

DevOps Interview Answers

Jenkins CI/CD, Docker, Kubernetes, and AWS Architecture

Scenario-based answers with practical steps and commands

Use this as an interview script: answer in 45-90 seconds, then expand with the scenario and 2-4 commands.

Section	Questions	What you should emphasize
CI/CD (Jenkins)	4	pipeline design, gates, promotion, security, K8s/AWS integration
Docker	3	real wins, image optimization, scanning/signing, ECR hygiene
Kubernetes	4	production debugging, workload types, HA, autoscaling
AWS & Architecture	4	HA patterns, multi-region decisions, IAM least privilege, DR and cost

Tip: When you mention commands, also explain the signal you are looking for (events, readiness flaps, throttling, failed health checks, replacements in plans, etc).

Quick Index

CI/CD (Jenkins) (4 questions)

- 1. How did you design a production pipeline?
- 2. How do you handle rollbacks?
- 3. How do you secure Jenkins in enterprise setups?
- 4. How did you integrate it with Docker, Kubernetes, and AWS?

Docker (3 questions)

- 1. What real problems did it solve in your environment?
- 2. How do you optimize images?
- 3. How do you handle image security and ECR management?

Kubernetes (4 questions)

- 1. Describe a real production issue you handled.
- 2. Deployment vs StatefulSet - when and why?
- 3. How do you troubleshoot CrashLoopBackOff?
- 4. How do you design HA and autoscaling?

AWS & Architecture (4 questions)

- 1. How do you design highly available systems?
- 2. Multi-AZ vs multi-region - when to use each?
- 3. How do you enforce least-privilege IAM?
- 4. How do you design disaster recovery and optimize cost?

CI/CD (Jenkins)

4 interview questions with suggested answers, scenarios, and practical steps.

Q1. How did you design a production pipeline?

Interview answer: I design production pipelines for safety, repeatability, and fast rollback. I keep a single source of truth (Git), build once and promote the same artifact across environments, add quality gates (tests, lint, SAST, image scan), require approvals for prod, and use progressive delivery (canary/blue-green) with automated health checks. I also make pipelines observable (logs, metrics) and idempotent (safe re-runs).

Scenario: We had ~30 services shipping to EKS. I standardized a Jenkins shared library so every repo used the same stages: build, test, scan, package, deploy to dev/stage, then promote to prod with an approval gate and SLO-based verification.

Practical approach

- Define stages: checkout -> unit tests -> build -> security scans -> package image -> push -> deploy -> verify -> promote.
- Build once: tag images with git SHA + semver; promote by reusing the same digest, not rebuilding.
- Use shared libraries for reuse; keep per-service differences in a small config file or Jenkinsfile params.
- Add gates: required checks, manual approval for prod, and automated post-deploy smoke tests.
- Protect secrets: use Jenkins credentials + AWS IAM roles; never store secrets in Jenkinsfile.
- Make it debuggable: artifact retention, timestamps, deployment links, and chat notifications.

Commands / snippets

```
# Example declarative Jenkinsfile outline (short)
pipeline {
  agent none
  options { timestamps(); disableConcurrentBuilds() }
  stages {
    stage('Test') { agent { label 'linux' }
      steps { sh 'make test' }
    }
    stage('Build image') { agent { label 'docker' }
      steps {
        sh 'docker build -t $IMAGE:$GIT_COMMIT .'
        sh 'trivy image --exit-code 1 --severity HIGH,CRITICAL $IMAGE:$GIT_COMMIT'
      }
    }
    stage('Push to ECR') { agent { label 'docker' }
      steps {
        sh 'aws ecr get-login-password | docker login --username AWS
          --password-stdin $ECR'
        sh 'docker push $IMAGE:$GIT_COMMIT'
      }
    }
    stage('Deploy (staging)') { agent { label 'kube' }
      steps { sh 'helm upgrade --install api charts/api -n staging -f
        values-staging.yaml --atomic' }
    }
    stage('Promote to prod') {
      when { branch 'main' }
      steps { input message: 'Approve prod deploy?' }
    }
  }
}
```

```
# Useful Jenkins patterns
# - shared library: vars/deploy.groovy
# - credentials binding: withCredentials(...)
```

Common pitfalls

- Rebuilding for prod instead of promoting the same artifact (causes 'works in staging, fails in prod').
- No concurrency controls causing overlapping deploys.
- Skipping post-deploy verification (green pipeline but broken user journey).

