared ReadWriteMany file storage across zones; and FSx for Lustre or NetApp for high-performance needs. StorageClasses define volume types, such as gp3, and parameters, with volume expansion and snapshot support. StatefulSets give pods stable identities and per-pod volumes, ordered startup, and stable network names, which stateful clustered applications need. On ECS, tasks can use EFS volumes, or EBS volumes attached at launch, and avoid relying on the container's ephemeral storage. Design considerations: because EBS volumes are tied to a zone, the pod must reschedule in the same zone, so use volume node affinity and topology-aware provisioning, and plan for zone failure with replication at the application layer or backups. Take snapshots, using VolumeSnapshots or AWS Backup, and test restores. Set resource requests and PodDisruptionBudgets. Consider operators for databases, which automate failover, backup, and upgrades. Often the better choice for critical data is a managed service like RDS, keeping containers stateless. Run stateful workloads in containers only when needed, and with strong operational practices.

Likely follow-ups

- Why does an EBS-backed pod have zone affinity?
- When is a managed database better than running one in a container?

```
# EBS CSI – block storage, single-AZ, ReadWriteOnce (one node):
```

```
storageClassName: ebs-sc # great for databases needing low-latency block I/O
```

```
# EFS CSI – shared NFS, multi-AZ, ReadWriteMany (many pods):
```

```
storageClassName: efs-sc # shared files across pods/AZs (slower than EBS)
```

```
# StatefulSet gives stable identity + per-pod PVC:
```

```
kubectl apply -f statefulset.yaml # stable network id + own volume per replica
```

Containers are ephemeral, so persistent state lives on volumes via CSI drivers. **EBS CSI** attaches block volumes (ReadWriteOnce, AZ-bound, low latency) — ideal for databases, but a pod with an EBS volume is pinned to that volume's AZ. **EFS CSI** provides shared NFS (ReadWriteMany, multi-AZ) for content many pods read/write, at higher latency. Use **StatefulSets** for stable pod identity and per-replica PVCs. The pragmatic AWS answer, though, is often to **not** self-host state in the cluster: use managed RDS/DynamoDB/ElastiCache/S3 and keep the cluster stateless — far less operational risk than running databases on Kubernetes.

Interview tip: Map access modes to drivers: EBS = RWO/single-AZ/fast (databases, pinned to an AZ — a scheduling gotcha), EFS = RWX/multi-AZ/shared (slower). StatefulSets for stable identity + PVC-per-pod. The senior judgment call interviewers want: **prefer managed data services (RDS/DynamoDB/S3) over stateful workloads in-cluster** — keep Kubernetes stateless when you can; reach for in-cluster state only when there's a real reason.

Official docs

- [Amazon EBS](#)
- [Kubernetes: StatefulSets](#)

Spoken answer · ~83 s

Zero-downtime deployments rely on starting new instances healthy before removing old ones, and on graceful shutdown. On ECS or EKS, rolling updates with proper settings: set minimum healthy percent and maximum surge so new tasks start before old ones stop; configure health checks and readiness probes so traffic reaches only ready containers; use the load balancer's connection draining, deregistration delay, and a preStop hook with a short sleep so in-flight requests complete before termination, and handle SIGTERM gracefully in the app. Use PodDisruptionBudgets so voluntary disruptions don't remove too many pods. For higher safety: blue/green deployments with CodeDeploy for ECS, or two target groups on the ALB, shifting traffic after validation; canary releases that route a small percentage first, with Argo Rollouts or CodeDeploy; and feature flags to decouple release from deploy. Add automatic rollback: CloudWatch alarms tied to deployment, such as 5xx rate or latency, trigger rollback, and ECS deployment circuit breakers roll back failed deployments. Database changes must be backward compatible, using expand-contract, so old and new versions run together. Keep previous images and manifests for quick rollback, and rehearse it. Verify with smoke tests after each stage, track deployment metrics, and automate everything in the pipeline.

Likely follow-ups

- What does a preStop hook with a sleep prevent?
- How do ECS deployment circuit breakers work?

```
# Kubernetes rolling update with surge/unavailable controls + probes:
```

```
strategy: { rollingUpdate: { maxSurge: 25%, maxUnavailable: 0 } }
```

```
readinessProbe: ... # don't send traffic until the pod is actually ready
```

```
kubectl rollout status deploy/api; kubectl rollout undo deploy/api # instant rollback
```

```
# ECS rolling or blue/green via CodeDeploy:
```

```
deploymentConfiguration: { minimumHealthyPercent: 100, maximumPercent: 200 }
```

```
# CodeDeploy canary shifts 10% then the rest, auto-rollback on alarms
```

Zero-downtime relies on health-gated gradual replacement. In Kubernetes, a **rolling update** with `maxUnavailable:0` + **readiness probes** ensures new pods receive traffic only when ready and old pods drain (`preStop` + `terminationGracePeriodSeconds`). For richer safety, **blue/green** (deploy a parallel version, switch the service/target group, roll back by switching back) or **canary** (shift a small % first, watch metrics, then proceed). ECS does the same via rolling config or **CodeDeploy** blue/green with canary traffic shifting and automatic rollback on CloudWatch alarms.

Interview tip: The mechanics that matter: readiness probes (gate traffic), `maxUnavailable:0/surge` (always enough healthy capacity), and graceful shutdown (`preStop` + grace period to drain in-flight requests). Then escalate to canary/blue-green with **automatic rollback on alarms** — the senior point is that rollback should be automatic and metric-driven, not a human noticing. ECS+CodeDeploy and Argo Rollouts/Flagger are good names to drop.

 Official docs

- [Kubernetes: Deployments](#)

Q96 **An EKS pod is stuck in Pending / CrashLoopBackOff / ImagePullBackOff. How do you debug each?**

Containers

 Spoken answer · ~86 s

Each status points to a different layer, so I start with `kubectl describe` on the pod and read the events. Pending means the pod isn't scheduled: look for insufficient CPU or memory on nodes, so check resource requests against capacity and whether the autoscaler is adding nodes; taints without matching tolerations; node selectors or affinities that match nothing; unbound PersistentVolumeClaims or volume zone conflicts; or IP exhaustion in subnets with the VPC CNI. Fix by adding capacity, adjusting requests or constraints, or fixing the storage. ImagePullBackOff means the image can't be fetched: check the image name and tag for typos, whether the tag exists, registry authentication, such as ECR permissions on the node role or the pull secret, network reachability to ECR, including NAT or VPC endpoints, and rate limits from public registries. CrashLoopBackOff means the container starts and exits repeatedly: read the logs, including the previous container's logs using the previous flag, and check the exit code; common causes are application errors, missing configuration or secrets, failed dependencies, an OOM kill, shown as exit code 137 with `OOMKilled` in the status, wrong command or entrypoint, and failing liveness probes that kill healthy but slow containers. Fix by correcting configuration, increasing memory limits, or adjusting probes.

Likely follow-ups

- What does exit code 137 indicate?
- Why can a liveness probe cause a crash loop?

```
kubectl describe pod
```

```
# Events at the bottom are the #1 clue
```

```
# Pending -> no node fits: resources, taints, AZ/volume affinity, no capacity
```

```
kubectl get events --sort-by=.lastTimestamp; kubectl top nodes
```

```
# CrashLoopBackOff -> container starts then exits; read its logs:
```

```
kubectl logs
--previous      # logs from the crashed instance
```

```
# ImagePullBackOff -> registry/auth/tag issue:
```

```
# wrong tag, ECR perms on the node/pod role, or cross-account/region pull
```

Always start with `kubectl describe pod` — the Events list usually names the cause. **Pending** = scheduler can't place it: insufficient CPU/memory (check requests vs `kubectl top nodes`), taints without tolerations, node selectors/affinity, or an EBS volume pinning the pod to an AZ with no capacity (then Cluster Autoscaler/Karpenter should add a node — if not, check IAM/limits). **CrashLoopBackOff** = the container keeps exiting: read logs `--previous`, check the command/args, missing config/secrets, failing liveness probe, or OOMKilled (check limits in describe). **ImagePullBackOff** = can't pull the image: wrong tag/digest, ECR permissions on the node/pod IAM role, or wrong region/account.

Interview tip: Give a crisp per-state playbook anchored on `describe pod` Events. Map causes: Pending=scheduling (resources/taints/affinity/AZ-volume/capacity), CrashLoop=app exit (logs `--previous`, config/secrets, OOMKilled, bad probe), ImagePull=registry/auth/tag. The AWS-specific gotchas that score: ECR pull permissions via the pod/node IAM role, and EBS volumes pinning pods to an AZ. OOMKilled (exit 137) hiding behind CrashLoop is a classic catch.

Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Kubernetes: Debug Pods](#)
- [Kubernetes: Images \(pull policy, registries\)](#)

Q97 **Explain the AWS cost levers: Savings Plans vs Reserved Instances vs Spot, and how to choose.**

Cost Opt

🗣 Spoken answer · ~88 s

These are three purchasing options with different commitment and flexibility. On-demand is full price with no commitment. Reserved Instances commit to a specific instance family, region, and term, one or three years, for a discount of up to roughly seventy percent; standard RIs give the deepest discount but least flexibility, while convertible RIs allow exchanges. Savings Plans commit to a dollar amount of hourly usage for one or three years: Compute Savings Plans apply across instance families, sizes, regions, operating systems, and also to Fargate and Lambda, giving the most flexibility with somewhat lower discounts; EC2 Instance Savings Plans give higher discounts but are tied to a family in a region. Savings Plans have largely replaced RIs for flexibility, though RIs remain relevant for RDS, ElastiCache, and others. Spot instances use spare capacity at up to ninety percent off, with interruption risk, for fault-tolerant workloads. To choose: measure steady baseline usage over months, commit conservatively to the stable floor, perhaps sixty to eighty percent of baseline, using Compute Savings Plans for flexibility; use spot for tolerant, flexible, burst capacity; keep on-demand for the variable remainder. Review coverage and utilisation monthly, ladder commitments over time to avoid renewal cliffs, and account for planned migrations, such as to Graviton or serverless, before committing.

Likely follow-ups

- Why do Compute Savings Plans offer more flexibility than reserved instances?
- How much of your baseline should you commit?

On-Demand -> no commitment, full price (baseline / unpredictable)

Reserved Inst -> 1-3yr commit to a specific family/region, up to ~72% off

Savings Plans -> 1-3yr commit to a \$/hr spend, more flexible:

Compute SP -> any region/family/Lambda/Fargate (most flexible, ~66%)

EC2 SP -> locked to a family in a region (bigger discount)

Spot -> spare capacity, up to ~90% off, interruptible

Match commitment to predictability. Cover your **steady baseline** with **Savings Plans** (Compute SP is the modern default — applies across EC2 families/regions, Fargate, and Lambda, so you don't over-commit to one shape) or RIs where you want the deepest discount on a fixed family. Run **fault-tolerant/burst** capacity on **Spot** (up to 90% off, handle interruptions). Leave truly unpredictable or short-lived workloads on **On-Demand**. The strategy: right-size first, then layer commitments on the stable floor and Spot on the variable top.

Interview tip: The framework is baseline vs burst: Savings Plans/RIs on the predictable floor, Spot on the elastic peak, On-Demand for the unpredictable middle. Recommend **Compute Savings Plans** over standard RIs for

flexibility (covers Fargate/Lambda, no family lock-in) unless a deep family-specific RI fits. Critical caveat: **right-size before committing** — a 3-year RI on an oversized instance locks in waste. Mention coverage/utilization reports to avoid under/over-committing.

Official docs

- [AWS Cost Explorer](#)
- [AWS Savings Plans](#)
- [Azure Reservations](#)

Q98 How do you find and eliminate wasted AWS spend? Walk through a cost-optimization audit.

Cost Opt

Spoken answer · ~84 s

A cost audit is systematic. Step one, get visibility: ensure cost allocation tags are activated, then use Cost Explorer and the Cost and Usage Report with Athena to break down spend by service, account, team, and usage type; identify the top five or ten line items, which usually drive most of the bill. Step two, hunt waste: idle or underused EC2 and RDS instances found via utilisation metrics and Compute Optimizer; unattached EBS volumes, old snapshots and AMIs; unused elastic IPs and idle load balancers; forgotten dev and test environments; oversized databases; excessive CloudWatch log retention and ingestion; and orphaned resources from deleted projects. Step three, right-size: move to appropriate instance sizes and newer generations, Graviton where compatible, and gp3 volumes. Step four, optimise purchasing: Savings Plans and reservations for steady load, spot for tolerant workloads, and scheduling for non-production. Step five, storage and data transfer: S3 lifecycle and Intelligent-Tiering, cutting cross-zone and cross-region traffic, NAT gateway costs, and VPC endpoints. Step six, architecture: autoscaling, serverless for spiky loads, caching. Step seven, governance: budgets, anomaly detection, ownership per team, and a monthly review. Prioritise by savings versus effort and risk, change safely, record the savings achieved, and make it a recurring process.

Likely follow-ups

- Which tools surface idle resources?
- Why prioritise by savings versus effort and risk?

```
# Cost Explorer: group by service/tag, find the top movers + anomalies:
```

```
aws ce get-cost-and-usage --granularity MONTHLY --group-by Type=DIMENSION,Key=SERVICE ...
```

```
# Compute Optimizer: right-sizing recommendations from real utilization:
```

```
aws compute-optimizer get-ec2-instance-recommendations
```

```
# Common waste to hunt:
```

```
- idle/oversized EC2 + RDS; unattached EBS volumes + old snapshots
```

- unassociated Elastic IPs; idle NAT Gateways; orphaned load balancers

- S3 with no lifecycle policy; over-provisioned gp2; dev running 24/7

A systematic audit: (1) **Visibility** — Cost Explorer grouped by service and by tag to see where money goes and catch anomalies; enforce cost-allocation tags. (2) **Right-size** — Compute Optimizer/Cost Explorer recommendations to shrink over-provisioned EC2/RDS/EBS based on actual utilization. (3) **Eliminate waste** — delete unattached EBS, old snapshots, idle NAT Gateways/EIPs/LBs, and add S3 lifecycle rules to tier/expire data. (4) **Schedule** — stop non-prod outside business hours. (5) **Commit** — Savings Plans on the right-sized baseline + Spot for burst. (6) **Architect** — Graviton, serverless for spiky loads, reduce cross-AZ/NAT data transfer.

Interview tip: Show an ordered method: visibility (tagging + Cost Explorer) → right-size (Compute Optimizer) → kill waste (unattached EBS, idle NAT/EIP/LB, snapshots, no-lifecycle S3) → schedule non-prod → commit (SP/Spot) → architect (Graviton, serverless, cut data-transfer). The order matters — right-size **before** committing. Naming sneaky line items (NAT Gateway data processing, cross-AZ transfer, idle EIP charges) signals you've actually read a real bill.



Official docs

- [RFC 3022: Traditional NAT](#)
- [Amazon EBS](#)

Spoken answer · ~87 s

Data transfer is expensive because AWS charges for traffic leaving its network or crossing certain boundaries, while ingress is mostly free. Costs arise from internet egress, cross-region transfer, cross-availability-zone transfer, and NAT gateway processing; these are easy to overlook because they accumulate quietly in the architecture. To minimise: keep traffic within a zone where practical, since cross-zone traffic is charged in both directions; but balance with availability, so don't sacrifice resilience for small savings. Use topology-aware routing in Kubernetes and zone-aware load balancing. Avoid unnecessary cross-region replication and chatty cross-region calls. Use a CDN like CloudFront, whose egress to the internet is cheaper than serving directly from EC2 or S3, and it reduces origin traffic through caching. Use VPC gateway endpoints for S3 and DynamoDB, which are free, and interface endpoints where the volume justifies the cost, to avoid NAT gateway data-processing fees. Compress responses and data, use efficient formats, and cache. Place compute near data, and in the same region as storage. For large transfers, consider Direct Connect, where data transfer rates are lower. Analyse the bill by usage type, using Cost Explorer and VPC flow logs to find top talkers, then optimise the largest flows first. Design reviews should consider data movement as a cost dimension.

Likely follow-ups

- Why is cross-AZ traffic a hidden cost in Kubernetes?
- How does a CDN reduce egress cost?

Data transfer cost

Cheap or free	Expensive
Same AZ private IP	Cross-AZ and cross-region
Inbound, VPC endpoints for S3	NAT gateway processing, internet egress

The expensive flows (often surprising on the bill):

Internet egress → data OUT to internet is charged (in is free)

Cross-AZ → traffic between AZs charged BOTH directions (\$0.01/GB each way)

NAT Gateway → per-GB data processing on TOP of egress

Cross-Region → most expensive inter-region

Cheaper paths:

- VPC Gateway Endpoints for S3/DynamoDB = FREE (avoid NAT + egress)

- CloudFront for internet egress (cheaper + cached); keep traffic same-AZ

Data transfer is metered along many edges and adds up silently. **Internet egress** is charged per GB (ingress is free). **Cross-AZ** traffic is charged in both directions — chatty multi-AZ services (or a load balancer routing across AZs to backends) can rack up real cost. **NAT Gateway** charges a per-GB *processing* fee on top of egress, so routing S3/package traffic through NAT is doubly wasteful. To minimize: use **VPC Gateway Endpoints** for S3/DynamoDB (free, bypasses NAT and egress), **Interface Endpoints/PrivateLink** to keep traffic private, **CloudFront** for cheaper cached internet egress, keep chatty traffic within an AZ, and compress/batch.

Interview tip: The points that reveal real bill experience: cross-AZ is charged **both ways** (chatty cross-AZ chatter + the NAT data-processing fee are top hidden costs), and **S3/DynamoDB Gateway Endpoints are free** and remove both NAT processing and egress for that traffic — a frequent quick win. CloudFront for egress, AZ-affinity for chatty services, and PrivateLink to avoid NAT round-trips round it out. '\Ingress free, egress paid\' is the baseline fact.



Official docs

- [RFC 3022: Traditional NAT](#)
- [AWS PrivateLink](#)

Q100 How do you optimize S3 storage costs? Explain storage classes, lifecycle, and Intelligent-Tiering.

Cost Opt

🗣 Spoken answer · ~83 s

S3 cost optimisation combines the right storage class, lifecycle automation, and housekeeping. Storage classes: Standard for hot data; Intelligent-Tiering automatically moves objects between frequent, infrequent, and archive access tiers based on usage, with a small monitoring charge per object, ideal when access patterns are unknown; Standard-IA and One Zone-IA for infrequent access with retrieval fees and minimum durations; Glacier Instant Retrieval, Flexible Retrieval, and Deep Archive for increasingly cold data at lower prices with longer retrieval times. Lifecycle policies automate transitions and expiry: for example, logs to Standard-IA after thirty days, to Glacier after ninety, and deletion after a year. Also expire noncurrent versions of versioned objects, delete expired delete markers, and abort incomplete multipart uploads, which silently accumulate charges. Use Storage Lens and Storage Class Analysis to find where the bytes are and what is accessed. Watch for traps: small objects are poorly suited to IA classes and Intelligent-Tiering monitoring fees; early deletion fees on minimum-duration classes; retrieval costs if cold data is accessed unexpectedly; and request costs for transitions on billions of objects. Reduce volume by compression, columnar formats, deduplication, and deleting unneeded data. Evaluate the access pattern first, apply Intelligent-Tiering when unsure, and review quarterly.

Likely follow-ups

- What traps make cold storage classes more expensive than expected?
- Why abort incomplete multipart uploads?

Storage classes by access pattern:

Standard -> frequent access

Standard-IA / One Zone-IA -> infrequent (lower storage, retrieval fee)

Glacier Instant / Flexible / Deep Archive -> archive (cheapest, slower retrieval)

Intelligent-Tiering -> auto-moves objects between tiers by access (no retrieval fees)

Lifecycle policy: transition + expire automatically:

```
aws s3api put-bucket-lifecycle-configuration --bucket b --lifecycle-configuration file://lc.json
```

Match the **storage class** to the access pattern: Standard for hot data, Standard-IA for infrequent (cheaper storage, per-GB retrieval fee), Glacier tiers for archives (cheapest, retrieval latency from instant to hours), One Zone-IA for reproducible data you can afford to lose with an AZ. **Lifecycle policies** automate transitions

(e.g. logs: Standard → IA at 30d → Glacier at 90d → expire at 365d) and clean up incomplete multipart uploads and old versions. When access patterns are **unpredictable**, **Intelligent-Tiering** auto-moves objects between frequent/infrequent/archive tiers with no retrieval fees (small monitoring charge) — set-and-forget.

