

DevOps Interview Answers

Kubernetes, Docker, Terraform, and GitHub Actions

Interview-ready responses with real-world scenarios and practical commands

Prepared as a study guide (customize versions, cloud/provider names, and tooling to your environment before interview).

Section	Questions	Focus Areas
Kubernetes (K8s)	10	Ops incidents, debugging, Helm, multi-tenancy, zero downtime
Docker	5	Builds, cleanup, Compose, monitoring, multi-stage Dockerfile
Terraform	4	Versioning, multi-region, state scaling, DB safety
GitHub Actions	5	Reusable workflows, concurrency, failure handling

Tip: In interviews, replace sample versions, SLOs, namespaces, and cloud services with the exact values from your project.

Quick Index

Kubernetes (K8s) - 10 questions

- 1. What is the current version of K8s you are using in your project?
- 2. What was the last production issue you faced and how did you resolve it?
- 3. A pod is stuck in CrashLoopBackOff. Logs show failure during initialization - how do you troubleshoot?
- 4. How do you enforce tenant isolation in a multi-tenant Kubernetes setup?
- 5. During high traffic, your app shows intermittent 502 errors through Ingress - how do you debug?
- 6. How do you prevent bad configs from reaching production in a CI/CD pipeline?
- 7. How would you ensure zero-downtime deployment during a critical update?
- 8. Helm deployment fails due to insufficient cluster resources - what is your approach?
- 9. How do you share Helm charts internally?
- 10. What is Helm chart testing and how is it done?

Docker - 5 questions

- 1. How would you manage Docker workloads across multiple clouds?
- 2. How do you handle image cleanup to prevent disk space issues?
- 3. How do you manage multi-container dependencies using Docker Compose?
- 4. How do you monitor container performance in production?
- 5. Wrote a multi-stage Dockerfile during screen sharing.

Terraform - 4 questions

- 1. How do you manage Terraform provider versioning?
- 2. How would you provision infra across 10 AWS regions simultaneously?
- 3. What to do when your Terraform state file becomes too large?
- 4. Terraform plan shows destroy + recreate for a critical DB - how to prevent downtime?

GitHub Actions - 5 questions

- 1. How do you reuse workflows across repositories?
- 2. How to manage large workflow files efficiently?
- 3. What's the difference between public and private workflow repositories?
- 4. How to implement workflow concurrency?
- 5. How do you handle failed workflows?

Kubernetes (K8s)

10 interview questions with suggested answers, scenarios, and command examples.

Q1. What is the current version of K8s you are using in your project?

Interview-ready answer: In my current project, we run Kubernetes v1.29.x in production (EKS) and validate upgrades on v1.30.x in staging before rollout. We keep control plane and node groups within supported skew, track deprecations every release, and run upgrade rehearsals with Helm chart compatibility checks. I always mention the exact version from the live cluster during interviews if available.

Scenario example:

Scenario: During an audit, the team needed proof that the cluster was on a supported version and that APIs used by workloads were not deprecated. I exported the cluster version, scanned manifests for deprecated APIs, and created an upgrade plan with a canary node group.

Practical commands / snippets:

```
# Check cluster and node versions
kubectl version --short
kubectl get nodes -o wide
kubectl get nodes -o jsonpath='{range
.items[*]}{.metadata.name}{"\t"}{.status.nodeInfo.kubeletVersion}{"\n"}}{end}'

# Check for deprecated API usage (examples)
kubectl api-resources | grep -i ingress
helm list -A
kubectl get ingress -A -o yaml | grep 'apiVersion:' -n

# EKS example (adjust cluster name/region)
aws eks describe-cluster --name prod-cluster --region ap-south-1 --query
'cluster.version' --output text
```

Q2. What was the last production issue you faced and how did you resolve it?

Interview-ready answer: One recent production issue was a sudden spike in 5xx errors caused by a bad application config deployed with a feature flag set incorrectly. I confirmed the blast radius via metrics and logs, rolled back the Helm release, and then added a config validation step in CI plus canary rollout gates to prevent recurrence.

Scenario example:

Scenario: After a release, error rate jumped from 0.2% to 8% on one service. Pods were healthy, but app logs showed invalid downstream endpoint settings from ConfigMap. Rollback fixed traffic in under 5 minutes, then we hotfixed and redeployed with approval.

Practical commands / snippets:

```
# Confirm issue
kubectl get pods -n payments
kubectl logs -n payments deploy/api --since=10m | tail -100
kubectl describe ingress -n payments public-ing

# Check recent rollout history
kubectl rollout history deploy/api -n payments
helm history api -n payments

# Roll back quickly
kubectl rollout undo deploy/api -n payments
# or if deployed by Helm
helm rollback api 12 -n payments

# Verify recovery
kubectl rollout status deploy/api -n payments --timeout=120s
kubectl get events -n payments --sort-by=.lastTimestamp | tail -20
```

Q3. A pod is stuck in CrashLoopBackOff. Logs show failure during initialization - how do you troubleshoot?