Q2. How do you handle rollbacks?

Interview answer: I aim for fast, low-risk rollback paths. For Kubernetes, I prefer immutable releases and versioned Helm charts so rollback is a single command. If we use GitOps, rollback is reverting the commit. If the failure involves data/schema changes, I avoid blind rollbacks and use forward fixes or feature-flag mitigations. The decision is based on MTTR and safety.

Scenario: A release increased 5xx errors due to a misconfigured downstream endpoint. We rolled back Helm release within minutes, then fixed the config and redeployed with validation checks added to CI.

Practical approach

- Confirm the blast radius and tie the issue to a recent change (deploy, config, flag).
- Choose rollback when it is safe (backward compatible DB/schema) and fast.
- If rollback is unsafe (irreversible migration), mitigate with flags/config and ship a hotfix.
- Automate rollback where possible: Helm --atomic, Argo Rollouts, or SLO-based canary abort.
- After recovery: write a prevention action (tests, policy, lint, canary gate).

Commands / snippets

```
# Kubernetes rollback patterns
kubectl rollout history deploy/api -n prod
kubectl rollout undo deploy/api -n prod
kubectl rollout status deploy/api -n prod --timeout=180s

# Helm rollback
helm history api -n prod
helm rollback api 12 -n prod
helm status api -n prod

# GitOps rollback (concept)
git revert <bad-commit>
git push origin main
```

Common pitfalls

- Rolling back application after irreversible DB/schema changes.
 - No clear 'last known good' version (missing tags, no release notes).
 - Rollback without verifying success using user-facing metrics.
-

Q3. How do you secure Jenkins in enterprise setups?

Interview answer: I secure Jenkins by reducing its attack surface, hardening authn/authz, isolating build execution, and protecting secrets. Enterprise Jenkins should use SSO (SAML/OIDC), RBAC (Matrix/Folder-based), least-privilege agents, signed plugins, TLS everywhere, audit logs, and controlled network egress. I also separate controller from ephemeral agents and avoid running builds on the controller.

Scenario: In an enterprise environment, we integrated Jenkins with corporate SSO, enforced RBAC per team folder, moved builds to Kubernetes-based ephemeral agents, and locked down credentials and network egress. This reduced lateral movement risk and credential leakage.

Practical approach

- Use SSO + MFA and disable local admin accounts where possible.
- Use RBAC: folders per team, least privilege, and separate admin vs developer permissions.
- Run builds on ephemeral agents (Kubernetes plugin) with minimal permissions; do not run builds on controller.
- Manage secrets via credentials store + external secret manager (AWS Secrets Manager/Vault).
- Harden plugins: minimal set, pinned versions, regular patching, and remove unused plugins.
- Network controls: private subnets, restrict inbound, allow-list outbound to SCM/registries only.
- Enable audit logs, backups, and disaster recovery for the controller.

Commands / snippets

```
# Common hardening checklist (commands are conceptual)
# - Jenkins URL behind reverse proxy with TLS
# - Kubernetes agents via Jenkins Kubernetes plugin

# Example: run controller with no executors
# Manage Jenkins -> Nodes -> set # of executors = 0

# Example AWS IAM for ECR push (least privilege snippet)
```

```
# Allow only required ECR actions on a single repo ARN and region
```

Common pitfalls

- Over-privileged Jenkins AWS keys shared across jobs.
 - Long-lived static secrets in jobs or environment variables.
 - Running Docker socket mounts on shared agents without isolation.
-

Q4. How did you integrate it with Docker, Kubernetes, and AWS?