Interview tip: Lead with 'class to access pattern + lifecycle to automate'. The most-tested judgment: use **Intelligent-Tiering** when access is unpredictable (it avoids the IA/Glacier retrieval-fee surprise by moving data automatically), and use explicit lifecycle transitions when you *know* the pattern (logs aging out). Don't forget cleanup rules (incomplete multipart uploads, old noncurrent versions silently cost money). One Zone-IA only for reproducible data.

Official docs

- [Kubernetes: StorageClasses](#)
- [Amazon S3: Lifecycle management](#)
- [Amazon S3 User Guide](#)

Q101 What is FinOps, and how do you build cost accountability across engineering teams?

Cost Opt

Spoken answer · ~84 s

FinOps is a practice that brings financial accountability to cloud spending, through collaboration between engineering, finance, and product, so teams make spending decisions based on value. The cycle is inform, optimise, operate. Inform: make costs visible and attributable. Enforce tagging, or use account structure and Kubernetes labels, to allocate cost to teams, services, and environments; use showback or chargeback so teams see their spend; provide dashboards and unit-cost metrics, such as cost per customer or per thousand requests, which relate cost to business value rather than raw totals. Optimise: teams own optimisation of their services, supported by central guidance: right-sizing recommendations, commitment purchasing managed centrally to get the best discounts across the organisation, waste reports, and engineering best practices. Operate: set budgets and alerts per team, anomaly detection, regular cost reviews in engineering rituals, and include cost in architecture decisions and design reviews. Build accountability by making each team the owner of its costs, recognising and rewarding efficiency, and avoiding blame. A central FinOps function or platform team provides tooling, forecasting, and governance. Be careful to balance cost against speed and reliability: the goal is value, not simply cutting. Track metrics such as commitment coverage, waste percentage, and forecast accuracy, and iterate.

Likely follow-ups

- What is the difference between showback and chargeback?
- Why use unit economics instead of total spend?

```
# Foundation: consistent cost-allocation tags (enforced, not optional):
```

```
tags: { team: payments, env: prod, service: checkout, cost-center: 4021 }
```

```
# Enforce tags at creation with SCP / Config / tag policies:
```

```
aws organizations create-policy --type TAG_POLICY ...
```

```
# Show teams their spend + alert on anomalies:
```

```
aws ce get-cost-and-usage --group-by Type=TAG,Key=team ... # per-team chargeback
```

```
aws ce create-anomaly-monitor ... # automatic spend-spike detection
```

```
aws budgets create-budget ... # per-team budgets + alerts
```

FinOps is the practice of bringing financial accountability to cloud spend — making engineering, finance, and product share ownership of cost as a first-class metric. The mechanics: enforce **cost-allocation tags** (team/service/env/cost-center) so spend can be attributed; use multi-account structure so accounts map to teams/environments; give each team **visibility** into their own cost (dashboards, showback/chargeback); set **budgets and anomaly alerts**; and bake cost into engineering decisions (architecture reviews, CI cost estimates with Infracost). The cultural shift — engineers see and own their bill — drives more savings than any single tool.

Interview tip: Frame FinOps as culture + accountability, not just 'turn on Cost Explorer'. The enablers: enforced tagging (via tag policies/SCP/Config — un-enforced tags are useless), per-team showback/chargeback so cost is visible to the people who create it, budgets + anomaly detection, and account-per-team isolation. The maturity signal: shifting cost **left** (Infracost in PRs, cost in architecture reviews) so it's a design input, not a quarterly surprise.

Official docs

- [AWS Cost Explorer](#)
- [AWS Well-Architected: Cost optimization pillar](#)

 Spoken answer · ~88 s

I reduce database cost by matching resources to real needs and removing waste, with measurement at each step. Right-size: use CloudWatch and Performance Insights to see actual CPU, memory, and connection usage, and move over-provisioned instances to smaller classes; consider Graviton-based instance types for better price-performance. Choose storage wisely: gp3 instead of gp2 or over-provisioned io1, and provision IOPS only if needed; review allocated storage and delete unused snapshots. Use purchasing options: reserved instances for steady production databases, which give a large discount. For variable or spiky workloads, consider Aurora Serverless v2, which scales capacity with demand, and stop or schedule non-production databases outside working hours. Consolidate many small, lightly used databases onto fewer instances where isolation allows. Optimise the workload so it needs less hardware: fix slow queries, add the right indexes, add caching with ElastiCache to reduce read load, and use read replicas only where needed since each is a full instance cost. Control backup costs with sensible retention, and move old data to cheaper storage or archive it, for example to S3 via export. For Multi-AZ, keep it in production but not in dev. Review Aurora I/O costs and consider the I/O-optimised configuration if I/O is a large share of cost. Validate every change with performance testing.

Likely follow-ups

- How can caching reduce database cost?
- When does Aurora Serverless v2 save money?

```
# Right-size from real metrics (CPU, connections, IOPS, memory):
```

```
aws compute-optimizer get-rds-database-recommendations
```

```
# Commit to baseline with Reserved Instances (DBs are steady -> good RI fit):
```

```
aws rds purchase-reserved-db-instances-offering --reserved-db-instances-offering-id ...
```

```
# Offload reads to replicas (smaller primary); cache hot reads:
```

```
# ElastiCache in front of RDS; Aurora Serverless v2 for variable load
```

```
# gp3 storage instead of provisioned IOPS where possible; stop non-prod DBs
```

Databases are usually steady-state, so they're strong **Reserved Instance** candidates once right-sized. Reduce the instance you need by **right-sizing** from real utilization (Compute Optimizer), offloading reads to **read replicas** and a **cache** (ElastiCache) so the primary can be smaller, and choosing **gp3** storage instead of pricey provisioned IOPS unless you genuinely need it. For variable/intermittent load, **Aurora Serverless v2** scales capacity to demand (and scales non-prod down). Stop or downsize non-prod databases off-hours, tier old data to cheaper storage or archive, and watch backup/snapshot and cross-AZ replication costs.

Interview tip: The ordered play: right-size (real metrics) → reduce load (replicas + cache so the primary shrinks) → commit (RIs on the steady baseline) → storage (gp3 vs provisioned IOPS) → non-prod scheduling/serverless. The insight that impresses: a cache/read-replica lets you run a *smaller* primary, compounding savings, and Aurora Serverless v2 fits spiky/dev workloads. Don't commit RIs before right-sizing.

Official docs

- [Amazon RDS User Guide](#)

Q103 How do Graviton (arm64) and serverless change the cost equation? When do they pay off?

Cost Opt

Spoken answer · ~85 s

Graviton instances use AWS-designed arm64 processors, offering typically twenty percent lower price and up to forty percent better price-performance than comparable x86 instances for many workloads, plus better energy efficiency. They pay off for workloads that are portable to arm64: most interpreted and managed-runtime languages, containers, web services, databases like RDS and ElastiCache on Graviton, and open-source stacks. Effort lies in building multi-architecture images, checking native dependencies, and testing performance. Wins are largest for compute-heavy, scale-out workloads; less so for software with x86-specific binaries or licensing. Serverless changes the cost equation by converting fixed capacity into pay-per-use. Lambda, Fargate, DynamoDB on-demand, and Aurora Serverless cost nothing when idle and scale with demand, which saves money for spiky, low-utilisation, or unpredictable workloads and removes the operations cost. But at steady high utilisation, serverless often costs more than well-utilised, committed instances, so break-even analysis matters. Lambda can also be cheaper on arm64. The total cost includes engineering time, so serverless can win on operations even when unit prices are higher. Decide by modelling: estimate usage patterns, compute costs under each option, include operations and engineering effort, and run proofs of concept. Re-evaluate as traffic changes, and combine, for example Graviton containers for baseline plus Lambda for spikes.

Likely follow-ups

- What work is involved in moving to Graviton?
- How do you compute the break-even point between Lambda and containers?

```
# Graviton: ~20-40% better price/performance, broadly supported:
```

```
instance types: t4g, m7g, c7g, r7g; RDS/ElastiCache/Lambda all support arm64
```

```
# Build multi-arch so the same image runs on x86 and arm64:
```

```
docker buildx build --platform linux/amd64,linux/arm64 --push -t REP0:v1 .
```

```
# Serverless: pay-per-use beats always-on for spiky/low-duty workloads:
```

Lambda/Fargate: no idle cost; but steady high load -> reserved EC2 cheaper

Graviton (AWS arm64 CPUs) typically delivers 20-40% better price/performance for general workloads (web servers, microservices, caches, many databases) and is supported across EC2, Lambda, Fargate, RDS, and ElastiCache — often a near-free win once you build multi-arch images and validate your dependencies on arm64. **Serverless** (Lambda/Fargate) eliminates idle cost: it's cheapest for spiky, event-driven, or low-duty-cycle workloads where you'd otherwise pay for idle servers. But for **steady, high-utilization** workloads, a committed/reserved EC2 or container fleet is cheaper than per-invocation pricing — run the breakeven math.

Interview tip: Two distinct levers. Graviton is the low-risk, high-ROI move (build multi-arch, test arm64 compatibility, switch — 20-40% off across many services). Serverless is workload-shape-dependent: a win for spiky/intermittent (no idle pay), a loss for steady high throughput (do the cost-per-hour breakeven vs reserved compute). The mature answer rejects 'serverless is always cheaper' and frames both as utilization-driven decisions.

Official docs

- [AWS Lambda Developer Guide](#)

Q104 How do you forecast spend, set budgets, and catch cost anomalies before they blow up?

Cost Opt

Spoken answer · ~84 s

Cost control requires forecasting, budgets, and anomaly detection working together. Forecasting: use Cost Explorer's forecast for a baseline, then improve it with business drivers, building a model from unit costs, such as cost per customer, multiplied by expected growth, and adjust for planned launches, migrations, and commitments. Budgets: set AWS Budgets by account, team, service, or tag, with alert thresholds at percentages of both actual and forecasted spend, such as eighty and one hundred percent, and notify owners through SNS or chat; budget actions can apply restrictive policies in sandbox accounts. Anomaly detection: AWS Cost Anomaly Detection uses machine learning to identify unusual spend patterns by service, account, or tag, and alerts with root cause hints; tune the monitors and thresholds to reduce noise. Add leading-indicator controls: service quotas to limit runaway scaling, SCPs to block expensive instance types in sandboxes, and tagging policies to make attribution possible. Operationally, review spend weekly with team owners, daily dashboards for large accounts, and use the Cost and Usage Report in Athena for investigations. Define runbooks for responding to an alert: identify the service, the account, and recent changes via CloudTrail; stop the bleeding; then fix the cause. Review forecast accuracy over time and refine.

Likely follow-ups

- How does anomaly detection differ from a fixed budget alert?
- Which preventive controls stop runaway spend?

Budgets with threshold alerts (and optional actions):

```
aws budgets create-budget --budget file://budget.json # alert at 80%/100%/forecasted
```

```
# Cost Anomaly Detection – ML baseline, alerts on unusual spikes:
```

```
aws ce create-anomaly-monitor --anomaly-monitor file://monitor.json
```

```
aws ce create-anomaly-subscription ... # email/SNS when anomalies exceed $threshold
```

```
# Forecast from trend:
```

```
aws ce get-cost-forecast --metric UNBLENDED_COST --granularity MONTHLY --time-period ...
```

Proactive cost control combines forecasting, guardrails, and detection. **AWS Budgets** set spend (or usage/RI-coverage) thresholds per account/team/tag with alerts at actual *and forecasted* percentages — and can trigger actions (e.g. apply a restrictive policy). **Cost Anomaly Detection** builds an ML baseline of normal spend per service/account and alerts on unusual spikes (a misconfigured job, a leaked key spinning up instances, a runaway data-transfer) before the monthly bill reveals it. **Cost forecasts** project the trend so you see overruns coming. Combine with per-team budgets and tag-based attribution so alerts reach the team that owns the spend.

Interview tip: The key distinction: Budgets = **threshold** guardrails you define (including forecasted-to-exceed alerts and budget actions), Anomaly Detection = **ML** catching the unexpected (a spike you didn't set a threshold for — like a compromised key launching crypto-mining or a runaway pipeline). Routing alerts to the owning team via tags closes the loop. 'Catch it in days, not at month-end' — and budget actions to auto-contain runaway spend — is the senior framing.

Official docs

- [AWS Cost Explorer](#)

**Q105 Explain the AWS native CI/CD stack (CodePipeline, CodeBuild, CodeDeploy).
When use it vs GitHub Actions?**

CI/CD

 Spoken answer · ~84 s

The AWS-native stack has CodePipeline orchestrating stages, CodeBuild running builds and tests in managed containers defined by buildspec files, and CodeDeploy automating deployments to EC2, on-premises servers, Lambda, and ECS, with in-place or blue/green strategies, traffic shifting, lifecycle hooks, and automatic rollback on alarms. CodeArtifact and ECR store packages and images. Strengths: tight integration with IAM, no stored credentials, everything within your AWS account and compliance boundary, deploy actions across accounts and regions, and pay-per-use. Weaknesses: slower evolution, a less polished developer experience, clumsy pipeline authoring compared with modern tools, a smaller ecosystem, and weaker pull-request-centred workflows. GitHub Actions offers a large marketplace of reusable actions, concise YAML workflows living beside the code, first-class pull-request checks, matrix builds, and a strong community. It authenticates to AWS using OIDC federation, with short-lived credentials and no stored keys. Choose the native stack when the organisation mandates everything inside AWS, uses CodeCommit or strict compliance and data-residency requirements, or needs CodeDeploy's deployment features. Choose GitHub Actions when code is on GitHub and developer experience matters. A hybrid is common: Actions for build and test, then deploying via CodeDeploy or CodePipeline. Either way, keep pipelines as code, use least privilege, and add approval gates for production.

Likely follow-ups

- How does GitHub Actions authenticate to AWS securely?
- What deployment features does CodeDeploy add?

```
# CodePipeline orchestrates stages; CodeBuild builds; CodeDeploy deploys:
```

```
Source (CodeCommit/GitHub) -> Build (CodeBuild) -> Test -> Deploy (CodeDeploy/ECS/CFN)
```

```
# CodeBuild spec:
```

```
buildspec.yml: phases: { build: { commands: [ "docker build -t $REPO:$TAG ." ] } }
```

```
# CodeDeploy blue/green for ECS with automatic rollback on alarm:
```

```
appspec: { Resources: [...], Hooks: { BeforeAllowTraffic: lambda-validate } }
```

The native stack: **CodePipeline** orchestrates the stages, **CodeBuild** runs builds/tests in managed containers (buildspec.yml), and **CodeDeploy** handles deployment strategies (in-place, blue/green, canary) with hooks and automatic rollback on CloudWatch alarms. It's fully IAM-integrated, runs inside your account/VPC, and needs no third-party credentials. **GitHub Actions** (or GitLab CI) is often preferred for developer experience,

marketplace ecosystem, and keeping CI next to the code — authenticating to AWS via OIDC (no stored keys). Many teams use Actions for build/test and CodeDeploy for the AWS-side deployment, or go fully GitHub-native.

Interview tip: Know what each piece does and give a balanced 'when': native stack shines for tight IAM/VPC integration, compliance (everything in-account), and AWS-managed deploys (CodeDeploy blue/green + alarm rollback); GitHub Actions wins on DX, ecosystem, and being where the code lives (use OIDC to auth, no static keys). The pragmatic answer is often hybrid. Mentioning CodeDeploy's automatic rollback on CloudWatch alarms shows deployment-safety awareness.

Official docs

- [AWS CodePipeline](#)
- [AWS CodeBuild](#)
- [AWS CodeDeploy](#)

Q106 What is Infrastructure as Code on AWS? Compare CloudFormation, CDK, and Terraform.

CI/CD

Spoken answer · ~87 s

Infrastructure as Code defines infrastructure in versioned files that tooling applies, giving repeatability, review, and auditability. CloudFormation is AWS's native service: declarative JSON or YAML templates deployed as stacks, with change sets for previewing, automatic rollback on failure, drift detection, and no state file to manage because AWS holds the state; it supports new AWS features quickly but is verbose and AWS-only. The CDK lets you write infrastructure in general-purpose languages like TypeScript or Python, synthesising CloudFormation; you get loops, abstractions, and reusable constructs with high-level defaults, which speeds up development and fits developer teams, but still deploys through CloudFormation with its limits. Terraform, from HashiCorp, uses HCL, supports many providers beyond AWS, such as Kubernetes, Datadog, and GitHub, has a rich module ecosystem and a plan and apply workflow, but you manage state yourself, in a remote backend with locking; OpenTofu is the open-source fork. Choose CloudFormation or CDK for AWS-only organisations that want native integration and no state management; CDK for developers who prefer real languages; Terraform for multi-cloud or multi-provider needs and where the team already has the skills. Whichever you choose: modularise, use remote state or stacks per environment, review plans in pull requests, apply least-privilege deployment roles, detect drift, and avoid manual changes.

Likely follow-ups

- Why does CDK still depend on CloudFormation?
- What is the state management difference between Terraform and CloudFormation?

```
# CloudFormation – AWS-native declarative YAML/JSON, managed state (stacks):
```

```
aws cloudformation deploy --template-file tpl.yaml --stack-name prod --capabilities
CAPABILITY_IAM
```

```
# CDK – write infra in TypeScript/Python; synthesizes to CloudFormation:
```

```
cdk deploy # real programming language, loops/abstractions, AWS-native state
```

```
# Terraform – multi-cloud HCL, external state, huge provider ecosystem:
```

```
terraform plan; terraform apply # state in S3 + DynamoDB lock
```

CloudFormation is AWS-native declarative IaC with state managed by AWS (stacks, change sets, drift detection, automatic rollback) — no state file to babysit, but AWS-only and verbose. **CDK** lets you author in a real language (TypeScript/Python) with loops and reusable constructs, then synthesizes to CloudFormation — great developer ergonomics, still AWS-native state. **Terraform** uses HCL, manages its own **state** (typically S3 + DynamoDB lock), and supports many providers — the choice for multi-cloud or when you want one tool across AWS/GCP/Azure/SaaS. All enable versioned, reviewable, repeatable infra.

Interview tip: Decide by ecosystem + state model: AWS-only and want managed state/no state file → CloudFormation (or CDK for nicer authoring); multi-cloud or richest provider ecosystem and comfortable owning remote state → Terraform. The key technical contrast is **state**: CFN manages it for you; Terraform makes you manage it (S3 backend + DynamoDB locking — and protect/encrypt it). CDK = imperative authoring, declarative CFN underneath. Whatever the tool, emphasize plan/review/version-control and no manual console changes.



Official docs

- [AWS CloudFormation](#)

Q107 How do you manage Terraform state safely for a team (remote state, locking, drift)?

CI/CD

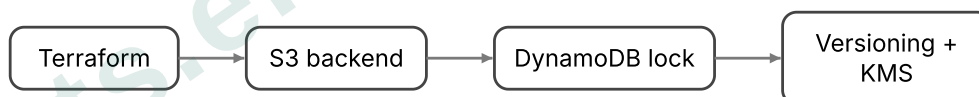
🗣 Spoken answer · ~86 s

Terraform state maps your configuration to real resources, and mishandling it causes outages, so treat it as critical. Store it remotely, in an S3 bucket with versioning, encryption with KMS, restricted access, and public access blocked, never in git, because it can contain secrets. Enable locking so concurrent applies can't corrupt it: the S3 backend now supports native lockfile locking, previously DynamoDB was used. Split state into small units by environment and component, such as network, data, and application, to limit the blast radius, speed plans, and reduce contention; share outputs through remote state data sources or well-defined interfaces. Control access: only CI should apply to production, with a dedicated role, and humans get read-only. Run plan in pull requests and require review before apply, using tools like Atlantis or pipelines with manual approval. Drift, when real infrastructure differs from state, arises from manual changes; detect it with scheduled plans or terraform plan in refresh-only mode, alert on differences, and either import the change or revert it; discourage console changes with SCPs and least-privilege access. Protect critical resources with lifecycle prevent_destroy. Back up state through bucket versioning so you can recover a prior version, and document break-glass procedures for state surgery using terraform state commands, carefully.