Interview-ready answer: I start by separating app failures from platform failures. For CrashLoopBackOff, I inspect the pod events, previous container logs, init containers, env/config, secrets, mounts, and startup probes. If needed, I run an ephemeral debug container and validate network/DNS dependencies. Then I patch the deployment or config and monitor rollout status.

Scenario example:

Scenario: A migration-enabled app crashed during startup because the DB hostname secret was mounted from the wrong namespace. The main container exited immediately. I used previous logs plus describe output, confirmed secret mismatch, corrected ExternalSecret mapping, and restarted the deployment.

Practical commands / snippets:

```
# Identify crash details
kubectl get pods -n app
kubectl describe pod api-7f8b9c9d6f-abcde -n app
kubectl logs api-7f8b9c9d6f-abcde -n app --previous
kubectl logs api-7f8b9c9d6f-abcde -n app -c init-db-check --previous

# Check config, secrets, and mounts
kubectl get deploy api -n app -o yaml | less
kubectl get cm,secret -n app
kubectl exec -it deploy/api -n app -- printenv | sort # if container stays up
kubectl get events -n app --sort-by=.metadata.creationTimestamp | tail -50

# Ephemeral debugging (K8s 1.25+)
kubectl debug -it pod/api-7f8b9c9d6f-abcde -n app --image=busybox:1.36 --target=api
nslookup postgres.app.svc.cluster.local
nc -vz postgres.app.svc.cluster.local 5432
```

Q4. How do you enforce tenant isolation in a multi-tenant Kubernetes setup?

Interview-ready answer: I enforce tenant isolation at multiple layers: separate namespaces per tenant, strict RBAC, NetworkPolicies default deny, ResourceQuota and LimitRange, Pod Security Admission labels, dedicated service accounts, and policy-as-code (Kyverno/Gatekeeper). For stronger isolation, I use node pools/taints for high-risk tenants and separate clusters for regulated workloads.

Scenario example:

Scenario: We hosted multiple internal teams in one platform cluster. A tenant accidentally created noisy workloads that impacted others. I added quotas and default limits, enforced network isolation, and restricted cross-namespace secret access via RBAC. After that, noisy tenants were contained and audit posture improved.

Practical commands / snippets:

```
# Namespace + quota + limits
kubectl create ns tenant-a
kubectl apply -f quota-tenant-a.yaml
kubectl apply -f limitrange-tenant-a.yaml

# RBAC example
kubectl -n tenant-a create serviceaccount tenant-a-ci
kubectl apply -f role-tenant-a.yaml
kubectl apply -f rolebinding-tenant-a.yaml

# NetworkPolicy (default deny + allow needed traffic)
kubectl apply -n tenant-a -f netpol-default-deny.yaml
kubectl apply -n tenant-a -f netpol-allow-egress-dns.yaml
kubectl apply -n tenant-a -f netpol-allow-ingress-from-ingress.yaml

# Pod Security Admission labels (namespace level)
kubectl label ns tenant-a pod-security.kubernetes.io/enforce=restricted --overwrite
```

Q5. During high traffic, your app shows intermittent 502 errors through Ingress - how do you debug?

Interview-ready answer: I debug 502s from the edge inward: Ingress controller logs and metrics, upstream service endpoints, pod readiness, connection saturation, timeouts, and app logs. Under high traffic, common causes are insufficient replicas, slow upstreams, readiness flapping, keepalive/timeouts mismatch, or node/network bottlenecks. I correlate timestamps across Ingress, service, and application metrics before tuning.

Scenario example:

Scenario: We saw intermittent 502s only during traffic peaks. NGINX Ingress logs showed upstream timed out; app pods had CPU throttling and readiness probe failures. I increased HPA min replicas, tuned requests/limits, extended upstream read timeout, and optimized a slow DB query. 502 rate dropped to near zero.

Practical commands / snippets:

```
# Ingress/controller logs
kubectl logs -n ingress-nginx deploy/ingress-nginx-controller --since=15m | grep ' 502'

kubectl describe ingress public-api -n app

# Service endpoints and pod readiness
kubectl get svc,Endpoints -n app api -o wide
kubectl get pods -n app -l app=api -w
kubectl describe pod -n app <pod-name> | egrep 'Readiness|Liveness|Restart'

# Resource pressure / HPA
kubectl top pods -n app
kubectl top nodes
kubectl get hpa -n app
kubectl describe hpa api -n app

# NGINX timeout annotations (example)
kubectl annotate ingress public-api -n app
nginx.ingress.kubernetes.io/proxy-read-timeout='120' --overwrite
kubectl annotate ingress public-api -n app
nginx.ingress.kubernetes.io/proxy-send-timeout='120' --overwrite
```

Q6. How do you prevent bad configs from reaching production in a CI/CD pipeline?