Interview answer: I integrate Jenkins by using: (1) Docker build agents with BuildKit/buildx, (2) Kubernetes plugin for ephemeral agents, and (3) AWS IAM role assumption for ECR/EKS. Jenkins builds images, scans them, pushes to ECR, and deploys to EKS using Helm or GitOps. Artifacts are versioned by commit SHA and promoted across environments.

Scenario: A typical flow: PR builds run tests + scans; merges to main build and push to ECR; staging deploy happens automatically; prod deploy requires approval and runs helm upgrade --atomic with post-deploy checks.

Practical approach

- Use a standard image build stage with caching (BuildKit) and a security scan step.
- Authenticate to AWS using short-lived credentials (assume-role) instead of static access keys.
- Push images to ECR with immutable tags and lifecycle policy.
- Deploy to EKS using Helm (or ArgoCD) with environment-specific values files.
- Verify via rollout status + smoke tests; alert and rollback on failure.

Commands / snippets

```
# AWS auth best practice: assume role (example)
aws sts assume-role --role-arn arn:aws:iam::<acct>:role/jenkins-deployer
  --role-session-name jenkins

# ECR login + push
aws ecr get-login-password --region ap-south-1 | docker login --username AWS
  --password-stdin <acct>.dkr.ecr.ap-south-1.amazonaws.com
docker push <acct>.dkr.ecr.ap-south-1.amazonaws.com/api:$GIT_COMMIT

# EKS deploy
aws eks update-kubeconfig --name prod-eks --region ap-south-1
helm upgrade --install api charts/api -n prod -f values-prod.yaml --atomic --timeout
  10m
kubectl rollout status deploy/api -n prod --timeout=180s
```

Common pitfalls

- Using kubeconfig with cluster-admin permissions in Jenkins instead of scoped RBAC.
 - No separation between CI and CD permissions (build jobs should not deploy to prod).
 - No image scanning and no signing/verification controls.
-

Docker

3 interview questions with suggested answers, scenarios, and practical steps.

Q1. What real problems did it solve in your environment?

Interview answer: Docker solved environment drift (works on my machine), simplified packaging, and made CI consistent. We moved from snowflake servers to immutable images, so the runtime matched what we tested. It also improved deployment speed, rollback reliability, and team onboarding.

Scenario: Before Docker, different hosts had different OS packages and runtime versions. After containerizing services, we standardized base images, built once in CI, and ran the same image in dev/stage/prod.

Practical approach

- Containerize the app with a clear runtime contract (ports, env vars, volumes).
- Standardize base images and security updates via a controlled base image pipeline.
- Use registries as the source of truth and version images with commit SHA.
- Shift-left tests and scans in CI so runtime issues are caught early.

Commands / snippets

```
docker build -t org/api:$(git rev-parse --short HEAD) .
docker run --rm -p 8080:8080 org/api:<tag>
docker history org/api:<tag>
docker inspect org/api:<tag> | head
```

Common pitfalls

- Treating containers as VMs (running multiple unrelated processes, no health checks).
 - No tagging strategy or 'latest' everywhere.
 - Huge images causing slow deploys and wasted bandwidth.
-

Q2. How do you optimize images?

Interview answer: I optimize by reducing attack surface and size: multi-stage builds, slim/distroless base images, fewer layers, caching, .dockerignore, and running as non-root. I also pin base image versions and remove build tools from runtime stage.

Scenario: A Node service image was 1.2 GB and slow to deploy. After multi-stage build + npm prune + alpine base, it dropped below 200 MB and deployment time improved.

Practical approach

- Use multi-stage build: builder stage contains compilers; runtime stage contains only artifacts.
- Minimize layers: combine related RUN steps, but keep readability.
- Use .dockerignore to avoid copying node_modules, git history, test data.
- Pin base images and use slim variants.
- Run as non-root and set a minimal ENTRYPOINT/CMD.