Likely follow-ups

- Why split state into multiple units?
- How do you detect and handle drift?

🚧 Terraform state on AWS



Remote backend: S3 for state + DynamoDB for locking (prevents concurrent apply):

```
terraform { backend "s3" { bucket="tf-state" key="prod/terraform.tfstate"

  region="us-east-1" dynamodb_table="tf-locks" encrypt=true } }
```

Detect drift (real infra vs state):

```
terraform plan -detailed-exitcode # exit 2 = drift detected
```

Isolate blast radius: separate state per environment/component (workspaces or dirs)

Team-safe Terraform centers on the **state file**. Store it **remotely** (S3, encrypted, versioned) so everyone shares one source of truth, and use a **lock** (DynamoDB table) so two applies can't run simultaneously and corrupt state. **Separate state** per environment and per component (small blast radius — a network change

can't accidentally affect the database stack), via directories or workspaces. Run `plan` in CI on PRs for review and apply only after merge (gated, with OIDC auth). Detect **drift** with scheduled `plan -detailed-exitcode` and never edit resources in the console (it desyncs state).

Interview tip: The non-negotiables: remote encrypted/versioned state (S3) + locking (DynamoDB) — explain *why* locking matters (concurrent applies corrupt state). Then blast-radius isolation (state per env/component), PR-gated plan/apply in CI with OIDC, and drift detection. The cardinal rule to state out loud: **no manual console changes** — they cause drift and the next apply may revert or conflict. Mention `terraform import` for adopting existing resources and protecting state secrets.

Official docs

- [Amazon DynamoDB Developer Guide](#)
- [Amazon S3 User Guide](#)

Q108 What is GitOps, and how do you implement it on EKS with ArgoCD or Flux?

CI/CD

Spoken answer · ~86 s

GitOps uses Git as the single source of truth for the desired state of infrastructure and applications, with an automated agent in the cluster continuously reconciling the live state to match Git. Changes happen through pull requests, which gives review, audit history, easy rollback by reverting, and consistent environments. On EKS, ArgoCD and Flux are the main tools. Setup: store Kubernetes manifests, Helm charts, or Kustomize overlays in a repository, often separate from application source code; install ArgoCD or Flux in the cluster; configure applications that point to repository paths and target namespaces or clusters; choose sync policies, automatic with pruning and self-healing, or manual approval for production. CI builds and pushes the image, then updates the image tag in the config repository, via a pull request or an image updater, and the agent deploys it. Benefits: pull-based deployments mean the cluster credentials stay inside the cluster, drift is automatically corrected, and the state is auditable. Practices: structure repositories per environment, use app-of-apps or ApplicationSets to manage many clusters, handle secrets securely with External Secrets, Sealed Secrets, or SOPS rather than plain text, add progressive delivery with Argo Rollouts, apply RBAC and SSO, monitor sync status and drift, and protect the main branch with required reviews.

Likely follow-ups

- Why is pull-based deployment more secure?
- How do you manage secrets in a GitOps workflow?

```
# Git is the single source of truth; a controller reconciles cluster -> Git:
```

```
# ArgoCD watches a repo and syncs manifests into the cluster:
```

```
argocd app create api --repo https://git/infra --path k8s/prod \\\
```

```
--dest-server https://kubernetes.default.svc --sync-policy automated --self-heal
```

Deploy = merge a PR that changes the image tag; ArgoCD applies it.

Drift: if someone kubectl-edits live, self-heal reverts to Git.

GitOps makes Git the single source of truth for cluster state: you describe the desired state declaratively in a repo, and an in-cluster controller (**ArgoCD** or **Flux**) continuously **reconciles** the live cluster to match Git. Deployments become Git operations (merge a PR bumping an image tag), giving you review, audit, and one-command rollback (`git revert`). The controller **pulls** from Git (no CI system needs cluster credentials), and **self-heal** reverts any manual kubectl drift back to the declared state. It separates CI (build/test/push image, update manifest) from CD (the controller syncs).

Interview tip: The defining ideas: Git as source of truth + continuous reconciliation + pull-based (cluster pulls from Git, so no external CI holds kube creds — a security win over push pipelines). Benefits: auditable history, easy rollback (revert the commit), and automatic drift correction (self-heal). Clarify the CI/CD split: CI builds and updates the manifest/tag in Git; the GitOps controller (Argo/Flux) does the actual deploy. Mention app-of-apps and progressive delivery (Argo Rollouts/Flagger).



Official docs

- [OpenGitOps principles](#)
- [Amazon EKS: User guide](#)
- [Argo CD documentation](#)

Q109 How do you build a secure software supply chain (signing, SBOM, scanning) in a pipeline?

CI/CD

🗣 Spoken answer · ~87 s

A secure software supply chain protects the integrity of what you build from source to production. Controls. Source: protected branches, required reviews, signed commits, and scanning for committed secrets. Dependencies: pin and lock versions, use private mirrors such as CodeArtifact, scan for known vulnerabilities and licence issues in CI with tools like Dependabot, Trivy, or Inspector, and block builds on critical findings. Build: use ephemeral, isolated, hermetic build runners, minimal permissions with OIDC short-lived credentials, and reproducible builds where possible. SBOM: generate a software bill of materials, in SPDX or CycloneDX formats, with tools like Syft, attach it to the artefact, and store it, so that when a new vulnerability is disclosed you can quickly find affected services. Scanning: scan images and artefacts for vulnerabilities, misconfigurations, and secrets before push and continuously in the registry. Signing and provenance: sign images and artefacts with Sigstore Cosign or AWS Signer, and generate provenance attestations, following SLSA levels, recording how and where something was built. Enforcement: verify signatures and policies at deployment using admission controllers like Kyverno or Gatekeeper, so only signed, scanned images from trusted pipelines run. Also secure the pipeline itself: restrict who can modify workflows, audit changes, and protect secrets. Monitor and respond with alerts on new CVEs.

Likely follow-ups

- What does an SBOM let you do when a new CVE is announced?
- How do you ensure only signed images run in the cluster?

```
# Scan dependencies + image in CI; fail on criticals:
```

```
trivy image --exit-code 1 --severity CRITICAL,HIGH $REPO:$TAG
```

```
# Generate an SBOM (software bill of materials):
```

```
syft $REPO:$TAG -o spdx-json > sbom.json
```

```
# Sign the image (keyless) so deploys can verify provenance:
```

```
cosign sign $REPO@$DIGEST; cosign verify $REPO@$DIGEST
```

```
# Admission control: only run signed images from trusted registries
```

A secure supply chain protects everything from source to running container. In the pipeline: **scan** dependencies and the built image (Trivy/Grype, ECR scanning, Inspector) and fail on critical CVEs; generate an **SBOM** (Syft) so you know exactly what shipped and can answer 'are we affected by CVE-X?' instantly. **Sign** images (cosign/Sigstore) and verify signatures at deploy time with an **admission controller**

(Kyverno/OPA) so only signed, provenance-verified images from trusted registries run. Pin dependencies and base-image digests, use OIDC (no static CI keys), and aim toward **SLSA** provenance attestations for build integrity.

Interview tip: Cover the stages: scan (deps + image, gate on severity), SBOM (Syft — the 'what's in it' artifact that makes the next CVE a 5-minute query), sign + verify (cosign + admission control so unsigned/untrusted images can't deploy), and provenance (SLSA attestations). Tie it to real incidents (Log4Shell, SolarWinds, xz) to show why it matters. The standout point is verifying signatures at admission, not just signing — signing without enforcement is theater.

Official docs

- [SPDX specification](#)
- [CycloneDX](#)

Q110 How do you implement progressive delivery (canary, feature flags, automated rollback)?

CI/CD

Spoken answer · ~85 s

Progressive delivery releases changes gradually, observing real behaviour, so problems are caught with limited impact. Techniques. Canary: send a small percentage of traffic to the new version, such as five percent, then increase in steps if metrics stay healthy; implement with weighted ALB target groups, CodeDeploy canary configurations, Route 53 weights, a service mesh, or Argo Rollouts and Flagger on Kubernetes. Blue/green: deploy the new version alongside the old and shift traffic when validated, keeping the old for instant rollback. Feature flags separate deployment from release: code ships turned off and is enabled for internal users, a percentage, or a segment, with a kill switch to disable instantly without redeploying; tools include LaunchDarkly, AppConfig, or open-source options. Automated analysis is the key: define success metrics, such as error rate, latency, and business metrics, compare the canary against the baseline, and automatically promote or roll back based on alarms or statistical analysis. Requirements: strong observability tagged by version, backward-compatible database changes, small changes, and ways to route consistently, such as sticky sessions for user-facing consistency. Set guardrails like maximum blast radius and duration of each stage. Clean up flags after rollout to avoid flag debt. Rehearse rollbacks. This approach reduces risk and allows frequent deployment.

Likely follow-ups

- How does automated canary analysis decide to roll back?
- Why is flag cleanup important?

```
# Canary: shift a small % of traffic, watch metrics, then proceed or abort:
```

```
# Argo Rollouts / Flagger automate the analysis + rollback:
```

```
steps: [ {setWeight: 10}, {pause: {duration: 5m}}, {analysis: success-rate}, {setWeight: 50} ]
```

```
# Analysis queries Prometheus/CloudWatch; auto-rollback if error rate/latency breach:
```

```
metric: error-rate < 1% AND p99-latency < 300ms else ROLLBACK
```

```
# Feature flags decouple deploy from release (LaunchDarkly / AppConfig):
```

```
aws appconfig start-deployment ... # flip features without redeploying
```

Progressive delivery reduces deployment risk by rolling out gradually and automating the safety checks. A **canary** shifts a small traffic slice to the new version, runs **automated analysis** against real metrics (error rate, latency, saturation from Prometheus/CloudWatch), and only proceeds if healthy — otherwise it **auto-rolls-back**. Tools like **Argo Rollouts** and **Flagger** orchestrate this on Kubernetes. **Feature flags** (LaunchDarkly, AWS AppConfig) decouple *deploy* from *release*: ship code dark, then enable it for a cohort/percentage at runtime (and kill-switch it instantly) without a redeploy.

Interview tip: The core insight: rollback should be **automatic and metric-driven**, not a human paging at 3am — canary + automated analysis (Argo Rollouts/Flagger querying SLO metrics) gives that. The second is decoupling deploy from release via **feature flags** (ship dark, enable progressively, instant kill-switch), which also enables A/B tests and limits blast radius. Define the metrics that gate promotion (error rate, p99 latency, saturation) — vague 'watch the dashboards' loses points.

Official docs

- [Kubernetes: Deployments](#)
- [AWS Builders' Library: Automating safe, hands-off deployments](#)
- [Google SRE Workbook: Canarying Releases](#)

🗣 Spoken answer · ~84 s

A multi-environment pipeline promotes the same artefact through increasingly production-like stages, so what you test is exactly what you ship. Build once: produce an immutable, versioned artefact, such as a container image tagged with the commit SHA, with tests and security scans, and promote it unchanged, with environment-specific configuration injected at deploy time, never rebuilt per environment. Stages: dev, staging, and production, often in separate AWS accounts for isolation, with the pipeline assuming a dedicated least-privilege deployment role in each account. Staging should mirror production in configuration and data shape. Promotion gates: automated tests at each stage, integration, end-to-end, performance, and security checks; for production, add approvals where required, change windows for sensitive systems, and verification that staging is healthy. Deploy to production with progressive strategies, canary or blue/green, with automatic rollback on alarms, and post-deployment smoke tests. Use infrastructure as code for environments so they are consistent, and manage configuration and secrets per environment through Parameter Store or Secrets Manager. Keep database migrations safe with backward-compatible steps. Provide fast feedback and keep the pipeline quick. Track DORA metrics, make rollbacks one click, keep audit logs of who deployed what, and use branch protections so only reviewed code reaches the main branch.

Likely follow-ups

- Why build once and promote the same artefact?
- Why use separate AWS accounts per environment?

```
# Promote the SAME artifact through environments (build once):
```

```
build -> push image (immutable digest) -> deploy dev -> test -> staging -> prod
```

```
# Config differs per env, NOT the artifact (12-factor):
```

```
# env-specific values via Parameter Store / Secrets Manager / Helm values
```

```
# Prod gated by approval + checks:
```

```
CodePipeline manual approval action / GitHub environment protection rules
```

The cardinal rule: **build once, promote the same immutable artifact** (by digest) through dev → staging → prod — never rebuild per environment (that defeats the purpose of testing). Environment **differences live in configuration** (Parameter Store/Secrets Manager, Helm values), not in the artifact, following 12-factor. Each environment mirrors prod as closely as practical. Production promotion is **gated**: required passing tests/scans, manual approval (CodePipeline approval action / GitHub environment protection), and ideally automated canary analysis. Account-per-environment isolates blast radius, with the pipeline assuming a cross-account deploy role via OIDC/STS.

Interview tip: The principle interviewers want stated explicitly: **build once, promote the same artifact** — rebuilding for prod means you tested something different. Config (not code) varies by env. Then production guardrails: gated approvals, required checks, canary, and account-per-env isolation with cross-account assume-role (OIDC, no static keys). Mention staging parity with prod and that secrets/config come from Parameter Store/Secrets Manager per environment.

 **Start here (official docs)**

- [AWS documentation](#)

Q112 **A deployment succeeded but the app is broken in production. How do you respond and prevent recurrence?**

CI/CD

 **Spoken answer** · ~86 s

First, restore service before diagnosing. If a deployment just happened and the symptoms fit, roll back immediately, or turn off the feature flag; a fast, rehearsed rollback is the best mitigation. Communicate: declare the incident, assign roles, and update stakeholders. Then confirm impact using error rates, latency, and business metrics, and compare against pre-deployment baselines. Why did the deployment report success but the app is broken? Typically because health checks were shallow and only verified that the process started, not that critical paths work; or configuration, secrets, or environment differences; a database migration or dependency incompatibility; a bug only triggered by production data or load; or an untested integration. After stabilisation, run a blameless post-incident review: build the timeline, find root causes and contributing factors, including why tests and monitoring missed it. Prevention: add automated post-deploy smoke and synthetic tests against critical user journeys, with deployment gates; use progressive delivery, canaries with automated rollback on error-rate alarms, so the blast radius is small; improve health checks to reflect real readiness; make database changes backward compatible; strengthen test coverage for the failure found; keep staging closer to production; and add monitoring and alerting that would have detected the issue sooner. Track action items to completion, and share lessons.

Likely follow-ups

- Why is rollback the first move after a bad deploy?
- How would a canary have limited the damage?

```
# 1. Roll back FIRST (restore service), diagnose after:
```

```
kubectl rollout undo deploy/api # or shift traffic to blue / revert the GitOps commit
```

```
# 2. Confirm recovery via the same metrics that detected the break:
```

```
# error rate, p99 latency, saturation back to baseline?
```

```
# 3. Diagnose on the failed version (logs, traces, config diff, what changed):
```

```
kubectl logs ...; compare config/env between good and bad releases
```

Treat it as an incident: **mitigate before root-causing**. Roll back immediately to the last known-good version (Kubernetes rollout undo, blue/green traffic switch back, or git revert in GitOps) — restoring service beats finding the cause in the moment. Verify recovery with the metrics that flagged the issue. Then diagnose the failed release: logs/traces, the config/env diff between good and bad, the migration or dependency that changed. **Prevent recurrence**: the deploy \"succeeded\" but broke prod means your health checks were too shallow — add real readiness/health validation, smoke tests post-deploy, canary with metric-based auto-rollback, and a blameless postmortem producing the specific guardrail (test/alarm/check) that would have caught it.

Interview tip: Lead with **roll back first, diagnose second** (mitigate over investigate). The deeper insight interviewers reward: a \"successful\" deploy that broke prod is a **validation gap** — your pipeline checked \"pods started\" not \"app works.\" Fix it with deeper health checks, post-deploy smoke tests, and canary + automated metric-based rollback so the next bad deploy is caught and reverted automatically. Close with a blameless postmortem and a concrete preventive action item.



Start here (official docs)

- [AWS documentation](#)

Q113 Users report intermittent 5xx errors from an app behind an ALB. How do you diagnose?

Troubleshooting

🗣 Spoken answer · ~86 s

Intermittent 5xx errors behind an ALB need me to separate errors generated by the load balancer from those returned by the targets. In CloudWatch, compare `HTTPCode_ELB_5XX_Count` with `HTTPCode_Target_5XX_Count`. If the targets return 500s, the application is failing: check application logs and traces for exceptions around those times, dependency failures, and resource exhaustion. If the ALB generates 502, the target closed the connection or sent a malformed response: common causes are keep-alive timeout mismatches, where the target closes idle connections sooner than the ALB's idle timeout; crashes or restarts; or oversized headers. Make the application's keep-alive timeout longer than the ALB's. A 503 means no healthy targets or capacity issues: check target group health, healthy host count, and deployments. A 504 means a target timed out: examine slow requests, the idle timeout setting, and backend latency. Use ALB access logs, which include error reasons, target status codes, and processing times per request, queried with Athena, to find patterns by target, path, or time. Check whether errors concentrate on a single instance, indicating a bad host, or zone, or correlate with deployments, scaling events, or health check flapping. Check security groups, connection limits, and CPU or memory pressure. Fix the cause, tune timeouts, and add alarms on 5xx rates.

Likely follow-ups

- How do you tell an ALB-generated 5xx from a target-generated one?
- Why do keep-alive timeout mismatches cause 502s?

```
# Distinguish WHO is erroring – ALB vs target – via ALB metrics + logs:
```

```
HTTPCode_ELB_5XX_Count -> the ALB itself (no healthy targets, timeouts)
```

```
HTTPCode_Target_5XX_Count -> the app returned 5xx
```

```
# Enable + query ALB access logs (per-request detail):
```

```
# fields: elb_status_code, target_status_code, target_processing_time, target:port
```

```
# Check target health + the usual intermittent causes:
```

```
aws elbv2 describe-target-health --target-group-arn tg-arn
```

First localize the 5xx: ALB CloudWatch metrics separate **ELB 5xx** (the load balancer — e.g. 502 from a target connection reset, 503 no healthy targets, 504 target timeout) from **Target 5xx** (the app actually returned an error). Enable **ALB access logs** and inspect `elb_status_code` vs `target_status_code` and `target_processing_time` per request. For **intermittent** errors, suspect: some targets failing health checks

(flapping in/out), idle-timeout mismatches (app keep-alive shorter than ALB's 60s → 502s), connection/thread pool exhaustion under load, a slow dependency causing 504s, or one bad instance in the pool. Correlate error timestamps with target health, deploys, and traffic spikes.

Interview tip: The first move that signals competence: split **ELB 5xx vs Target 5xx** (CloudWatch + access logs) — they point at completely different layers. Know the ALB code meanings: 502 = bad/reset target connection (classic keep-alive/idle-timeout mismatch — set the app's keep-alive higher than the ALB), 503 = no healthy targets, 504 = target too slow. 'Intermittent' = one flapping target or a saturated pool. Access logs per-request are the unlock.

Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Elastic Load Balancing](#)
- [Application Load Balancer](#)

Q114 An EC2 instance can't reach the internet (or a service can't be reached). How do you debug connectivity?