Commands / snippets

```
# Example multi-stage Dockerfile sketch
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build && npm prune --omit=dev

FROM node:20-alpine AS runtime
WORKDIR /app
ENV NODE_ENV=production
COPY --from=build /app/dist ./dist
COPY --from=build /app/node_modules ./node_modules
USER node
CMD ["node", "dist/server.js"]
```

```
# BuildKit caching (CI)
DOCKER_BUILDKIT=1 docker build -t org/api:$GIT_COMMIT .
```

Common pitfalls

- Copying everything before dependency install (breaks build cache).
 - Leaving secrets in build context or layers.
 - Using unpinned 'latest' base images.
-

Q3. How do you handle image security and ECR management?

Interview answer: I enforce image security with scanning, signing, and least-privilege access to ECR. CI runs vulnerability scans (Trivy/Grype), blocks critical findings, and optionally signs images (cosign). In ECR, I enable scan on push, image tag immutability, lifecycle policies, and cross-account access via IAM roles.

Scenario: We reduced vulnerability exposure by failing builds on critical CVEs and rotating base images weekly. ECR lifecycle policies kept only 30 days of unpromoted tags, preventing storage growth.

Practical approach

- Scan in CI before push; scan again in registry for defense-in-depth.
- Sign images and verify signatures at deploy time (policy gate).
- Use immutable tags for release builds and enforce digest pinning in prod.
- Use ECR lifecycle policies and replication where needed.
- Lock ECR permissions to push/pull only for specific repos and roles.

Commands / snippets

```
# Trivy scan
trivy image --severity HIGH,CRITICAL --exit-code 1 org/api:$GIT_COMMIT

# ECR login
aws ecr get-login-password --region ap-south-1 | docker login --username AWS
--password-stdin <acct>.dkr.ecr.ap-south-1.amazonaws.com

# ECR lifecycle policy (idea)
aws ecr put-lifecycle-policy --repository-name api --lifecycle-policy-text
file://lifecycle.json

# Optional signing (cosign)
cosign sign --key cosign.key <acct>.dkr.ecr.ap-south-1.amazonaws.com/api@$DIGEST
```

Common pitfalls

- Allowing broad ECR permissions across all repositories.
 - Deploying by tag only (mutable) instead of digest pinning for prod.
 - No cleanup policies leading to registry storage bloat.
-

Kubernetes

4 interview questions with suggested answers, scenarios, and practical steps.

Q1. Describe a real production issue you handled.

Interview answer: I handled an incident where users saw intermittent 502s via Ingress during peak traffic. The root cause was readiness probe flapping under CPU throttling and an upstream timeout mismatch. Fixes included right-sizing requests/limits, increasing HPA min replicas, and tuning ingress timeouts while optimizing a slow DB query.

Scenario: During a marketing campaign, traffic spiked. NGINX Ingress showed upstream timed out. Pods were running but readiness failed repeatedly. After scaling and tuning, error rate dropped and latency stabilized.

Practical approach

- Start at the edge: ingress/controller logs and error codes; correlate timestamps.
- Validate service endpoints and pod readiness; check for flapping readiness/liveness.
- Check resource pressure: CPU throttling, memory pressure, node saturation.
- Inspect upstream timeouts and retries (ingress annotations, app client timeouts).
- Mitigate: scale replicas, adjust HPA, tune timeouts, and address slow dependency.

Commands / snippets

```
kubectl logs -n ingress-nginx deploy/ingress-nginx-controller --since=15m | grep '502'
kubectl describe ingress api -n prod
kubectl get svc,Endpoints -n prod api -o wide
kubectl get pods -n prod -l app=api -w
kubectl top pods -n prod
kubectl describe hpa api -n prod

# Example timeout tune (NGINX Ingress)
kubectl annotate ingress api -n prod
  nginx.ingress.kubernetes.io/proxy-read-timeout='120' --overwrite
```

Common pitfalls

- Fixing symptoms only (timeouts) without addressing CPU throttling or dependency latency.
- Scaling app tier without considering DB connection pool limits.
- Readiness endpoints that are too strict and flap under load.