Troubleshooting

Spoken answer · ~89 s

I debug connectivity by walking the path hop by hop. Clarify source, destination, port, and protocol, and whether the failure is a timeout, a refusal, or a DNS error. Timeouts suggest dropped packets; refusals suggest the destination isn't listening. On the source: check DNS resolution, local firewall, and outbound security group rules. Check the route table of the source subnet for a route to the destination or default route: private subnets need a NAT gateway, or endpoints for AWS services; public subnets need an internet gateway route plus a public IP or elastic IP. Verify the NAT gateway is in a public subnet and healthy. Check the destination: inbound security group rules allowing the source, NACLs on both subnets in both directions with ephemeral ports, the service is listening on the right port and interface, and the host firewall. For cross-VPC or on-premises connectivity, check peering or Transit Gateway routes and attachments in both directions, overlapping CIDRs, and VPN or Direct Connect status. For private endpoints, check endpoint policies, security groups, and private DNS. Use VPC Reachability Analyzer to analyse the configured path and identify the blocking component, flow logs to see accepted and rejected packets, and tools like curl, telnet, traceroute, and tcpdump. Record the cause and fix in infrastructure as code.

Likely follow-ups

- What does Reachability Analyzer tell you that flow logs don't?
- How do you distinguish a security group block from a routing problem?

Walk the VPC path systematically:

1. Security Group -> allows the traffic (outbound + the inbound on target)?

2. NACL -> subnet allows BOTH directions (stateless, ephemeral ports)?

3. Route table -> 0.0.0.0/0 -> IGW (public) or NAT (private)? right target?

4. Public IP / NAT -> public subnet needs a public IP; private needs NAT GW

Confirm with VPC Reachability Analyzer + Flow Logs:

```
aws ec2 create-network-insights-path ...; aws ec2 start-network-insights-analysis ...
```

Walk the VPC layers in order. **Security Group** (stateful, per-ENI) — does it allow the traffic? **NACL** (stateless, per-subnet) — does it allow *both* directions, including the ephemeral return ports (the classic stateless trap)? **Route table** — is there a default route to an **Internet Gateway** (public subnet) or **NAT Gateway** (private subnet)? Does the instance have a public IP (public path) or is NAT correctly placed (private path)? Then DNS (can it resolve?) and the target's own SG. **VPC Reachability Analyzer** tests the path declaratively and tells you exactly which hop blocks it; **VPC Flow Logs** show ACCEPT/REJECT per flow to confirm where packets die.

Interview tip: Recite the ordered VPC checklist: SG → NACL (both directions + ephemeral ports!) → route table (IGW vs NAT) → public IP/NAT placement → DNS → target SG. The two highest-yield gotchas: the **stateless NACL ephemeral-return-port** trap, and a private subnet missing a NAT route (or NAT in the wrong subnet). Naming **Reachability Analyzer** (pinpoints the blocking hop without packet captures) and Flow Logs (REJECT entries) shows modern AWS troubleshooting, not just 'check the firewall.'

Official docs

- [Kubernetes: Debug Services](#)
- [Amazon EC2 User Guide](#)

 Spoken answer · ~85 s

For a slow RDS instance with high CPU or connections, I first find what is consuming it. Open Performance Insights: database load by wait events, top SQL, users, and hosts, which usually points to a handful of queries. Check CloudWatch: CPU, connections, IOPS and latency, freeable memory, and replica lag, and see when it began and what changed, such as a deployment, traffic growth, or a batch job. For high CPU: identify expensive queries, look at execution plans, and add missing indexes, rewrite inefficient queries, or stop runaway ones. For connection exhaustion: find which application is opening too many connections, fix connection pool settings and leaks, and introduce RDS Proxy or PgBouncer; long-idle or idle-in-transaction sessions may need timeouts. For locks and blocking: find blocking sessions and long transactions, and resolve or terminate them carefully. For IO: check IOPS limits, burst credits, and storage type; increase provisioned IOPS or throughput, or reduce IO through indexing and caching. To stabilise quickly: kill problem queries, scale up the instance class, add read replicas and route read traffic to them, enable caching, throttle the offending workload, or roll back a recent change. Then fix the root cause, add alarms for CPU, connections, and slow queries, and review capacity.

Likely follow-ups

- What does Performance Insights show first?
- How do you stabilise a database quickly while diagnosing?

```
# Performance Insights: find the top SQL + wait events (the fastest path):
```

```
# shows DB load by waits (CPU, IO, lock) and the worst queries
```

```
# Enhanced Monitoring + CloudWatch: CPU, FreeableMemory, Read/WriteIOPS, connections:
```

```
DatabaseConnections near max_connections -> connection exhaustion
```

```
# Inspect slow queries + plans:
```

```
EXPLAIN ANALYZE the-query; # missing index? full scan? bad plan?
```

```
# Stabilize: add read replica, RDS Proxy (pool conns), scale, add index
```

Use **Performance Insights** first — it visualizes DB load by wait event and surfaces the top SQL, immediately telling you whether you're CPU-bound, I/O-bound, or lock-bound and which queries are responsible. Pair with CloudWatch/Enhanced Monitoring (CPU, freeable memory, IOPS/burst-credit exhaustion, **DatabaseConnections** vs `max_connections`). Common causes: a missing index causing full scans (confirm with `EXPLAIN ANALYZE`), connection exhaustion (too many short-lived connections from many app instances/Lambdas), a runaway query, or undersized storage IOPS. Stabilize by killing the offending query, adding the index, offloading reads to a **read replica**, pooling connections with **RDS Proxy** (especially for Lambda), and scaling the instance/storage.

Interview tip: Go to **Performance Insights** first (top SQL + wait-event breakdown is the fastest root-cause). Then the AWS-specific stabilizers that impress: **RDS Proxy** for connection-pool exhaustion (the canonical Lambda-hammering-RDS fix), read replicas to offload reads, and watching for **storage burst-credit/IOPS** exhaustion causing sudden slowness. Always tie a slow query back to EXPLAIN ANALYZE (missing index/full scan) rather than just '\scale up.'

 Official docs

- [Amazon RDS User Guide](#)

Q116 How do you debug latency in a distributed AWS application using X-Ray and CloudWatch?

Troubleshooting

Spoken answer · ~87 s

I start with the trace to find where time is spent, then use metrics and logs to find why. In X-Ray or an OpenTelemetry backend, filter traces by latency or error, sort by duration, and open slow ones to see the waterfall of segments. The service map highlights which node has elevated latency or errors. Identify the slow span: your own code, a database call, a downstream service, an external API, or a queue wait. Compare slow traces against fast ones and look at annotations, such as customer tier or region, to see if latency correlates with specific inputs. Then pivot to CloudWatch for that component: CPU, memory, throttling, connection pool saturation, and garbage collection for services; Performance Insights for databases; throttling and concurrency for Lambda and DynamoDB; and network metrics. Use the trace ID to find the correlated log lines for detail. Check the time pattern: a spike at certain times suggests batch jobs, deployments, or autoscaling lag. Look at percentiles, not averages. Distributed issues include retries amplifying load, timeouts misconfigured across layers, fan-out calls where the slowest dominates, and cold starts. Gaps in traces suggest uninstrumented code, so add instrumentation. After identifying the cause, fix it, and verify with before-and-after traces. Add SLO-based alerts on latency.

Likely follow-ups

- How do you compare slow and fast traces?
- Why can retries make latency worse?

```
# X-Ray distributed tracing: see the full request path + per-segment time:
```

```
# service map shows which hop (ALB -> service -> DynamoDB -> downstream) is slow
```

```
# instrument with the X-Ray SDK / ADOT (OpenTelemetry) -> traces to X-Ray
```

```
# CloudWatch: correlate with metrics + use ServiceLens to unify traces/metrics/logs:
```

```
# p50/p99 latency per service, then drill into the slow trace
```

```
# CloudWatch Logs Insights to query structured logs by request id:
```

```
fields @timestamp, duration | filter requestId="abc" | sort duration desc
```

In distributed systems, aggregate latency hides *where* the time goes — use **distributed tracing**. **X-Ray** (or OpenTelemetry via ADOT) stitches a request across services into one trace with per-segment timing and a **service map**, so you can see which hop (the database call, a downstream API, a cold-started Lambda) owns the latency. Correlate with **CloudWatch** percentile metrics (watch **p99**, not averages — averages hide tail latency) and **ServiceLens** to jump from a latency spike to the offending traces and logs. Use **CloudWatch Logs Insights** to query structured logs by request/trace ID for the full story of a slow request.

Interview tip: The core principle: distributed latency needs **tracing**, not just metrics — X-Ray/OTel localizes the slow hop via the service map and per-segment timing. The metric-literacy point that lands: look at **p99/p99.9**, not averages (tail latency is what users feel and averages mask). Tie it together with ServiceLens (traces+metrics+logs) and structured logging with a propagated trace ID so you can follow one request end-to-end. Mention ADOT (OpenTelemetry) as the vendor-neutral modern choice.

Official docs

- [AWS X-Ray](#)
- [Amazon CloudWatch User Guide](#)

Spoken answer · ~85 s

For a bill spike, speed matters because costs accumulate hourly. Open Cost Explorer with daily granularity, and group by service first to see which line item jumped, then by account, region, usage type, and tag to narrow it down. Typical suspects: a runaway autoscaling group or Kubernetes cluster, forgotten large instances or GPU instances, NAT gateway or data transfer surges, a logging loop inflating CloudWatch Logs ingestion, S3 request or storage growth, a Lambda in an infinite loop, a DynamoDB on-demand traffic spike, or a compromised account running cryptominers in unusual regions. Check Cost Anomaly Detection for flagged anomalies and root cause hints. Use the Cost and Usage Report in Athena for resource-level detail. Correlate with CloudTrail for who created what and when, and with deployment history. Check all regions, since unexpected resources may sit in ones you don't use. If it looks like a security incident, treat it as one: rotate credentials and contain. Stop the bleeding: terminate or scale down the offending resources, fix the loop, or set limits. Then prevent recurrence: budgets with alerts, anomaly detection, service quotas, SCPs restricting regions and instance types, tagging for attribution, and a review of how it escaped notice. Document the cause and the savings.

Likely follow-ups

- Which signals suggest the spike is a security incident?
- How do you narrow a spike down using Cost Explorer?

```
# Cost Explorer: group by SERVICE, then by USAGE_TYPE + REGION to isolate:
```

```
aws ce get-cost-and-usage --granularity DAILY \\  
--group-by Type=DIMENSION,Key=SERVICE Type=DIMENSION,Key=USAGE_TYPE
```

```
# Find WHEN it started (daily granularity) and WHICH account/tag (group by tag):
```

```
# Check Cost Anomaly Detection findings for the flagged root cause
```

```
# Common spikes: data transfer, NAT, forgotten big instances, runaway Lambda/logs,  
leaked credentials launching instances (check CloudTrail + GuardDuty)
```

Move from coarse to fine in **Cost Explorer**: group by **service** to find which service jumped, switch to **daily** granularity to pin *when* it started, then group by **usage type**, **region**, and **tag/account** to isolate the exact driver. Check **Cost Anomaly Detection** — it often already identifies the root cause. Frequent culprits: a surge in **data transfer** (cross-AZ/egress) or **NAT Gateway** processing, a forgotten large/GPU instance or oversized RDS, runaway Lambda invocations or CloudWatch Logs ingestion, an un-lifecycled S3 bucket, or — critically — **leaked credentials** spinning up instances for crypto-mining (correlate with CloudTrail RunInstances and GuardDuty findings).

Interview tip: Show the drill-down method: service → daily (when) → usage type/region/tag (what + who). Lean on Cost Anomaly Detection for an instant lead. The two answers that stand out: the sneaky-but-common drivers (data transfer/NAT, logs ingestion, forgotten instances) and treating a sudden compute spike as a possible **security incident** — leaked keys mining crypto — checked via CloudTrail + GuardDuty. The latter shows you connect cost anomalies to security, which many candidates miss.

Official docs

- [AWS Cost Explorer](#)

Q118 An AWS service is returning throttling/RateExceeded errors. How do you handle and prevent it?

Troubleshooting

 Spoken answer · ~86 s

Throttling occurs when requests exceed an AWS service's rate limits or provisioned capacity, returning errors such as `ThrottlingException` or `RateExceeded`. To handle it in clients: implement retries with exponential backoff and jitter, which the AWS SDKs do by default, and tune the retry configuration; make operations idempotent so retries are safe; and respect the retry-after guidance. Avoid synchronized retry storms, which jitter prevents. To prevent it: first identify which limit is hit, using the error message, CloudWatch metrics, and service quotas; then reduce demand by batching calls, caching results, such as caching describe API responses or secrets, using pagination efficiently, and removing needless polling in favour of events; spread load over time and across keys or partitions; and use queues to smooth bursts, with controlled consumer concurrency. Request quota increases where limits are adjustable, with lead time, and monitor quota usage with Service Quotas and CloudWatch alarms at eighty percent. For DynamoDB, fix hot keys, adjust capacity or use on-demand, and add DAX; for Lambda, adjust concurrency settings; for API Gateway, configure usage plans and throttling to protect backends. Use different accounts or regions to distribute control-plane limits when appropriate. Add circuit breakers and graceful degradation so throttled dependencies don't cascade, and alert on throttle metrics.

Likely follow-ups

- Why add jitter to exponential backoff?
- How do you spot a quota limit before it bites?

```
# Symptoms: ThrottlingException, RateExceeded, ProvisionedThroughputExceeded, 429/503
```

```
# Handle: retry with EXPONENTIAL BACKOFF + JITTER (SDKs do this by default):
```

```
retry: base * 2^attempt + random_jitter    # never tight-loop retries (makes it worse)
```

```
# DynamoDB: switch to on-demand or raise provisioned capacity / fix hot partition:
```

```
# Lambda: raise reserved concurrency or the account concurrency limit
```

API-level: request a service quota increase (Service Quotas console)

Throttling means you exceeded a rate limit — either an **API request quota** or a **resource throughput limit**. Handle it gracefully with **exponential backoff + jitter** (the AWS SDKs do this automatically — don't tight-loop retries, which amplify the problem and can cause retry storms). Then address the root cause by service:

DynamoDB — switch to on-demand capacity, raise provisioned RCU/WCU, or fix a **hot partition** (poor partition-key design concentrating load); **Lambda** — reserved/account concurrency limits; **Kinesis** — shard count; general APIs — request a **Service Quotas** increase. Smooth bursty producers with a queue (SQS) so you process at a sustainable rate, and cache to cut call volume.

Interview tip: First answer is always **exponential backoff with jitter** (and that SDKs do it — naive fixed retries cause retry storms). Then root-cause by service: the standout is DynamoDB **hot partitions** (bad partition key → one partition throttles while capacity looks fine overall) and the on-demand-vs-provisioned trade-off. Mention buffering bursts with SQS, caching to reduce calls, and Service Quotas for hard API limits. Distinguishing API-rate throttling from resource-throughput throttling shows depth.

Official docs

- [Microsoft: Cloud design pattern – Throttling](#)

Q119 How do you investigate a security incident in AWS (compromised credentials or instance)?

Troubleshooting

Spoken answer · ~87 s

I follow an incident-response process: contain, preserve evidence, investigate, eradicate, recover, and learn. For compromised credentials: immediately deactivate the access key or revoke the role's sessions, using an IAM policy denying actions for sessions issued before a time, and rotate secrets. Use CloudTrail to determine what the credentials did: filter by access key ID or principal, look for unusual API calls, new IAM users, roles, or keys, policy changes, resource creation in odd regions, data access or exfiltration through S3 and snapshots sharing, disabled logging, and persistence mechanisms. GuardDuty and Security Hub findings supply context. For a compromised instance: isolate it, by replacing its security group with a quarantine group that blocks all traffic, rather than terminating it; keep it running if memory evidence is needed, snapshot the EBS volumes, and capture memory if you have tooling; revoke the instance role's sessions. Analyse from a separate forensic account or environment, not on the live host. Check how the attacker got in: vulnerable application, exposed credentials, weak SSH, or SSRF through IMDS. Eradicate: remove backdoors, rebuild from known-good images rather than cleaning, patch the entry point, rotate everything reachable. Notify stakeholders and meet legal obligations. Afterwards, strengthen controls, detections, and run a blameless review. Practise the process in advance.

Likely follow-ups

- Why quarantine an instance instead of terminating it?
- What do you look for in CloudTrail after a credential leak?

1. Detect/scope via the audit trail:

```
aws cloudtrail lookup-events --lookup-attributes AttributeKey=Username,AttributeValue=the-key
```

GuardDuty findings often flag it first (unusual API calls, crypto-mining, recon)

2. Contain – revoke the credential / isolate the instance:

```
aws iam delete-access-key / update-access-key --status Inactive # kill the key
```

attach a deny-all SG, isolate the instance (don't terminate -> preserve evidence)

3. Eradicate + recover: rotate ALL secrets, rebuild from clean AMI/IaC

Run the IR loop with AWS tooling. **Detect/scope:** **GuardDuty** usually surfaces it (anomalous API calls, mining, credential use from a new geo); **CloudTrail** is the authoritative audit log to trace exactly what the compromised principal did and when. **Contain** without destroying evidence: disable/delete the exposed access keys or detach the role policy, and for a compromised instance attach an isolation security group (deny all) and snapshot it for forensics — don't terminate (you'd lose volatile evidence). **Eradicate/recover:** rotate *all* potentially exposed secrets, revoke active STS sessions, rebuild the instance from a known-good AMI/IaC (don't 'clean' it), and remove any persistence the attacker added (new IAM users/keys, backdoor roles). Then close the entry vector (the leaked key, the SSRF, the public bucket) and post-mortem.

Interview tip: Map the IR phases to AWS services: GuardDuty (detect) + CloudTrail (scope — what did they do) → contain (disable keys, isolate instance, **snapshot before terminating** to preserve evidence, revoke STS sessions) → eradicate/recover (rotate all secrets, rebuild from clean AMI, hunt for attacker-created IAM persistence) → close the vector + postmortem. The senior details: revoke active sessions (not just keys), preserve forensic evidence, and rebuild rather than clean. Tie a leaked key back to the IMDSv2/SSRF lesson.

Official docs

- [NIST: Computer Security Incident Handling Guide \(SP 800-61\)](#)

Q120 How do you build effective observability on AWS (metrics, logs, traces, alarms, SLOs)?

Troubleshooting

 Spoken answer · ~84 s

Observability on AWS means being able to understand system behaviour from its outputs, built from metrics, logs, traces, and meaningful alerting, tied to user experience. **Metrics:** CloudWatch for AWS and custom metrics, Container Insights for containers, or Prometheus and Grafana managed services, with the golden signals for every service: latency, traffic, errors, and saturation. **Logs:** structured JSON logs with correlation IDs, shipped to CloudWatch Logs, S3, or OpenSearch, with retention and cost controls, and queried with Logs Insights or Athena. **Traces:** OpenTelemetry instrumentation exporting to X-Ray, so you can follow requests across services, with sampling. Connect the three through trace IDs so you can jump from an alert to the trace to the logs. Define SLIs and SLOs for key user journeys and alert on error-budget burn rates, rather than lots of low-level thresholds. Add synthetic canaries to test critical paths from outside, and real user monitoring. Build dashboards at the service level, plus an overview. Make alerts actionable: owners, runbooks, severity routing, and regular review for noise. Use cross-account observability for multi-account setups, and treat configuration as code. Watch the observability platform's own health and cost, and use incidents to find gaps in detection. Culture matters: teams own their instrumentation.

Likely follow-ups

- How do you link metrics, logs, and traces together?
- Why define alerts from SLOs?

Three pillars, unified:

Metrics -> CloudWatch (+ custom/EMF), Container Insights, Prometheus/AMP

Logs -> CloudWatch Logs + Logs Insights (structured JSON), centralized

Traces -> X-Ray / ADOT (OpenTelemetry) -> ServiceLens ties all three

Alarm on SYMPTOMS (SLOs), not just causes; use composite alarms to cut noise:

```
aws cloudwatch put-metric-alarm --metric-name p99Latency ... # alert on user-facing pain
```

Good observability combines the three pillars and ties them together. **Metrics** (CloudWatch, EMF for custom, Container Insights for EKS/ECS, or Prometheus/AMP) for the numbers; **structured logs** (JSON to CloudWatch Logs, queried with Logs Insights, centralized across accounts) for detail; **traces** (X-Ray or OpenTelemetry via ADOT) for request flow — unified in **ServiceLens** so you pivot from a metric spike to the traces and logs behind it. Alert on **symptoms/SLOs** (user-facing: error rate, p99 latency, availability — the golden signals) rather than every cause, use **composite alarms** to reduce noise, and define **SLOs/error budgets** so alerting and release decisions are driven by user impact, not vanity metrics. Propagate a trace/correlation ID through everything.