Q2. Deployment vs StatefulSet - when and why?

Interview answer: Use Deployment for stateless workloads where pods are interchangeable. Use StatefulSet for stateful workloads needing stable identity, ordered rollout, and stable storage (PVC per pod). StatefulSets also work with headless Services for stable DNS per replica, which is important for databases, message brokers, and clustered systems.

Scenario: We ran stateless APIs as Deployments behind a Service. For Kafka and a stateful cache cluster, we used StatefulSets to guarantee stable pod names and persistent volumes.

Practical approach

- Ask: does each replica need a stable identity and storage? If yes, consider StatefulSet.
- If traffic can go to any replica and state is externalized (DB/cache), use Deployment.
- For StatefulSet, design storage class, backup/restore, PDBs, and careful rolling updates.
- Avoid StatefulSet for stateless apps; it adds complexity with little benefit.

Commands / snippets

```
# Quick comparison
kubectl get deploy,statefulset -n prod
kubectl describe statefulset kafka -n prod
kubectl get pvc -n prod | grep kafka
kubectl get pod -n prod kafka-0 -o jsonpath='{.metadata.name}{.spec.nodeName}{"\n"}'
```

Common pitfalls

- Running a database as a Deployment (no stable identity, unsafe storage assumptions).
- StatefulSet without backups and disaster recovery plan.
- Updating StatefulSet images without considering ordered rollout and quorum.

Q3. How do you troubleshoot CrashLoopBackOff?

Interview answer: I troubleshoot CrashLoopBackOff by focusing on the first failure signal: events and previous logs. I check init containers, env/config/secrets, mounts, probes, and dependency connectivity. If needed, I use kubectl debug to enter the pod namespace and validate DNS/network. Then I fix config or image and monitor rollout.

Scenario: A pod crashed during init because the DB secret was missing. describe showed secret mount error; previous logs confirmed connection string was empty. Fixing ExternalSecret mapping resolved the crash.

Practical approach

- kubectl describe pod: look at Events for OOMKilled, image pull, mount errors, probe failures.
- kubectl logs --previous: capture the crash reason from the previous container run.
- Check init containers and startup probe logic.
- Validate config maps/secrets and environment variables.
- Use kubectl debug or ephemeral containers to test DNS and connectivity.

Commands / snippets

```
kubectl get pods -n app
kubectl describe pod <pod> -n app
kubectl logs <pod> -n app --previous
kubectl logs <pod> -n app -c <init-container> --previous
kubectl get cm,secret -n app

# Ephemeral debug (if supported)
kubectl debug -it pod/<pod> -n app --image=busybox:1.36 --target=<container>
nslookup postgres.app.svc.cluster.local
nc -vz postgres.app.svc.cluster.local 5432
```

Common pitfalls

- Ignoring --previous logs and only reading current logs.
 - Probe configuration that restarts healthy pods.
 - Trying many changes at once and losing the root cause.
-

Q4. How do you design HA and autoscaling?

Interview answer: I design HA by removing single points of failure: multi-AZ node groups, multiple replicas across nodes, PDBs, proper requests/limits, and resilient dependencies (managed DB, multi-AZ). Autoscaling uses HPA (CPU/memory/custom metrics), cluster autoscaler or Karpenter, and safe rollout strategies. I also use topology spread constraints and pod anti-affinity to avoid replicas on one node or AZ.

Scenario: For a critical API, we ran 6 replicas, spread across 3 AZs, used PDB to keep at least 4 available, and HPA based on CPU and request rate. Cluster autoscaler ensured node capacity during spikes.

Practical approach

- Set requests/limits and readiness probes correctly so the scheduler and HPA behave predictably.
- Use at least 2-3 replicas, PDB, and anti-affinity/topology spread to distribute risk.
- Configure HPA with sensible min/max and stabilization windows to avoid thrash.
- Enable cluster autoscaler/Karpenter to add nodes when pods cannot schedule.
- Test failure modes: drain a node, kill an AZ, and verify service remains healthy.