Interview tip: Name the three pillars and how to unify them (ServiceLens / OpenTelemetry-ADOT), then the maturity points: **alert on symptoms (SLOs/golden signals), not causes** — page on user-facing pain (p99 latency, error rate), not CPU at 80%; use composite alarms to fight alert fatigue; and define **SLOs + error budgets** to make alerting and 'should we ship?' objective. Structured logging + a propagated correlation ID is the practical glue. This frames observability as enabling fast MTTR, not just dashboards.

Official docs

- [Google SRE Book: Service Level Objectives](#)
- [OpenTelemetry: Signals \(traces, metrics, logs\)](#)

Q121 What are IAM permissions boundaries, and how are they different from SCPs and session policies?

IAM & Security

Spoken answer · ~62 s

A permissions boundary is a managed policy attached to a user or role that defines the ceiling of what that identity can do. It never grants anything by itself. The effective permission is the intersection of the identity policy and the boundary, so even an administrator policy cannot exceed it. Service control policies work the same way but at the account or organizational unit level, and session policies limit a single assumed-role session. I use boundaries for delegated administration: platform teams let developers create their own roles, but only if the new role carries a required boundary, enforced with a condition on `iam:PermissionsBoundary` in the developer's policy. That prevents privilege escalation through self-created admin roles. When debugging an unexpected access denied, I check all layers: SCP, boundary, session policy, identity policy and resource policy, because an explicit deny or a missing allow in any of them blocks the call.

Likely follow-ups

- How would a developer escalate privileges without a boundary?
- How do explicit denies interact with boundaries?

A permissions boundary is a managed policy that sets the maximum permissions an IAM user or role can ever have. Effective permissions are the intersection of the identity policy and the boundary. SCPs do the same at account level and session policies at session level.

```
aws iam put-role-permissions-boundary --role-name dev-role --permissions-boundary
arn:aws:iam::123456789012:policy/DevBoundary
```

```
aws iam get-role --role-name dev-role --query 'Role.PermissionsBoundary'
```

Interview tip: The classic use is delegated administration: let teams create roles, but only roles that carry the boundary.

Official docs

- [AWS IAM: Permissions boundaries](#)
- [AWS Organizations: Service control policies](#)

Q122 Why use IAM Identity Center (SSO) instead of IAM users for people, and how do permission sets work?

IAM & Security

Spoken answer · ~59 s

IAM users carry long-lived passwords and access keys that leak, are shared and are hard to offboard across many accounts. IAM Identity Center connects to the company identity provider, such as Okta or Entra ID, so people sign in once with their normal credentials and multi-factor authentication, then pick an account and role. They receive short-lived credentials that expire automatically. Administrators define permission sets, which are reusable bundles of policies, and assign a group to a permission set in specific accounts. Behind the scenes, Identity Center creates an IAM role in each target account. Offboarding becomes one action in the identity provider, and CloudTrail shows who really did what. For workloads I do not use users at all, I use roles. I keep one break-glass IAM user or role, with hardware MFA and alerts, in case the identity provider is down.

Likely follow-ups

- What is a break-glass account?
- How do you map IdP groups to permission sets?

Identity Center gives people short-lived credentials via your identity provider, one login across accounts, and central assignment. A permission set is a template that is turned into an IAM role in each assigned account.

```
aws configure sso
```

```
aws sso login --profile prod-admin
```

```
aws sts get-caller-identity --profile prod-admin
```

Interview tip: Say that long-lived IAM user access keys for humans are the first thing a security review flags.

Official docs

- [OAuth 2.0 / OIDC overview \(OpenID\)](#)

 Spoken answer · ~58 s

Access Analyzer helps in three ways. External access findings show S3 buckets, roles, KMS keys, queues and other resources that can be reached from outside the account or organization, so I can find accidental public or cross-account exposure. Unused access findings show roles, users and permissions that have not been used for a defined period, which lets me remove stale access. Policy generation reads CloudTrail logs for a role over a period, often ninety days, and drafts a policy that only includes the actions actually used. I review it, replace wildcards on resources where possible and attach it. Policy validation and custom checks in the pipeline catch problems like overly broad actions or missing conditions before they reach production. My workflow is to begin with a reasonably scoped policy, observe real usage, then tighten, and repeat on a schedule.

Likely follow-ups

- What are the limits of CloudTrail-based policy generation?
- How do you handle rarely used permissions such as disaster recovery?

Access Analyzer finds resources shared outside your zone of trust, flags unused access and permissions, and can generate a least-privilege policy from CloudTrail activity. Policy validation catches errors before deploy.

```
aws accessanalyzer list-findings --analyzer-arn <arn> --filter '{"status":{"eq":["ACTIVE"]}}'
```

```
aws accessanalyzer start-policy-generation --policy-generation-details principalArn=<role-arn>
```

```
aws accessanalyzer validate-policy --policy-type IDENTITY_POLICY --policy-document file://p.json
```

Interview tip: Start broad in dev, then tighten using generated policies from real usage, rather than guessing in advance.

 Official docs

- [AWS IAM: Security best practices](#)
- [AWS IAM: Least privilege \(grant least privilege\)](#)

Q124 **Gateway versus interface VPC endpoints: what is the difference and when do you use each?**

Networking

 Spoken answer · ~63 s

Both let private subnets reach AWS services without going over the internet or a NAT Gateway. A gateway endpoint is a target in the route table, available only for S3 and DynamoDB, and it has no hourly or data charge. An interface endpoint is built on PrivateLink: it creates elastic network interfaces with private IPs in your subnets, uses security groups, supports private DNS, and works for most other services such as ECR, STS, Secrets Manager, CloudWatch and SQS. It has an hourly cost per availability zone and a per-gigabyte charge. In practice I add the S3 gateway endpoint to every VPC, because it reduces NAT costs, and add interface endpoints for services that private workloads use heavily, such as ECR and logs for EKS nodes. I use endpoint policies to restrict which buckets or actions can be reached, and deploy interface endpoints in each zone the workloads run in.

Likely follow-ups

- How do endpoint policies restrict S3 access?
- Why does an EKS private cluster need several endpoints?

Gateway endpoints exist only for S3 and DynamoDB, are free, and work through route table entries. Interface endpoints use PrivateLink, create ENIs with private IPs in your subnets, cost per hour and per GB, and support many services.

```
aws ec2 create-vpc-endpoint --vpc-id vpc-1 --service-name com.amazonaws.eu-west-1.s3 --route-table-ids rtb-1
```

```
aws ec2 create-vpc-endpoint --vpc-endpoint-type Interface --service-name com.amazonaws.eu-west-1.ecr.api --subnet-ids subnet-1 --security-group-ids sg-1
```

Interview tip: Adding an S3 gateway endpoint is the quickest cut to NAT Gateway data charges.

 Official docs

- [AWS PrivateLink](#)

 Spoken answer · ~56 s

A Site-to-Site VPN creates IPsec tunnels over the public internet. It is fast to set up, cheap and encrypted, but each tunnel is limited in throughput and latency can vary. Direct Connect provides a dedicated private connection from your data center or colocation facility, with predictable latency, higher bandwidth and lower data transfer charges, but lead times are longer and traffic is not encrypted unless you add MACsec or run a VPN over it. For resilience I never rely on a single link. A common design is two Direct Connect connections at different locations as primary, with a VPN as a backup path, all advertised through BGP with proper preference settings, terminating on a Transit Gateway or Direct Connect Gateway. I test failover regularly and monitor BGP session state and tunnel status with alarms.

Likely follow-ups

- How do you prefer one path over another with BGP?
- How does Direct Connect Gateway work with multiple VPCs?

VPN is quick to set up, encrypted over the internet, and limited to about 1.25 Gbps per tunnel with variable latency. Direct Connect is a private dedicated link with consistent latency and higher bandwidth, but takes weeks and is not encrypted by default.

```
aws ec2 describe-vpn-connections --query 'VpnConnections[].VgwTelemetry'
```

```
aws directconnect describe-virtual-interfaces
```

Interview tip: The usual answer: Direct Connect as primary, VPN as backup, and MACsec or VPN over Direct Connect if encryption is required.

 Official docs

- [Cloudflare Learning: What is a VPN?](#)
- [AWS Direct Connect](#)
- [AWS Site-to-Site VPN](#)

Q126 How does Route 53 Resolver work for hybrid DNS between AWS and on-premises?

Networking

 Spoken answer · ~56 s

Each VPC has a built-in DNS resolver that handles public names and private hosted zones. For hybrid setups, you need a way to cross the boundary. An inbound resolver endpoint gives on-premises DNS servers IP addresses inside the VPC to forward queries to, so they can resolve private hosted zone records and internal names. An outbound endpoint lets the VPC resolver forward queries for specific domains, such as corp.example.com, to on-premises DNS servers, using forwarding rules. Rules can be shared across accounts and associated with many VPCs using Resource Access Manager, which suits a multi-account landing zone. For reliability I place endpoint IPs in at least two availability zones, secure them with security groups, and monitor query volume. I also keep a clear ownership model for which domains are resolved where, to avoid loops.

Likely follow-ups

- How do you share private hosted zones across VPCs?
- What causes DNS forwarding loops?

The VPC resolver at the base of the VPC CIDR plus 2 answers queries. Inbound endpoints let on-prem DNS query AWS private zones. Outbound endpoints with forwarding rules send chosen domains to on-prem DNS servers.

```
aws route53resolver create-resolver-endpoint --direction INBOUND --security-group-ids sg-1
--ip-addresses SubnetId=subnet-1 SubnetId=subnet-2
```

```
aws route53resolver create-resolver-rule --rule-type FORWARD --domain-name corp.example.com
--target-ips Ip=10.1.0.2
```

Interview tip: Put at least two IPs in different zones on each endpoint and share forwarding rules through AWS RAM.

Official docs

- [Amazon Route 53](#)

Q127 **What are EC2 placement groups, and when would you use cluster, spread or partition?**

Compute

 Spoken answer · ~58 s

Placement groups influence where instances run on the underlying hardware. A cluster group places instances close together in a single availability zone, giving the lowest latency and highest network throughput, which suits HPC, tightly coupled jobs and some distributed training. The risk is a shared failure domain and capacity errors, so I launch the whole group at once. A spread group places each instance on separate racks, limited to seven running instances per zone, which is right for a small number of critical instances that must not fail together, such as primary and standby. A partition group divides instances into logical partitions on separate racks, which fits large, rack-aware systems like Kafka, Cassandra or HDFS, where the application knows partition boundaries. I choose based on whether latency or fault isolation matters more, and check capacity before relying on them.

Likely follow-ups

- Why can a cluster placement group hit capacity errors?
- How does partition awareness help Kafka?

Cluster packs instances close together in one zone for low latency and high throughput. Spread puts each instance on distinct hardware, up to 7 per zone. Partition groups instances into racks that do not share hardware, for large distributed systems.

```
aws ec2 create-placement-group --group-name hpc --strategy cluster
```

```
aws ec2 run-instances --placement GroupName=hpc --instance-type c6in.32xlarge --count 4 --image-id <ami>
```

Interview tip: Cluster groups can fail with insufficient capacity, so launch all instances at once and use a single instance type.

 Start here (official docs)

- [AWS documentation](#)

Q128 **When does Lambda cost more than containers, and how do you decide between them?**

Compute

 Spoken answer · ~56 s

Lambda bills for requests and for memory multiplied by duration, with no cost when idle, so it is excellent for spiky, unpredictable or low-volume work, event-driven glue and workloads that scale to zero. As traffic becomes steady and high, paying per millisecond costs more than running containers at good utilisation. Other factors also matter: Lambda has time limits, cold starts, concurrency limits and a package size ceiling, while containers handle long-running processes, websockets and special runtimes more naturally. I compare with a simple model: requests per second, average duration and memory for Lambda, versus the number of tasks needed and their utilisation for Fargate or EC2. I also include operational cost, because Lambda removes much patching and scaling work. Many systems use both, with Lambda for events and containers for the steady core API.

Likely follow-ups

- How does memory size change Lambda cost and speed?
- What is provisioned concurrency?

Lambda charges per request and per GB-second, so it is cheap for spiky or idle workloads and expensive for steady high-volume traffic. Containers on Fargate or EC2 are cheaper at constant utilisation, but you manage scaling and idle capacity.

```
aws lambda get-function-configuration --function-name api --query '[MemorySize,Timeout]'
```

```
aws compute-optimizer get-lambda-function-recommendations
```

Interview tip: Do a break-even calculation with real request rate, duration and memory rather than choosing by fashion.

 Official docs

- [AWS Lambda Developer Guide](#)

Q129 How do Auto Scaling warm pools and instance refresh help with deployments and scale-out speed?

Compute

 Spoken answer · ~59 s

When an application takes minutes to boot, install packages or warm caches, scaling out in response to load is too slow. A warm pool keeps a set of pre-initialised instances on the side, either stopped, which costs only storage, or running, so that when the group needs capacity it takes from the pool and the instances reach service much sooner. Lifecycle hooks let you run initialisation before an instance joins and cleanup before it leaves. Instance refresh handles rollouts: when you update the launch template, AMI or instance type, it replaces instances in batches, keeping a minimum healthy percentage and waiting for a warmup period, with automatic rollback on alarms if configured. I combine this with baked AMIs to reduce boot time, health checks tied to the load balancer, and checkpoints so I can pause a rollout if error rates rise.

Likely follow-ups

- Stopped versus running warm pool: which is cheaper?
- How does instance refresh roll back?

A warm pool keeps pre-initialised stopped or running instances ready, so scale-out skips long boot and bootstrap time. Instance refresh replaces instances in an ASG gradually when the launch template or AMI changes, with a minimum healthy percentage.

```
aws autoscaling put-warm-pool --auto-scaling-group-name web --pool-state Stopped --min-size 2
```

```
aws autoscaling start-instance-refresh --auto-scaling-group-name web --preferences '{"MinHealthyPercentage":90,"InstanceWarmup":120}'
```

Interview tip: Use lifecycle hooks so warm instances finish their setup before they enter service.



Start here (official docs)

- [AWS documentation](#)

Q130 Managed node groups, self-managed nodes, Fargate and Karpenter on EKS: how do you choose?

Containers

 Spoken answer · ~55 s

Managed node groups let EKS create and update the Auto Scaling group, drain nodes during updates and keep AMIs patched, so they are a safe default. Self-managed nodes make sense when you need a custom AMI, special kernel settings or unusual bootstrap logic, but you own the lifecycle. Fargate removes nodes entirely and bills per pod resources, which is good for isolated or bursty workloads, but it lacks DaemonSets, privileged pods and GPUs, and has higher cost per unit. Karpenter watches pending pods and launches nodes of the right size and type, including Spot, then consolidates and removes underused nodes, giving better packing than fixed node groups. My usual design is a small managed node group for system components and Karpenter for application capacity, with Fargate only where per-pod isolation is valuable.

Likely follow-ups

- What can't run on Fargate?
- How does Karpenter consolidation work?

Managed node groups automate provisioning and rolling updates of EC2 workers. Self-managed gives full control over the AMI and bootstrap. Fargate runs each pod in its own microVM with no nodes to manage. Karpenter provisions right-sized nodes on demand.

```
aws eks create-nodegroup --cluster-name prod --nodegroup-name ng-1 --subnets subnet-1
subnet-2 --node-role <arn>
```

```
aws eks describe-nodegroup --cluster-name prod --nodegroup-name ng-1 --query
'nodegroup.updateConfig'
```

Interview tip: A common production answer is managed node groups or Karpenter for general workloads, plus Fargate for small isolated jobs.

Official docs

- [Amazon EKS: Managed node groups](#)
- [Karpenter documentation](#)
- [Amazon EKS: User guide](#)

Q131 EKS Pod Identity versus IRSA: how do they give pods AWS permissions, and which is better?

Containers

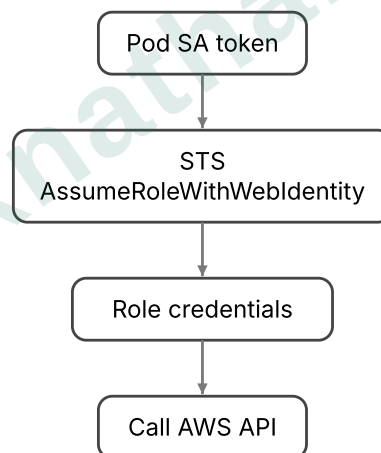
🗣 Spoken answer · ~62 s

Both remove the need for static keys and for broad node instance roles. With IRSA, the cluster has an OIDC identity provider. A Kubernetes service account is annotated with a role ARN, and the role's trust policy allows web identity tokens from that provider for a specific namespace and service account. The SDK exchanges the projected token for temporary credentials. It works well but needs an OIDC provider and a separately edited trust policy for every cluster. EKS Pod Identity moves this into an EKS feature: a DaemonSet agent serves credentials, and you create an association between cluster, namespace, service account and role. The role trusts the EKS pods service principal, and the same role can be reused across clusters. Pod Identity is simpler to operate and adds session tags. I choose it for new clusters, and keep IRSA where older SDK versions or Fargate constraints require it.

Likely follow-ups

- Why avoid node instance profile permissions?
- How are session tags used for ABAC?

🚀 Pod to AWS IAM



IRSA uses the cluster's OIDC provider and a service account annotation, with a trust policy per cluster. EKS Pod Identity uses an agent and an association between a service account and a role, with one reusable trust policy and no OIDC setup per cluster.

```
aws eks create-pod-identity-association --cluster-name prod --namespace app --service-account web --role-arn <arn>
```

```
kubectl annotate sa web eks.amazonaws.com/role-arn=<arn> # IRSA
```

Interview tip: Mention that both avoid node-wide instance profile permissions, which every pod on the node would share.

Official docs

- [Amazon EKS: IAM roles for service accounts \(IRSA\)](#)
- [Amazon EKS: Pod Identity](#)
- [Amazon EKS: User guide](#)

Q132 **How do ECS task definitions, services and task roles fit together, and what is the difference between the execution role and the task role?**

Containers

Spoken answer · ~61 s

A task definition is the blueprint: container images, ports, CPU and memory, environment, logging and IAM roles. A task is one running instance of it. A service ensures a desired count of tasks, replaces failed ones, integrates with a load balancer and performs rolling or blue-green deployments. There are two roles that people mix up. The task execution role belongs to the ECS agent, and grants permission to pull images from ECR, fetch secrets for injection and send logs to CloudWatch. The task role is assumed by your application code inside the container, so it carries permissions for S3, DynamoDB or SQS. Following least privilege means giving each service its own task role. If a container starts but the app gets access denied, I check the task role. If the task never starts or cannot pull its image, I check the execution role and network path.

Likely follow-ups

- Which role is needed to read a secret injected as env var?
- How does awsvpc mode affect security groups?

A task definition describes containers, CPU, memory and roles. A service keeps a desired number of tasks running behind a load balancer. The execution role lets ECS pull images and write logs; the task role is what your application code uses to call AWS APIs.

```
aws ecs register-task-definition --cli-input-json file://td.json
```

```
aws ecs update-service --cluster prod --service web --task-definition web:42 --force-new-deployment
```

```
aws ecs describe-services --cluster prod --services web --query 'services[0].events[:5]'
```

Interview tip: Mixing the two roles up is the most common ECS permissions bug in interviews and in production.