Commands / snippets

```
kubectl get hpa -n prod
kubectl describe hpa api -n prod
kubectl get pdb -n prod
kubectl describe deploy api -n prod | egrep -i 'strategy|replicas|topology|affinity'
kubectl top pods -n prod
kubectl top nodes
```

Common pitfalls

- Autoscaling on CPU only when the bottleneck is DB or external API latency.
 - No PDB or anti-affinity leading to correlated failures.
-

AWS & Architecture

4 interview questions with suggested answers, scenarios, and practical steps.

Q1. How do you design highly available systems?

Interview answer: I design HA by assuming components fail and building redundancy: multi-AZ for compute and data, load balancing, stateless services, decoupling with queues, caching, and automated recovery. I define SLOs and design to meet them: health checks, autoscaling, timeouts, retries with backoff, and graceful degradation.

Scenario: A web platform used ALB + Auto Scaling across 3 AZs, RDS Multi-AZ for relational data, ElastiCache for caching, and SQS for async work. Single-AZ failures did not cause outages.

Practical approach

- Start with requirements: availability target, RTO/RPO, and user journey critical paths.
- Use multi-AZ for everything in-region: subnets, ASG/EKS node groups, managed DB Multi-AZ.
- Eliminate state from instances; store state in managed services (RDS/DynamoDB/S3).
- Use load balancers + health checks and deploy with rolling/canary strategies.
- Add backpressure and circuit breakers: timeouts, retries with jitter, and queue-based smoothing.

Commands / snippets

```
# AWS quick checks (examples)
aws elbv2 describe-load-balancers
aws autoscaling describe-auto-scaling-groups
aws rds describe-db-instances --query 'DBInstances[].MultiAZ'
aws ec2 describe-subnets --filters
    Name=availability-zone,Values=ap-south-1a,ap-south-1b

# Route53 health checks (multi-region patterns)
aws route53 list-health-checks
```

Common pitfalls

- Single AZ databases or single NAT gateway used by critical paths.
 - Retries without backoff causing retry storms.
 - No capacity headroom for failover events.
-

Q2. Multi-AZ vs multi-region - when to use each?

Interview answer: Multi-AZ protects against AZ failures and is the default for HA inside one region. Multi-region protects against regional outages and can reduce latency for global users, but adds complexity: data replication, consistency, failover orchestration, and higher cost. I use multi-region when business requirements demand it (regulatory, global latency, very high availability).

Scenario: For most services we used multi-AZ in one region. For a customer-facing service with strict availability requirements, we added a second region with warm standby and Route53 failover.

Practical approach

- Use multi-AZ by default for compute and managed data stores.
- Choose multi-region if you need regional disaster tolerance, global latency, or data residency constraints.
- Define RTO/RPO and pick DR strategy: backup/restore, pilot light, warm standby, active-active.
- Design data replication carefully (DynamoDB global tables, Aurora global DB, async replication).
- Test failover regularly (game days) and automate runbooks.

Commands / snippets

```
# Route53 failover policy setup is typically IaC, but validate like:
aws route53 list-hosted-zones
aws route53 list-resource-record-sets --hosted-zone-id <zone-id>

# Cross-region replication examples
aws s3api get-bucket-replication --bucket <bucket>
aws dynamodb describe-table --table-name <table> | grep -i replica
```

Common pitfalls

- Multi-region without tested failover (false confidence).
- Assuming synchronous replication across regions (latency and cost).
- Not aligning RTO/RPO with the chosen DR architecture.

Q3. How do you enforce least-privilege IAM?

Interview answer: Least privilege means granting only the actions and resources required, with conditions, and using roles instead of long-lived keys. I enforce it with scoped IAM policies, permission boundaries, SCPs (Organizations), session tags, and continuous monitoring via CloudTrail and IAM Access Analyzer. I also separate human vs workload roles and use short-lived credentials (STS).