Official docs

- [Amazon ECS Developer Guide](#)

 Spoken answer · ~58 s

gp3 is the default choice. It gives a baseline of three thousand IOPS and one hundred twenty-five megabytes per second regardless of size, and lets you raise IOPS and throughput separately, at a lower price than gp2. io2 and io2 Block Express offer provisioned IOPS up to very high levels, sub-millisecond latency and higher durability, suited to critical databases. st1 and sc1 are magnetic volumes for big sequential workloads like logs or data warehouses and cannot be boot volumes. Snapshots are point-in-time backups stored in S3, and after the first full copy they are incremental, so only changed blocks are saved. A snapshot is crash-consistent, so for databases I quiesce or use application-aware tooling. Volumes can be modified online, so I migrate gp2 to gp3 without downtime, and automate backups with Data Lifecycle Manager or AWS Backup, plus restore tests.

Likely follow-ups

- What is the first-read penalty after restoring a snapshot?
- How do you make EBS backups application-consistent?

gp3 is general-purpose SSD with baseline 3,000 IOPS and 125 MB/s that you can raise independently. io2 provides provisioned, high-durability IOPS for databases. st1 is throughput-optimised HDD for large sequential workloads. Snapshots are incremental and stored in S3.

```
aws ec2 modify-volume --volume-id vol-1 --volume-type gp3 --iops 6000 --throughput 250
```

```
aws ec2 create-snapshot --volume-id vol-1 --description nightly
```

```
aws ec2 describe-volumes-modifications --volume-ids vol-1
```

Interview tip: Moving gp2 to gp3 online is a free cost and performance win, which interviewers like to hear.

 Official docs

- [Amazon EBS](#)

Q134 What are S3 presigned URLs, and how do you use them securely for uploads and downloads?

Storage

 Spoken answer · ~62 s

A presigned URL lets a user or app perform a specific S3 operation on a specific object for a limited time without having AWS credentials. The URL carries a signature created by an IAM principal, and S3 evaluates the request with that principal's permissions, so if the signer loses access, or if temporary credentials used to sign expire first, the URL stops working. I use them so browsers upload large files directly to S3 and skip my servers, and to serve private downloads. To use them securely I set short expiries, generate the key server-side so users cannot choose it, restrict content type and size using presigned POST conditions, keep the bucket private with Block Public Access, and log access. Because anyone holding the link can use it, I avoid putting sensitive links in places that get logged or shared, and consider CloudFront signed URLs for broader distribution.

Likely follow-ups

- Presigned URL versus CloudFront signed URL?
- What happens if the signing role's session expires first?

A presigned URL embeds temporary credentials and a signature so a client can GET or PUT one object without AWS credentials. It inherits the permissions of whoever signed it and expires after the time you set.

```
aws s3 presign s3://bucket/report.pdf --expires-in 300
```

```
python -c "import boto3; print(boto3.client('s3').generate_presigned_url('put_object', Params={'Bucket':'b','Key':'k'}, ExpiresIn=300))"
```

Interview tip: Keep expiry short, pin the key and content type, and remember the URL is a bearer token: anyone who has it can use it.

Official docs

- [Amazon S3 User Guide](#)

Q135 EFS performance and throughput modes: how do you pick and avoid surprises?

Storage

 Spoken answer · ~59 s

EFS provides shared, elastic NFS storage across many instances and zones. Performance mode should almost always be General Purpose, since Max I/O is a legacy option. Throughput mode matters more. Elastic automatically scales throughput with the workload and is the best default when demand is unpredictable. Provisioned lets you pay for a fixed throughput regardless of size, useful when you know the need. Bursting ties throughput to the amount of data stored, which can starve small file systems. EFS performs well for large sequential access and shared content, but small files and heavy metadata operations have higher latency than local disk. I mount through the target in the same zone, use TLS and IAM authorisation, use access points to enforce paths and identities for containers, apply lifecycle policies to move cold data to Infrequent Access, and avoid it for latency-sensitive databases.

Likely follow-ups

- What do EFS access points do?
- When would you pick FSx instead?

EFS is elastic NFS storage. Throughput modes are Elastic, Provisioned and Bursting; performance mode is General Purpose for almost all workloads. Metadata-heavy small-file patterns and cross-AZ mounts are the usual sources of slowness.

```
aws efs describe-file-systems --query 'FileSystems[].[FileSystemId,ThroughputMode,PerformanceMode]'
```

```
aws efs update-file-system --file-system-id fs-1 --throughput-mode elastic
```

```
mount -t efs -o tls,iam fs-1:/ /mnt/efs
```

Interview tip: Mount using the mount target in the same availability zone as the client to avoid cross-AZ latency and charges.

Official docs

- [Amazon EFS](#)

Q136 **RDS Multi-AZ versus read replicas: what does each give you, and do they replace each other?**

Databases

 Spoken answer · ~58 s

Multi-AZ is about availability. RDS maintains a standby in another zone with synchronous replication, and if the primary fails, or during maintenance, it promotes the standby and flips the DNS endpoint, usually within a minute or two. In the classic single-standby setup you cannot read from the standby, although the newer Multi-AZ cluster option provides two readable standbys. Read replicas are about scale and distance. They use asynchronous replication, so they can lag, and you point read traffic at their endpoints. They can also be placed in other regions and can be promoted for disaster recovery, but promotion is manual and loses unreplicated writes. So they do not replace each other. A typical production design uses Multi-AZ for the primary and one or more replicas for read-heavy reporting. My application must handle reconnects after failover and tolerate replica lag.

Likely follow-ups

- How does failover change the connection?
- What does Aurora do differently?

Multi-AZ keeps a synchronous standby for high availability and automatic failover, and you cannot read from the classic standby. Read replicas use asynchronous replication to scale reads and can be cross-region. They solve different problems.

```
aws rds modify-db-instance --db-instance-identifier prod --multi-az --apply-immediately
```

```
aws rds create-db-instance-read-replica --db-instance-identifier prod-ro1 --source-db-instance-identifier prod
```

```
aws rds reboot-db-instance --db-instance-identifier prod --force-failover
```

Interview tip: Add that replica lag means reads can be stale, so read-your-own-writes paths must use the primary.

 Official docs

- [Amazon RDS: Multi-AZ deployments](#)
- [Amazon RDS: Read replicas](#)
- [Amazon RDS User Guide](#)

 Spoken answer · ~57 s

DynamoDB Streams records item-level changes, inserts, updates and deletes, in order per item, and keeps them for twenty-four hours. You choose what each record contains: keys only, new image, old image, or both. A Lambda function is typically subscribed to react to changes, for example to update a search index, send notifications, maintain aggregates or replicate to another store. Processing must be idempotent because records may be delivered more than once. TTL lets you mark an attribute holding an epoch timestamp, and DynamoDB deletes expired items in the background at no write cost, which suits sessions, tokens and temporary data. Deletion can lag by up to a couple of days, so queries should filter expired items explicitly. TTL deletes show up in the stream with a service identity, so consumers can distinguish them from user deletions.

Likely follow-ups

- How do you make stream consumers idempotent?
- How is DynamoDB export to S3 different?

Streams capture an ordered 24-hour log of item changes, which Lambda or Kinesis consumers can process. TTL deletes expired items for free in the background, within a day or two of expiry, and the deletions also appear in the stream.

```
aws dynamodb update-time-to-live --table-name sessions --time-to-live-specification
Enabled=true,AttributeName=expiresAt
```

```
aws dynamodb update-table --table-name orders --stream-specification
StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Interview tip: TTL is not exact: expired items can stay readable, so filter on the timestamp in queries.

 Official docs

- [Amazon DynamoDB Developer Guide](#)

 Spoken answer · ~62 s

RDS Proxy sits between the application and an RDS or Aurora database. It maintains a pool of established connections and multiplexes many client connections onto fewer database connections, which protects the database from connection storms. That is common with Lambda, where each concurrent invocation could open its own connection and exhaust the max connections limit. The proxy also shortens failover impact, because it keeps client connections and reconnects to the new primary on the backend, and it supports IAM authentication and credentials stored in Secrets Manager, so apps do not embed passwords. There are costs and caveats: it adds a small latency hop and a per-vCPU charge, and features like session variables, temporary tables or prepared statements can cause session pinning, which reduces multiplexing. I use it for serverless or bursty workloads, and for a stable pool with PgBouncer or an application pool I may not need it.

Likely follow-ups

- What causes session pinning?
- RDS Proxy versus PgBouncer?

RDS Proxy pools and shares database connections, reduces failover time by holding connections, and can use IAM auth with Secrets Manager credentials. It is most useful with Lambda or apps that open many short connections.

```
aws rds create-db-proxy --db-proxy-name app-proxy --engine-family POSTGRESQL --auth
AuthScheme=SECRETS,SecretArn=<arn> --role-arn <arn> --vpc-subnet-ids subnet-1 subnet-2
```

```
aws rds describe-db-proxy-targets --db-proxy-name app-proxy
```

Interview tip: Watch for session pinning, because pinned connections cannot be shared and reduce the benefit.

Official docs

- [Amazon RDS Proxy](#)
- [Amazon RDS User Guide](#)

Q139 How do you use CloudWatch Logs Insights to investigate an incident quickly?

Monitoring

 Spoken answer · ~61 s

Logs Insights lets me query one or many log groups interactively. A typical incident flow is to pick the affected log groups and a tight time window around the start of the problem, then count errors per five-minute bin to find when it began. Next I group by dimensions such as path, status code, host or customer ID to see whether it is isolated. Then I look at percentile latency over time and sample the raw error messages. If logs are structured JSON, fields like level, requestId and duration are automatically discovered, which makes the queries short, otherwise I use parse with a pattern. I save common queries, add them to dashboards, and create metric filters for the patterns I want to alert on. Since I pay for data scanned, I scope queries carefully, and for long-term analytics I export logs to S3 and use Athena.

Likely follow-ups

- How do metric filters differ from Logs Insights?
- How do you trace one request across services?

Logs Insights runs a purpose-built query language over log groups. You filter, parse fields, aggregate and chart. Structured JSON logs make fields queryable immediately, so you can find error spikes, top offenders and slow requests in seconds.

```
fields @timestamp, @message | filter @message like /ERROR/ | stats count() by bin(5m)
```

```
filter status >= 500 | stats count() as c by path | sort c desc | limit 10
```

```
fields @timestamp, duration | stats pct(duration, 99) by bin(1m)
```

Interview tip: Query cost is per GB scanned, so narrow the time range and log groups first.

Official docs

- [CloudWatch Logs Insights](#)
- [Amazon CloudWatch User Guide](#)

Q140 **What does CloudWatch Container Insights give you for ECS and EKS, and what are its limits?**

Monitoring

 Spoken answer · ~55 s

Container Insights provides managed collection of container metrics without running your own monitoring stack. For EKS you install the CloudWatch observability add-on, which deploys the agent and Fluent Bit, and for ECS you enable it on the cluster. You get metrics for clusters, nodes, services, pods and tasks, such as CPU, memory, network, restarts and storage, performance log events with high-cardinality data, and curated dashboards plus support for application signals and trace links. It is quick to adopt and integrates with alarms, anomaly detection and X-Ray. The limits are cost, since every metric and log GB is billed, and flexibility, because PromQL, custom exporters and open-source ecosystem dashboards are not available. Many teams combine it for baseline infrastructure signals with Amazon Managed Prometheus and Grafana for application metrics and richer queries.

Likely follow-ups

- When would you choose managed Prometheus instead?
- How do you reduce Container Insights cost?

Container Insights collects cluster, node, pod and task level CPU, memory, network and storage metrics and performance logs, with prebuilt dashboards. It is easy to enable but can be expensive at scale and is less flexible than Prometheus.

```
aws eks create-addon --cluster-name prod --addon-name amazon-cloudwatch-observability
```

```
aws ecs update-cluster-settings --cluster prod --settings  
name=containerInsights,value=enhanced
```

Interview tip: Point out the cost lever: metrics and logs are charged per metric and per GB, so review which are enabled.

 Official docs

- [Amazon EKS: User guide](#)
- [Amazon CloudWatch User Guide](#)
- [Amazon ECS Developer Guide](#)

Q141 **What are CloudWatch composite alarms and anomaly detection, and how do they cut alert noise?**

Monitoring

 Spoken answer · ~57 s

A single metric threshold is often noisy. A composite alarm evaluates the state of other alarms using a boolean expression, so I can page only when, for example, the error rate is high and order volume is low, rather than whenever CPU briefly spikes. It can also suppress child notifications, so on-call receives one alert for the incident instead of twenty. Anomaly detection uses a machine learning model trained on history to create an expected band, adapting to hourly, daily and weekly patterns, so I can alarm when traffic or latency falls outside normal without choosing a static number. It is good for seasonal metrics, but I check the band is sensible and exclude past incidents from training. In general, I alert on symptoms customers feel, keep cause-level alarms for investigation, and review noisy alerts monthly.

Likely follow-ups

- Why suppress child alarm actions?
- When is a static threshold better?

A composite alarm combines other alarms with AND, OR and NOT rules so you page only when several symptoms agree. Anomaly detection learns the normal band for a metric, accounting for time-of-day patterns, and alarms outside that band.

```
aws cloudwatch put-composite-alarm --alarm-name checkout-down --alarm-rule "ALARM(high-5xx)
AND ALARM(low-orders)"
```

```
aws cloudwatch put-anomaly-detector --namespace AWS/ApplicationELB --metric-name
RequestCount --stat Sum
```

Interview tip: Alert on user-facing symptoms and use the cause alarms as suppressed inputs or dashboards.

 Official docs

- [Amazon CloudWatch: Alarms](#)
- [Google SRE Book: Practical Alerting](#)
- [GNU coreutils manual](#)

Q142 Compute Savings Plans versus EC2 Instance Savings Plans versus Reserved Instances: how do you choose and avoid over-commitment?

Cost Opt

 Spoken answer · ~59 s

All of these trade a one or three year commitment for lower prices. A Compute Savings Plan commits to spending a certain dollar amount per hour and automatically applies to EC2 regardless of family, size, region or operating system, and also to Fargate and Lambda, so it is the most flexible. An EC2 Instance Savings Plan gives a larger discount but is locked to a family in a region. Reserved Instances apply to specific instance attributes and can be sold or converted in some cases, and they are still used for RDS, ElastiCache and OpenSearch, which have their own reservations. I analyse the baseline using Cost Explorer's recommendations, commit only to the steady portion, stagger purchases through the year so expiry is smooth, and track coverage and utilisation. If I plan a migration to Graviton or containers, the flexible plan is safer.

Likely follow-ups

- What happens if you under-use a commitment?
- How do you stagger commitments?

Compute Savings Plans commit to a dollar-per-hour spend and apply across EC2, Fargate and Lambda in any region and family. EC2 Instance plans give a deeper discount for one family in one region. Reserved Instances are tied to specific instance attributes.

```
aws ce get-savings-plans-purchase-recommendation --savings-plans-type COMPUTE_SP --term-in-years ONE_YEAR --payment-option NO_UPFRONT --lookback-period-in-days THIRTY_DAYS
```

```
aws ce get-savings-plans-coverage --time-period Start=2026-09-01,End=2026-10-01
```

Interview tip: Commit to your stable baseline, around 60 to 70 percent, never to peak usage.

Official docs

- [AWS Savings Plans](#)
- [Azure Reservations](#)
- [Amazon EC2 User Guide](#)

Q143 How do you use Compute Optimizer and rightsizing recommendations without hurting performance?

Cost Opt

 Spoken answer · ~57 s

Compute Optimizer looks at utilisation history from CloudWatch and suggests a better fit: whether an instance is over-provisioned, under-provisioned or optimised, and which types would perform better or cost less. It also covers EBS volumes, Lambda memory settings, Auto Scaling groups and Fargate. To use it safely, I first make sure the right data is present, particularly memory through the CloudWatch agent and a long enough lookback, including month-end or seasonal peaks. Then I rank by savings and risk, test changes in a lower environment or on a canary instance, and apply them gradually using infrastructure as code. I confirm latency and error rates after each change and keep the previous size for easy rollback. Graviton options often show up as additional savings. I treat the tool as input to a decision, not an automatic instruction.

Likely follow-ups

- Why isn't CPU alone enough?
- How do you roll back a rightsizing change?

Compute Optimizer analyses CloudWatch utilisation, usually 14 days, and recommends smaller or different instance types, EBS volumes, Lambda memory and ECS on Fargate sizes. Validate against peak, memory metrics and business cycles before acting.

```
aws compute-optimizer get-ec2-instance-recommendations --query 'instanceRecommendations[].[instanceArn,finding,recommendationOptions[0].instanceType]'
```

```
aws compute-optimizer put-recommendation-preferences --resource-type Ec2Instance --enhanced-infrastructure-metrics Active
```

Interview tip: Memory is not collected by default, so install the CloudWatch agent or the recommendation misses the key signal.

Official docs

- [AWS Well-Architected: Cost optimization pillar](#)

 Spoken answer · ~55 s

A good strategy starts small and enforceable. I choose a handful of mandatory tags, such as team, owner, environment, application and cost center, with allowed values. In AWS Billing I activate them as cost allocation tags, because they do not appear in reports until activated and are not retroactive. Enforcement happens at several layers: tag policies in Organizations standardise keys and values, service control policies or IAM conditions can deny creation without required tags, and Terraform or CloudFormation modules apply default tags and pipeline checks fail untagged resources. Not everything can be tagged or attributed, so I use a separate account per team or environment as the strongest boundary, and allocate shared costs like networking proportionally. Then I build showback dashboards and review untagged spend weekly, so the tags remain accurate over time.

Likely follow-ups

- Why are accounts a stronger boundary than tags?
- How do you handle shared platform costs?

Define a small mandatory set such as owner, team, environment, cost-center and application. Activate them as cost allocation tags, enforce with tag policies, SCPs and IaC checks, and handle untaggable spend with account structure and shared-cost rules.

```
aws organizations create-policy --type TAG_POLICY --name std-tags --content  
file://tags.json
```

```
aws ce list-cost-allocation-tags --status Inactive
```

```
aws resourcegroupstaggingapi get-resources --tag-filters Key=team --query  
'ResourceTagMappingList[].ResourceARN'
```

Interview tip: Tags only apply from activation onward, so activate early. Tag policies check values but do not stop untagged resources by themselves.

 Official docs

- [AWS Cost Explorer](#)

Q145 How do you let GitHub Actions or another CI system deploy to AWS without static access keys?

CI/CD

 Spoken answer · ~58 s

Instead of storing an access key in CI secrets, I establish trust between AWS and the CI provider. In IAM I create an OpenID Connect identity provider for the provider's token issuer. Then I create a deployment role whose trust policy allows `sts:AssumeRoleWithWebIdentity` from that provider, with conditions on the audience and on the subject claim so only a particular repository, branch, tag or environment can assume it. During the pipeline run, the CI platform issues a short-lived signed token, the job exchanges it for temporary credentials that expire in about an hour, and no long-lived secret exists to leak or rotate. I give each role least-privilege permissions for its job, separate roles for plan and apply, and for production require a protected environment with approval. I also log role assumptions in CloudTrail and alert on unexpected subjects.

Likely follow-ups

- What does the sub claim contain?
- How would you separate plan and apply permissions?

OIDC from CI



Use OpenID Connect federation. Create an IAM OIDC identity provider for the CI issuer, then a role whose trust policy allows web identity from that provider with conditions on repository and branch. The job exchanges its token for temporary credentials.

```
aws iam create-open-id-connect-provider --url https://token.actions.githubusercontent.com -  
-client-id-list sts.amazonaws.com
```

```
aws sts assume-role-with-web-identity --role-arn <arn> --role-session-name ci --web-  
identity-token $TOKEN
```

Interview tip: Always constrain the sub claim to a specific repo and branch or environment. A trust policy that accepts any repository is a real vulnerability.

Official docs

- [AWS IAM: Temporary credentials \(STS\)](#)
- [GitHub Actions documentation](#)

 Spoken answer · ~58 s

With the CodeDeploy deployment controller, an ECS service has two target groups on the load balancer. When a deployment starts, CodeDeploy creates a replacement task set running the new task definition and registers it with the idle target group. It can send test traffic through a test listener and run Lambda hooks, for example `BeforeAllowTraffic`, to smoke test. Then it shifts the production listener to the new target group, either all at once, in linear steps like ten percent every minute, or as a canary with a small slice first. CloudWatch alarms monitored during the shift can trigger an automatic rollback by sending traffic back to the original task set, which stays running for a configurable wait time before termination. The AppSpec file defines the task definition, container and port. This gives fast, low-risk releases without in-place changes.

Likely follow-ups

- What does the termination wait time do?
- How would you do database migrations with blue/green?

CodeDeploy launches a new task set behind a second target group, shifts the ALB listener traffic all at once, linearly or as a canary, runs lifecycle hook tests, and can roll back automatically on alarms. The old task set is kept briefly for fast rollback.

```
aws deploy create-deployment --application-name web --deployment-group-name prod --revision
file://appspec.json
```

```
aws deploy get-deployment --deployment-id d-ABC123 --query 'deploymentInfo.status'
```

Interview tip: Use a test listener port and a `BeforeAllowTraffic` hook to smoke test before real users see the new version.

 Official docs

- [AWS CodeDeploy](#)
- [Amazon ECS Developer Guide](#)

Q147 **A Lambda function in a VPC times out when calling RDS or an external API. How do you debug it?**

Troubleshooting

 Spoken answer · ~62 s

I begin with the symptom. A connection that hangs until the timeout usually means packets are silently dropped, while an immediate error suggests a refusal or wrong credentials. First I look at the function's VPC configuration: which subnets and security groups it uses. For RDS, the database security group must allow the function's security group on the database port, and the subnets must be able to route to it. For an external API, a Lambda in a private subnet needs a route through a NAT Gateway, or a VPC endpoint if the target is an AWS service, because it does not get a public address. I also check network ACLs, DNS resolution, and whether the function timeout is shorter than the downstream call. For databases I look at connection exhaustion and use RDS Proxy, and I reuse connections across invocations. Reachability Analyzer is helpful for confirming the path.

Likely follow-ups

- How does Reachability Analyzer help?
- Why is NAT needed for outbound internet from a VPC Lambda?

Check the function's subnets, security groups and routes. Reaching RDS needs a security group rule from the function to the database port. Reaching the internet needs a NAT Gateway, since Lambda in a VPC has no public IP. Also check the timeout and connection reuse.

```
aws lambda get-function-configuration --function-name f --query 'VpcConfig'
```

```
aws ec2 describe-route-tables --filters Name=association.subnet-id,Values=subnet-1
```

```
aws logs tail /aws/lambda/f --follow
```

Interview tip: Timeouts, rather than refusals, usually mean a dropped packet: missing route, security group or NACL.

 Official docs

- [Amazon RDS User Guide](#)
- [AWS Lambda Developer Guide](#)

Q148 An ALB target group shows targets as unhealthy. How do you find out why?

Troubleshooting

 Spoken answer · ~58 s

The describe target health output includes a reason code that points to the layer. Target.Timeout means the health check could not connect, often because the target security group does not allow the load balancer's security group on the traffic or health check port, a network ACL blocks it, or the app is not listening. Target.ResponseCodeMismatch means the app responded but with a code that is not in the configured matcher, like a redirect or an authentication error on the health path. Target.FailedHealthChecks can be a slow response or a flapping app. I test from within the VPC using curl against the exact path and port, confirm the container port mapping and bind address, and check the app logs. Then I tune the interval, thresholds and timeout, and use a lightweight health endpoint that checks readiness without hitting heavy dependencies.

Likely follow-ups

- What path should a health check use?
- How does ALB health differ from ASG health?

Read the health check reason code, then verify the path, port and success codes match what the app serves, the security group allows the ALB to reach the target, the app is listening, and timeouts and thresholds are realistic.

```
aws elbv2 describe-target-health --target-group-arn <arn>
```

```
aws elbv2 describe-target-groups --target-group-arns <arn> --query 'TargetGroups[0].  
[HealthCheckPath,HealthCheckPort,Matcher]'
```

```
curl -v http://<target-ip>:8080/health
```

Interview tip: The reason field, such as Target.Timeout or Target.ResponseCodeMismatch, tells you which layer to look at.

Official docs

- [Elastic Load Balancing](#)
- [Application Load Balancer](#)

Q149 **New EKS worker nodes launch but never join the cluster. What do you check?**

Troubleshooting

 Spoken answer · ~63 s

I start on the instance with the console output or, if I can connect via Session Manager, the cloud-init and kubelet logs. Common causes are these. The node's IAM role is not authorised in the cluster, which means a missing access entry or aws-auth mapping, so the kubelet gets unauthorized errors. The bootstrap parameters are wrong, such as a wrong cluster name or endpoint, or a custom AMI missing the bootstrap. Networking problems stop the node reaching the API server: private endpoint only without a route, security groups missing the cluster security group rules, or no access to ECR, S3 and STS through NAT or endpoints. The VPC CNI may not be able to assign IPs, leaving the node NotReady. I fix the specific gap, then terminate and let the group replace the node. To avoid this, I use managed node groups and test launch template changes in a non-production cluster.

Likely follow-ups

- What replaced aws-auth ConfigMap?
- Why would a node be NotReady after joining?

Verify the node IAM role is mapped through an access entry or aws-auth, the bootstrap script and cluster name are correct, the node can reach the API endpoint and required AWS endpoints, security groups allow control plane traffic, and the CNI and kubelet start.

```
aws eks list-access-entries --cluster-name prod
```

```
kubectl get cm aws-auth -n kube-system -o yaml
```

```
sudo journalctl -u kubelet --no-pager | tail -50
```

```
aws ec2 get-console-output --instance-id i-1
```

Interview tip: Check the console output or system log first. Bootstrap errors usually appear there in plain text.

 Official docs

- [Amazon EKS: User guide](#)

Q150 A pod using IRSA gets 'WebIdentityErr' or AccessDenied from STS. What is wrong?

IAM & Security

 Spoken answer · ~59 s

The error means STS refused to exchange the pod's token for role credentials. I check from the pod outward. First, are the AWS role ARN and web identity token file environment variables present in the pod? If not, the service account annotation is missing or the pod was created before it was added, so it needs a restart. Next, I compare the cluster's OIDC issuer URL with the OIDC provider registered in IAM, since a mismatch or a deleted provider breaks everything. Then I read the role's trust policy: the principal must be that provider, the audience should be sts.amazonaws.com, and the subject condition must exactly match system:serviceaccount:namespace:name. Typos in the namespace or name are very common. Finally, I make sure the SDK in the image is recent enough to support web identity. I now prefer EKS Pod Identity for simpler trust.

Likely follow-ups

- What does the sub condition look like?
- Why must the pod restart after annotation?

Check that the service account annotation and the pod's token projection exist, the OIDC provider is registered in IAM, and the role trust policy's sub and aud conditions exactly match the namespace and service account name.

```
kubectl get pod <p> -o yaml | grep -E 'AWS_ROLE_ARN|AWS_WEB_IDENTITY_TOKEN_FILE'
```

```
aws iam get-role --role-name app --query 'Role.AssumeRolePolicyDocument'
```

```
aws eks describe-cluster --name prod --query 'cluster.identity.oidc.issuer'
```

Interview tip: The pod must be restarted after annotating the service account, because the token is injected at pod creation.

Official docs

- [AWS IAM: Temporary credentials \(STS\)](#)
- [Amazon EKS: IAM roles for service accounts \(IRSA\)](#)

 Spoken answer · ~61 s

I look at which status check is failing. A system status check failure means a problem with the underlying AWS infrastructure, such as host hardware or network connectivity. Stopping and starting the instance, not just rebooting, normally moves it to healthy hardware, and I can set a CloudWatch alarm with an auto-recover action for this. An instance status check failure means the operating system is not responding correctly, which could be a bad fstab, a full root volume, a kernel panic, broken network configuration or exhausted memory. I read the system log and screenshot from the console, and if I cannot log in, I stop the instance, detach the root volume, attach it to a rescue instance, fix the files, and reattach. For an Auto Scaling group I let health checks replace the instance, but still investigate the cause, because data on instance store volumes disappears.

Likely follow-ups

- Reboot versus stop and start?
- How do you set up auto-recovery?

System check failure points to AWS hardware or host, so stop and start the instance to move it to new hardware. Instance check failure points to the OS: boot problems, full disk, kernel panic, or bad network config. Use console output and screenshot, or attach the volume to a rescue instance.

```
aws ec2 describe-instance-status --instance-ids i-1
```

```
aws ec2 get-console-output --instance-id i-1 --latest
```

```
aws ec2 stop-instances --instance-ids i-1 && aws ec2 start-instances --instance-ids i-1
```

Interview tip: Remember reboot keeps the host, but stop then start usually migrates the instance. Instance store data is lost on stop.

 Official docs

- [Amazon EC2 User Guide](#)

Q152 **A VPC peering connection is active but instances cannot reach each other. What do you check?**

Networking

 Spoken answer · ~61 s

An active peering connection only creates the link. Nothing flows until the routes exist. So I check that the route table of each subnet in both VPCs has a route for the other VPC's CIDR with the peering connection as the target. Overlapping CIDRs make peering impossible or ambiguous, so I confirm they are distinct. Then I verify security groups, which can reference peer security groups in the same region, and network ACLs allow the ports in both directions. Peering is not transitive, so traffic cannot pass through a third VPC or through a VPN attached to one side, in which case I would use Transit Gateway. If connections work by IP but names fail, I enable DNS resolution over the peering for both sides. I use VPC Reachability Analyzer to verify the path, and then automate routes in infrastructure as code so they are not forgotten.

Likely follow-ups

- Why does peering not allow transit?
- When would you move to Transit Gateway?

Both VPCs need route table entries pointing the other CIDR to the peering connection, security groups and NACLs must allow the traffic, CIDRs must not overlap, and peering is not transitive. DNS resolution across peering must be enabled if you use private names.

```
aws ec2 describe-vpc-peering-connections --query 'VpcPeeringConnections[].[VpcPeeringConnectionId,Status.Code]'
```

```
aws ec2 describe-route-tables --filters Name=route.vpc-peering-connection-id,Values=pcx-1
```

```
aws ec2 modify-vpc-peering-connection-options --vpc-peering-connection-id pcx-1 --requester-peering-connection-options AllowDnsResolutionFromRemoteVpc=true
```

Interview tip: Reachability Analyzer shows exactly which hop blocks the path.

 Official docs

- [Amazon VPC: VPC peering](#)

Q153 **Private hosted zone names resolve in one VPC but not another. How do you fix it?**

Networking

 Spoken answer · ~62 s

A private hosted zone is attached to specific VPCs and only those VPCs' resolvers can answer for it. So the first step is to list the zones associated with the failing VPC. If the zone is missing, I associate the VPC with the hosted zone. When the zone and the VPC are in different accounts, the zone owner creates an association authorization and then the VPC owner completes the association. I also confirm that DNS support and DNS hostnames are enabled on the VPC, otherwise the VPC resolver does not work for private zones. For many VPCs, I would use Route 53 Resolver rules or share the zone via Resource Access Manager or Route 53 Profiles, rather than associating one by one. If hybrid names are involved, I check that resolver endpoints and forwarding rules are in place. Finally, I test with dig against the VPC resolver address.

Likely follow-ups

- What is the VPC resolver address?
- How do Route 53 Profiles simplify this?

A private hosted zone only answers queries from VPCs associated with it. Associate the other VPC, which for cross-account VPCs needs an authorization and association call, and make sure `enableDnsSupport` and `enableDnsHostnames` are on.

```
aws route53 list-hosted-zones-by-vpc --vpc-id vpc-2 --vpc-region eu-west-1
```

```
aws route53 associate-vpc-with-hosted-zone --hosted-zone-id Z123 --vpc VPCRegion=eu-west-1,VPCId=vpc-2
```

```
aws ec2 describe-vpc-attribute --vpc-id vpc-2 --attribute enableDnsSupport
```

Interview tip: For cross-account, first run `create-vpc-association-authorization` in the zone's account, then associate from the VPC's account.

 Start here (official docs)

- [AWS documentation](#)

Q154 **A CloudFormation stack is stuck in UPDATE_ROLLBACK_FAILED. How do you recover?**

CI/CD

 Spoken answer · ~60 s

This state means CloudFormation tried to roll back a failed update, but could not restore one of the resources. I read the stack events to find the first failure and the specific reason. Typical causes are someone manually deleting or modifying a resource the rollback needs, an IAM permission missing for the stack's role, a resource dependency, or a service limit. I fix the underlying problem, for example recreating the deleted item or granting the role permission, and then call continue update rollback. If a resource cannot be restored, I can tell CloudFormation to skip it, which moves the stack to a stable state but leaves that resource out of sync, so I then check drift and repair it. As prevention, I stop manual changes with guardrails, use change sets to review updates, set rollback triggers, and use stack policies for critical resources.

Likely follow-ups

- What is a change set?
- How does drift detection work?

Find the resource that failed to roll back in the stack events, fix the cause such as a manually changed or deleted resource, then use `ContinueUpdateRollback`, optionally skipping the stuck resources. Afterwards, address drift and retry the update.

```
aws cloudformation describe-stack-events --stack-name app --query 'StackEvents[?contains(ResourceStatus, `FAILED`)].[LogicalResourceId,ResourceStatusReason]'
```

```
aws cloudformation continue-update-rollback --stack-name app --resources-to-skip MyBucket
```

```
aws cloudformation detect-stack-drift --stack-name app
```

Interview tip: Skipping a resource leaves it out of sync, so you must reconcile it manually afterwards.

 Official docs

- [Kubernetes: Deployments](#)
- [AWS CloudFormation](#)

Q155 **An ECS deployment keeps failing and rolling back because new tasks fail health checks. How do you get to the cause?**

CI/CD

 Spoken answer · ~61 s

The service events and the stopped task details usually tell the story. A stopped reason such as Task failed ELB health checks means the app started but was not answering correctly, while an exit code of one thirty-seven suggests an out-of-memory kill and two-fifty-five or one points to an application crash. I read the container logs for startup errors, such as missing environment variables, secrets that the execution role cannot read, or a database that cannot be reached. I then verify the port mapping, the target group health check path and matcher, and that the security groups allow the load balancer. For slow starters I add a health check grace period and a longer deregistration timing. Circuit breaker rollback is helpful because it returns to the last good version automatically, but I always fix the underlying issue and test the task definition in a staging service first.

Likely follow-ups

- What is the ECS deployment circuit breaker?
- What does exit code 137 mean?

Look at the service events and stopped task reasons, then the container logs and exit code. Common causes are a wrong container port or health check path, missing secrets or environment, a slow startup that needs a grace period, and insufficient memory.

```
aws ecs describe-services --cluster prod --services web --query 'services[0].events[:10]'
```

```
aws ecs describe-tasks --cluster prod --tasks <arn> --query 'tasks[0].
[stoppedReason,containers[0].exitCode]'
```

```
aws logs tail /ecs/web --since 15m
```

Interview tip: Set healthCheckGracePeriodSeconds for slow-starting apps so the load balancer does not kill them during startup.

 Official docs

- [Kubernetes: Deployments](#)
- [Amazon ECS Developer Guide](#)

Q156 **The root EBS volume is almost full on a production instance. How do you expand it without downtime?**

Storage

 Spoken answer · ~60 s

First, I take a snapshot for safety. Then I use the volume modification API to increase the size, which works online for current generation instances. I wait for the modification to reach the optimizing state, which is enough for use. Inside the instance I grow the partition with growpart, then extend the filesystem using resize2fs for ext4 or xfs_growfs for XFS. There is a cooldown of about six hours before the same volume can be modified again, so I choose a size with headroom. Expanding is a stopgap, so I find why it filled: logs without rotation, old Docker images, core dumps or temp files, using du and checking deleted files still held open with lsof. I fix log rotation, move data to a separate volume, and set a CloudWatch alarm on disk usage with the agent, because disk metrics are not collected by default.

Likely follow-ups

- Why check inodes?
- Can you shrink an EBS volume?

Modify the volume size online, then grow the partition and filesystem inside the OS, with growpart and resize2fs or xfs_growfs. You can modify once every six hours per volume. Take a snapshot first, and find what filled the disk.

```
aws ec2 modify-volume --volume-id vol-1 --size 200
```

```
lsblk; sudo growpart /dev/nvme0n1 1
```

```
sudo xfs_growfs -d / # or: sudo resize2fs /dev/nvme0n1p1
```

```
du -xh / --max-depth=2 | sort -h | tail
```

Interview tip: Check inodes with `df -i` as well. A disk can be full of tiny files while showing free space.

Official docs

- [Amazon EBS](#)

Q157 Important objects were deleted from an S3 bucket. How do you recover them, and how do you prevent a repeat?

Storage

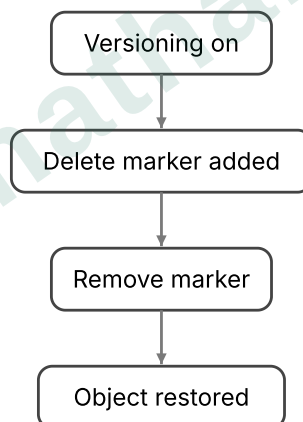
Spoken answer · ~56 s

My action depends on whether versioning was enabled. With versioning, a normal delete just places a delete marker on top, so I list the versions and either delete the marker or copy the previous version to become current again, which I can script for many keys. If a specific version was deleted, it is gone, so I check cross-region or same-region replication targets, other backups, AWS Backup for S3, or application-level copies. I also check CloudTrail data events, if enabled, to learn who deleted what and why. To prevent repeats, I enable versioning on important buckets, add lifecycle rules for noncurrent versions, consider Object Lock in compliance or governance mode for critical data, require MFA delete for permanent deletions, restrict delete permissions, and replicate to a separate account. Regular restore drills prove it works.

Likely follow-ups

- What does Object Lock compliance mode guarantee?
- How do you restore millions of objects?

S3 recovery



If versioning was on, the delete only added a delete marker, so remove the marker or copy the earlier version back. Without versioning, check replication targets, backups, or AWS Backup. Prevent with versioning, MFA delete, Object Lock and restrictive policies.

```
aws s3api list-object-versions --bucket b --prefix path/
```

```
aws s3api delete-object --bucket b --key path/file --version-id <delete-marker-id>
```

```
aws s3api put-bucket-versioning --bucket b --versioning-configuration Status=Enabled
```

Interview tip: Deleting a specific version is permanent, so only users who truly need it should have `s3:DeleteObjectVersion`.

Official docs

- [Amazon S3 User Guide](#)

Q158 After an RDS Multi-AZ failover, the application kept failing for several minutes. Why, and how do you fix it?

Databases

Spoken answer · ~60 s

During failover RDS promotes the standby and updates the DNS record of the endpoint, which typically takes one to two minutes, and existing connections drop. Applications fail longer than that when they cache the old IP address, for example a JVM with an infinite DNS cache TTL, or keep broken pooled connections and do not validate them. Fixing it involves several steps. I make sure the runtime respects a short DNS TTL, configure the connection pool to test connections and evict broken ones, and add retries with exponential backoff and jitter for transient errors. I set sensible connect and read timeouts so threads do not hang. RDS Proxy can mask much of the failover by keeping client connections open. I schedule a forced failover in staging regularly, observe how the app behaves, and document the expected recovery time in the runbook and alerts.

Likely follow-ups

- What is JVM `networkaddress.cache.ttl`?
- How does RDS Proxy reduce failover impact?

The endpoint DNS changes to the new primary, but apps cache DNS or hold dead connections. Use short DNS TTL honoured by the runtime, connection validation and retry with backoff, pool limits, and RDS Proxy to shield clients.