Scenario: We replaced shared AWS keys in CI with assume-role using OIDC, then restricted policies to specific ECR repos and EKS namespaces. Access Analyzer and CloudTrail helped confirm no over-broad permissions remained.

Practical approach

- Use IAM roles and STS: no static keys for workloads if possible (OIDC for CI).
- Scope resource ARNs, limit regions, and add conditions (aws:PrincipalTag, aws:RequestedRegion).
- Separate duties: deployer role vs read-only role vs break-glass admin role.
- Use permission boundaries for developers and SCPs to block dangerous actions.
- Continuously review with Access Analyzer and CloudTrail.

Commands / snippets

```
# Access Analyzer and policy simulation
aws accessanalyzer list-analyzers
aws accessanalyzer list-findings --analyzer-arn <arn>

aws iam simulate-principal-policy --policy-source-arn
arn:aws:iam::<acct>:role/jenkins-deployer --action-names ecr:PutImage
ecr:BatchCheckLayerAvailability --resource-arns
arn:aws:ecr:ap-south-1:<acct>:repository/api

# CloudTrail lookup (who did what)
aws cloudtrail lookup-events --lookup-attributes
AttributeKey=EventName,AttributeValue=AssumeRole --max-results 20
```

Common pitfalls

- Wildcards like Action:* or Resource:* for convenience.
- Long-lived access keys in CI/CD.
- No monitoring, so privilege creep goes unnoticed.

Q4. How do you design disaster recovery and optimize cost?

Interview answer: I design DR by defining RTO/RPO per system, choosing an appropriate pattern (backup/restore, pilot light, warm standby, active-active), and automating recovery with IaC and runbooks. For cost, I right-size, use autoscaling, choose managed services wisely, set retention policies, and use Savings Plans/Reserved Instances/Spot where appropriate. Cost optimization should not compromise reliability targets.

Scenario: For a medium-critical service, we used warm standby in a second region: replicated S3, periodic DB snapshots + restore automation, and minimal compute always on. This met RTO 30 minutes and controlled spend.

Practical approach

- Classify services by criticality and set RTO/RPO targets.
- Pick DR pattern per service; document failover and failback procedures.
- Automate backups: RDS snapshots, DynamoDB PITR, EBS snapshots, S3 versioning/replication.
- Run regular DR tests (quarterly) and measure actual RTO/RPO.
- Cost controls: budgets/alerts, rightsizing, turning off non-prod, log retention, and lifecycle policies.

Commands / snippets

```
# Backup/DR checks (examples)
aws rds describe-db-snapshots --db-instance-identifier prod-db --max-items 5
aws dynamodb describe-continuous-backups --table-name prod-table
aws s3api get-bucket-versioning --bucket <bucket>
aws s3api get-bucket-lifecycle-configuration --bucket <bucket>
```

```
# Cost controls
aws budgets describe-budgets --account-id <acct>
aws ce get-cost-and-usage --time-period Start=2026-02-01,End=2026-02-28
  --granularity MONTHLY --metrics UnblendedCost
```

Common pitfalls

- DR plan not tested - backups exist but restores fail when needed.
 - No retention policies for logs, snapshots, and images.
-

Interview cheat sheet (quick frameworks)

Use these short structures to keep answers crisp under pressure.

Production incident answer template

- Symptom: what users saw (errors/latency) and where (region/path).
- Scope: confirm blast radius with metrics; identify if edge, app, or dependency.
- Hypothesis: top 2 likely causes based on signals (events, throttling, timeouts).
- Mitigation: safest fastest change to restore service (rollback, scale, toggle).
- Verification: show recovery via user-facing metrics; document follow-ups.

Command selection rule of thumb

- Kubernetes: describe pod + previous logs + events + top pods.
- Ingress issues: controller logs + endpoints + readiness flaps.
- AWS: health of ALB/ASG + DB Multi-AZ + CloudTrail for recent changes.
- Jenkins: build logs + artifact digest + deployment history (Helm/rollout).

Practice: pick one question per category and answer it in 60 seconds, then expand with one real example and 2 commands.