```
aws rds describe-events --source-identifier prod --source-type db-instance --duration 120
```

```
dig +short prod.abc.eu-west-1.rds.amazonaws.com
```

```
aws rds reboot-db-instance --db-instance-identifier prod --force-failover
```

Interview tip: Test failover on purpose in staging. It is the only way to know your connection handling works.

Official docs

- [Amazon RDS: Multi-AZ deployments](#)
- [Amazon RDS User Guide](#)

Q159 **A DynamoDB table is throttling even though consumed capacity is below the provisioned limit. What is happening?**

Databases

 Spoken answer · ~60 s

Capacity is spread across partitions, and each partition has its own throughput ceiling. If requests concentrate on a small set of keys, such as a date, a popular product or a status value, a single partition can throttle while the table total still looks low. I confirm by looking at throttle metrics and enabling CloudWatch Contributor Insights to see the hottest keys. Fixes depend on the pattern. I redesign the partition key for higher cardinality, or add a shard suffix to spread writes and aggregate on read. For read-heavy hot items, I add DAX or an application cache. Adaptive capacity helps with some imbalance, but it is not a substitute for good key design. A global secondary index can also throttle the base table if its own key is hot, so I check those. On-demand mode removes capacity planning but does not remove partition limits.

Likely follow-ups

- What is write sharding?
- How does adaptive capacity help?

A hot partition. Traffic concentrated on a few partition keys exceeds the per-partition limits, about 1,000 writes and 3,000 reads per second. Fix the key design with higher-cardinality keys, write sharding, caching, or on-demand mode, and check for hot GSIs.

```
aws cloudwatch get-metric-statistics --namespace AWS/DynamoDB --metric-name
WriteThrottleEvents --dimensions Name=TableName,Value=orders --start-time 2026-10-
20T00:00:00Z --end-time 2026-10-21T00:00:00Z --period 300 --statistics Sum
```

```
aws dynamodb describe-table --table-name orders
```

Interview tip: Contributor Insights shows the most accessed partition keys directly, so enable it during diagnosis.

 Official docs

- [Microsoft: Cloud design pattern – Throttling](#)
- [Amazon DynamoDB Developer Guide](#)

 Spoken answer · ~57 s

Unlike classic RDS replicas, Aurora replicas read from the same distributed storage as the writer, so lag is typically small and relates to the replica refreshing its in-memory cache and page versions. If lag grows I first look at the writer: a burst of heavy writes, bulk loads or very large transactions create lots of changes for the replicas to apply. Then I check the replica itself: CPU or memory pressure, an instance class much smaller than the writer, and long-running queries that conflict with replay, which can delay or cancel work. Performance Insights shows the waits and top SQL. Fixes include breaking up large batch writes, right-sizing replicas, killing or optimising slow reporting queries, routing heavy analytics to a dedicated replica, and making the application tolerant of slight staleness by sending read-after-write requests to the writer.

Likely follow-ups

- Which reads must go to the writer?
- How do Aurora custom endpoints help?

Aurora replicas share storage, so lag is normally milliseconds. Growing lag usually means heavy write volume, long-running queries on the replica blocking replay, an undersized replica, or a large transaction. Check AuroraReplicaLag, CPU, and Performance Insights.

```
aws cloudwatch get-metric-statistics --namespace AWS/RDS --metric-name AuroraReplicaLag --dimensions Name=DBInstanceIdentifier,Value=prod-ro1 --start-time 2026-10-21T00:00:00Z --end-time 2026-10-21T06:00:00Z --period 60 --statistics Maximum
```

```
SELECT pid, now() - query_start AS dur, query FROM pg_stat_activity ORDER BY dur DESC LIMIT 5;
```

Interview tip: In Aurora, a replica that is smaller than the writer can struggle to apply changes during heavy write bursts.

 Official docs

- [Amazon RDS: Read replicas](#)
- [Amazon Aurora User Guide](#)

Q161 **The CloudWatch bill is suddenly very high. Where do you look and how do you cut it?**

Cost Opt

 Spoken answer · ~55 s

I start with Cost Explorer filtered to CloudWatch and grouped by usage type, which separates log ingestion, log storage, custom metrics, alarms and API requests. If ingestion dominates, I find the noisiest log groups using stored bytes and incoming bytes metrics, and typically see debug logging left on, a chatty library, or a retry storm. I lower log levels, sample noisy events, filter before shipping, and set retention on every group, with older data archived to S3. If custom metrics are the issue, high-cardinality dimensions such as user ID or request ID create enormous numbers of metrics, so I remove those dimensions or use embedded metric format with limited dimensions. VPC Flow Logs can go to S3 instead, which is cheaper. Finally I add a budget alert and anomaly detection on CloudWatch spend.

Likely follow-ups

- Why is cardinality costly for metrics?
- When would you ship logs to S3 instead?

Break the CloudWatch charges down by usage type. Log ingestion and storage, custom metrics with high-cardinality dimensions, detailed monitoring, vended logs such as VPC Flow Logs, and API polling are the usual drivers.

```
aws ce get-cost-and-usage --time-period Start=2026-10-01,End=2026-10-21 --granularity DAILY
--metrics UnblendedCost --filter '{"Dimensions":{"Key":"SERVICE","Values":
["AmazonCloudWatch"]}}' --group-by Type=DIMENSION,Key=USAGE_TYPE
```

```
aws logs describe-log-groups --query 'sort_by(logGroups,&storedBytes)[-10:].
[logGroupName,storedBytes,retentionInDays]'
```

Interview tip: Setting a retention period on every log group is the cheapest quick win, because the default is to keep logs forever.

 Official docs

- [Google SRE Book: Effective Troubleshooting](#)
- [GNU coreutils manual](#)
- [Amazon CloudWatch User Guide](#)

Q162 **Savings Plan coverage dropped after migrating workloads to Fargate and Graviton. What happened and what do you do?**

Cost Opt

 Spoken answer · ~56 s

I compare utilisation and coverage. Utilisation shows how much of what we committed is actually being used, and coverage shows how much of our eligible spend is covered. After the migration, an EC2 Instance Savings Plan tied to old instance families may sit unused, because those instances were retired, while Fargate and Graviton usage is billed at on-demand rates. A Compute Savings Plan would have followed the workload across EC2, Fargate and Lambda and across families, including Graviton. Remedies: if the unused commitment is large, I can move workloads back temporarily, use it up where possible, wait for expiry and avoid renewal. For new usage I buy a Compute Savings Plan sized to the new stable baseline. Going forward, I time commitment purchases after migrations settle, buy in smaller steps, and review monthly with finance.

Likely follow-ups

- Can Savings Plans be cancelled?
- Why keep some baseline uncommitted?

Instance-specific commitments, like EC2 Instance Savings Plans or RIs, do not cover Fargate, and changing architecture can leave commitments unused while new usage is billed on-demand. Review coverage and utilisation, and prefer Compute Savings Plans.

```
aws ce get-savings-plans-utilization --time-period Start=2026-09-01,End=2026-10-21
```

```
aws ce get-savings-plans-coverage --time-period Start=2026-09-01,End=2026-10-21 --granularity MONTHLY
```

Interview tip: Plan commitments around the target architecture, and keep some baseline uncommitted while migrations are in flight.

 Official docs

- [AWS Savings Plans](#)
- [Amazon ECS Developer Guide](#)

Q163 Lambda throttling in one function is starving other functions in the account. How do you contain it?

Compute

 Spoken answer · ~58 s

Lambda has a regional account limit on concurrent executions, shared by all functions unless reserved. A flood of invocations in one function, such as a retry storm or a large batch from S3 or SQS, can consume that pool and cause throttling for unrelated, more important functions. To contain it, I set reserved concurrency on the critical functions, which guarantees them capacity, and on the noisy one to cap how much it can take, which also protects downstream databases. For bursty producers I introduce an SQS queue between them and the function, with a batch size and maximum concurrency on the event source mapping, so the load is smoothed rather than spiking. I request a higher account limit when justified, and set alarms on the Throttles and ConcurrentExecutions metrics. For latency sensitive functions I can add provisioned concurrency.

Likely follow-ups

- Reserved versus provisioned concurrency?
- How do you cap SQS-triggered concurrency?

All functions in a region share the account concurrency limit. Set reserved concurrency on important functions to guarantee capacity and on noisy ones to cap them, add queues to smooth bursts, request a quota increase, and alarm on throttles.

```
aws lambda put-function-concurrency --function-name checkout --reserved-concurrent-executions 200
```

```
aws lambda get-account-settings --query 'AccountLimit.[ConcurrentExecutions,UnreservedConcurrentExecutions]'
```

```
aws cloudwatch get-metric-statistics --namespace AWS/Lambda --metric-name Throttles --dimensions Name=FunctionName,Value=ingest --start-time 2026-10-21T00:00:00Z --end-time 2026-10-21T06:00:00Z --period 60 --statistics Sum
```

Interview tip: Reserved concurrency is both a guarantee and a hard cap for that function, so size it carefully.

 Official docs

- [AWS Lambda: Concurrency](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [AWS Lambda Developer Guide](#)

Q164 **Instances in an Auto Scaling group keep terminating and relaunching in a loop. How do you stop it and find the cause?**

Compute

 Spoken answer · ~60 s

The scaling activity history gives the termination reason, for example failed ELB health checks or an EC2 status check failure. The common cause is that new instances are marked unhealthy before the application is ready, because bootstrap takes longer than the health check grace period, or the health check path or port is wrong, or a user data script fails. To investigate I temporarily suspend the terminate process, or put an instance in standby, so I can log in and read cloud-init and application logs. I test the launch template alone by launching a single instance. Fixes include extending the grace period, baking the AMI to cut boot time, correcting the health check, and fixing the failing script, then resuming the processes. I also use lifecycle hooks to wait for readiness, and set alarms on the unhealthy host count so such loops are noticed quickly.

Likely follow-ups

- What does standby do?
- Why bake AMIs instead of long user data scripts?

Usually the health check fails right after launch, from slow bootstrap, a wrong health check path, or an ELB check starting before the app is up. Look at scaling activity history, set a health check grace period, and test the launch template in isolation.

```
aws autoscaling describe-scaling-activities --auto-scaling-group-name web --max-items 10
```

```
aws autoscaling update-auto-scaling-group --auto-scaling-group-name web --health-check-grace-period 600
```

```
aws autoscaling suspend-processes --auto-scaling-group-name web --scaling-processes Terminate
```

Interview tip: Suspending the Terminate process lets you keep one failing instance alive to inspect, but resume it afterwards.

 Official docs

- [Amazon EC2 Auto Scaling](#)

 Spoken answer · ~59 s

The stopped reason usually states the cause. If it says the image was not found, I check the repository, tag and region, since a typo or a tag that was never pushed is common. If it says access denied or no basic auth credentials, the task execution role lacks ECR permissions such as `GetAuthorizationToken` and `BatchGetImage`, or the repository policy blocks a cross-account pull. If it times out, the network path is the problem: in private subnets tasks need a route through a NAT Gateway or the VPC endpoints for ECR API, ECR Docker and the S3 gateway endpoint, because layers are stored in S3, and security groups must allow outbound HTTPS. For images from Docker Hub, rate limiting can happen, so I mirror them into ECR or use pull-through cache. Very large images also slow starts, so I slim them.

Likely follow-ups

- Why is an S3 endpoint needed for ECR?
- What is an ECR pull-through cache?

The task cannot fetch its image. Check the image name and tag, the execution role's ECR permissions, network path to ECR and S3 (NAT or endpoints) for private subnets, registry rate limits, and available disk or image size.

```
aws ecs describe-tasks --cluster prod --tasks <arn> --query 'tasks[0].stoppedReason'
```

```
aws ecr describe-images --repository-name web --image-ids imageTag=v42
```

```
aws ecr get-login-password | docker login --username AWS --password-stdin  
<acct>.dkr.ecr.eu-west-1.amazonaws.com
```

Interview tip: Private Fargate tasks need `ecr.api`, `ecr.dkr` and the S3 gateway endpoint, or a NAT Gateway.

 Official docs

- [Amazon ECS Developer Guide](#)

Q166 **An EKS control plane upgrade is blocked or add-ons break afterwards. How do you plan to avoid that?**

Containers

 Spoken answer · ~60 s

Preparation matters more than the upgrade command. I run upgrade readiness insights and tools like Pluto to find deprecated or removed APIs in the cluster and in Git. I read the release notes for the target version, confirm versions of the VPC CNI, CoreDNS, kube-proxy, CSI drivers, ingress controllers and any operators support it, and test the whole thing in a non-production cluster. The control plane upgrades first, one minor version at a time, and EKS keeps it available. Then I update the add-ons, and roll the node groups, preferably with surge nodes so capacity remains. Drains can hang because of strict PodDisruptionBudgets, so I check those beforehand, and ensure workloads have more than one replica. I take backups of state like etcd-equivalent manifests, plan a maintenance window, and keep a rollback plan of building a new node group on the previous version.

Likely follow-ups

- Why can PDBs block node drains?
- What is a blue/green cluster upgrade?

Check upgrade insights for deprecated APIs, confirm add-on and node versions are compatible, upgrade one minor version at a time, upgrade add-ons and managed node groups after the control plane, and use PodDisruptionBudgets that allow drains.

```
aws eks list-insights --cluster-name prod --filter category=UPGRADE_READINESS
```

```
aws eks describe-addon-versions --kubernetes-version 1.33 --addon-name vpc-cni
```

```
aws eks update-cluster-version --name prod --kubernetes-version 1.33
```

Interview tip: A PDB with maxUnavailable zero and a single replica makes node drains hang forever, so audit PDBs before the upgrade.

 Official docs

- [Amazon EKS: Updating a cluster](#)
- [Amazon EKS: User guide](#)

Q167 **A CloudWatch alarm sits in INSUFFICIENT_DATA and never fires during a real outage. What is wrong?**

Monitoring

 Spoken answer · ~60 s

If the thing you monitor stops sending metrics, an error-count alarm has nothing to evaluate, so it goes to insufficient data instead of alarm. During a real outage, when the service is down and not emitting errors or traffic, the alarm stays quiet. I first check the alarm configuration: the metric name, namespace and dimensions must exactly match the data actually published, since a mismatch gives permanent insufficient data. Then I choose how to treat missing data. For availability signals, I treat missing as breaching, so no data means a problem, whereas for sparse metrics I treat it as not breaching. I also add alarms for the absence of expected traffic, for example request count below one for five minutes, use synthetic canaries that test the endpoint from outside, and a heartbeat metric. Alarms should be tested by deliberately causing failures in staging.

Likely follow-ups

- What are CloudWatch Synthetics canaries?
- When do you treat missing data as not breaching?

The alarm is not receiving data points, or the missing-data treatment hides the problem. A crashed service emits no metrics, so default behaviour stays in insufficient data. Set treat missing data to breaching, or alarm on a heartbeat or the absence of metrics.

```
aws cloudwatch describe-alarms --alarm-names api-5xx --query 'MetricAlarms[0].
[StateValue,TreatMissingData,Period,EvaluationPeriods]'
```

```
aws cloudwatch put-metric-alarm --alarm-name api-heartbeat --metric-name RequestCount --
namespace AWS/ApplicationELB --statistic Sum --period 60 --evaluation-periods 5 --threshold
1 --comparison-operator LessThanThreshold --treat-missing-data breaching
```

Interview tip: Alerting on error counts alone fails when the service is completely down, so also alarm on lack of traffic.

Official docs

- [Amazon CloudWatch: Alarms](#)
- [Amazon CloudWatch User Guide](#)

Q168 **A Lambda or ECS service runs but no logs appear in CloudWatch. What do you check?**

Monitoring

 Spoken answer · ~60 s

I work through the path logs take. Does the application write to standard output or standard error, rather than to a file inside the container? For Lambda, the execution role needs permission to create the log group and streams and put events, and a deleted log group or a missing permission is a frequent cause. For ECS, the task definition needs a log configuration using the awslogs driver or FireLens, with the right group, region and stream prefix, and the task execution role needs logs permissions. In private subnets there must be a route to CloudWatch Logs through NAT or a VPC endpoint. I also check whether a log group retention or resource policy exists, whether log volume is hitting service throttles, and if a sidecar log router is failing. I confirm using simulate-principal-policy and the agent or router logs, then fix and redeploy.

Likely follow-ups

- Why does ECS need an execution role for logs?
- What is FireLens?

Check the role has logs:CreateLogStream and PutLogEvents, the log group exists or can be created, the task definition log driver and options are right, network access to the logs endpoint exists from private subnets, and the app actually writes to stdout.

```
aws logs describe-log-groups --log-group-name-prefix /aws/lambda/f
```

```
aws ecs describe-task-definition --task-definition web --query  
'taskDefinition.containerDefinitions[0].logConfiguration'
```

```
aws iam simulate-principal-policy --policy-source-arn <role-arn> --action-names  
logs:PutLogEvents
```

Interview tip: For ECS, the execution role does log delivery, not the task role.

 Official docs

- [AWS Lambda Developer Guide](#)
- [Amazon CloudWatch User Guide](#)
- [Amazon ECS Developer Guide](#)

Q169 **Assuming a role across accounts fails with 'not authorized to perform sts:AssumeRole'. How do you fix it?**

IAM & Security

 Spoken answer · ~62 s

Role assumption across accounts requires agreement from both sides. In the target account, the role's trust policy must list the caller as a trusted principal: the account, a specific role, or user. In the source account, the calling identity needs an identity policy that allows sts:AssumeRole on that role's ARN, unless the trust policy names the exact principal and it is in the same account. When it fails, I read the trust policy for typos in account IDs and role names, and conditions like ExternalId, MFA required, source IP, or an organization ID that the request does not satisfy. I also check for permission boundaries and service control policies that deny assume role, and whether the session is a role chain, which has a one hour limit. I use the policy simulator and CloudTrail events to see the exact denial reason, then correct whichever side is missing and retest.

Likely follow-ups

- What is an ExternalId for?
- What is the role chaining limit?

Cross-account access needs two sides: the caller's identity policy allows sts:AssumeRole on the target role, and the target role's trust policy names the caller's account or principal. Check conditions such as ExternalId, MFA, and SCPs.

```
aws iam get-role --role-name Deploy --query 'Role.AssumeRolePolicyDocument'
```

```
aws sts assume-role --role-arn arn:aws:iam::222222222222:role/Deploy --role-session-name test
```

```
aws iam simulate-principal-policy --policy-source-arn <caller-arn> --action-names sts:AssumeRole --resource-arns <role-arn>
```

Interview tip: If the trust policy names an account root, the caller still needs its own explicit allow. Both sides must agree.

 Official docs

- [AWS IAM: Roles](#)
- [AWS IAM: Temporary credentials \(STS\)](#)

Q170 A dead-letter queue is filling up with messages. How do you investigate and drain it safely?

Monitoring

 Spoken answer · ~59 s

A growing dead-letter queue means messages exceeded the maximum receive count on the source queue. First I sample a few messages and their attributes, and correlate with consumer logs by message ID, to find the failure pattern. It could be malformed payloads that will never succeed, a code bug on a specific input, a downstream dependency that was down, or a visibility timeout shorter than the processing time, which makes messages reappear mid-processing and eventually move. For poison messages I fix or discard them with a record of what happened. For transient failures, I fix the dependency, deploy any code fix, and then use the redrive feature to move messages back to the source queue, possibly with a controlled rate. I make consumers idempotent, set an alarm on DLQ visible messages and oldest message age, and keep a runbook for it.

Likely follow-ups

- How does the visibility timeout cause duplicates?
- What should a consumer do with poison messages?

Inspect messages and attributes to find why processing failed: poison payloads, bugs, downstream outages, or visibility timeout too short. Fix the cause, then redrive messages back to the source queue, and alarm on queue depth and age.

```
aws sqs get-queue-attributes --queue-url <dlq> --attribute-names
ApproximateNumberOfMessagesVisible
```

```
aws sqs receive-message --queue-url <dlq> --max-number-of-messages 5 --attribute-names All
--message-attribute-names All
```

```
aws sqs start-message-move-task --source-arn <dlq-arn> --destination-arn <source-queue-arn>
```

Interview tip: Only redrive after the bug is fixed, or the messages will just return to the DLQ and burn retries.

 Official docs

- [Kubernetes: Safely drain a node](#)
- [Amazon SQS](#)