Interview-ready answer: I use layered controls: schema validation, linting, render checks, policy-as-code, environment-specific tests, and progressive delivery. For Kubernetes/Helm configs I run helm lint, helm template, kubeconform/kubeval, conftest or Kyverno policy checks, secret scanning, and dry-run apply against a test API server before deployment. Production deploys require protected branches and approvals.

Scenario example:

Scenario: A malformed values.yaml once disabled TLS by mistake in staging. To prevent this in production, I added JSON schema validation for Helm values, OPA policy checks that block insecure Ingress, and a mandatory preview environment smoke test before merge.

Practical commands / snippets:

```
# Helm validation
helm lint charts/api
helm template api charts/api -f values-prod.yaml > rendered.yaml

# Kubernetes manifest validation
kubeconform -strict -summary rendered.yaml
kubectl apply --dry-run=server -f rendered.yaml

# Policy checks (OPA Conftest example)
conftest test rendered.yaml --policy policy/

# Secret scanning and YAML linting (examples)
yamllint .
gitLeaks detect --source .
```

Q7. How would you ensure zero-downtime deployment during a critical update?

Interview-ready answer: I combine deployment strategy, readiness discipline, and rollback readiness. The baseline is rolling updates with maxUnavailable=0, solid readiness probes, PodDisruptionBudgets, graceful termination, preStop hooks, and capacity headroom. For risky releases I use canary or blue/green with traffic shifting and automated rollback on SLO breach.

Scenario example:

Scenario: We deployed a critical payments API during business hours. I pre-scaled replicas, set maxSurge/maxUnavailable for no capacity loss, validated readiness endpoints, and monitored latency/error SLOs. We released canary first, then progressively increased traffic. No customer-visible downtime occurred.

Practical commands / snippets:

```
# Deployment strategy (example patch)
kubectl patch deploy api -n payments -p '{
"spec":{"strategy":{"type":"RollingUpdate","rollingUpdate":{"maxSurge":"25%","maxUnavailable":0}}}
}'

# Monitor rollout
kubectl rollout status deploy/api -n payments --timeout=300s
kubectl get rs -n payments -l app=api
kubectl get pods -n payments -l app=api -w

# Fast rollback if SLO degrades
kubectl rollout undo deploy/api -n payments
# Helm equivalent
helm upgrade --install api charts/api -n payments -f values-prod.yaml --atomic
--timeout 10m
```

Q8. Helm deployment fails due to insufficient cluster resources - what is your approach?

Interview-ready answer: First I confirm whether the failure is scheduling-related (CPU/memory, PVC, node selectors, taints) or quota-related (namespace quota/limits). I inspect events and pending pods, compare requests/limits against actual cluster capacity, then either right-size resources, scale node groups, or split rollout. I avoid blindly lowering requests if it compromises SLOs.

Scenario example:

Scenario: A new release increased memory requests and pods stayed Pending. Describe output showed insufficient memory on nodes and a namespace quota ceiling. I temporarily scaled the node group, adjusted HPA min/max, and optimized memory settings for sidecars before rerunning Helm upgrade.

Practical commands / snippets:

```
# Helm failure details
helm upgrade --install api charts/api -n app -f values-prod.yaml --debug --dry-run
helm status api -n app

# Inspect scheduling / quota issues
kubectl get pods -n app
kubectl describe pod -n app <pending-pod>
kubectl get events -n app --sort-by=.lastTimestamp | tail -30
kubectl describe quota -n app
kubectl top nodes
kubectl describe nodes | egrep 'Allocated resources|cpu|memory' -A8

# Common fixes
kubectl scale deploy api -n app --replicas=3 # temporary right-size
# then re-run release
helm upgrade api charts/api -n app -f values-prod.yaml --atomic
```

Q9. How do you share Helm charts internally?

Interview-ready answer: I package charts and publish them to an internal OCI registry (for example ECR, GHCR, Harbor, or Artifactory) with semantic versions. Teams consume the charts via CI with pinned versions, and we maintain a changelog plus values schema. OCI-based sharing is simpler than legacy chart repositories and works well with existing registry auth.

Scenario example:

Scenario: Multiple teams needed a common app chart and a base ingress chart. We moved them to an internal Harbor OCI registry, enforced version tags, and added a chart release workflow. This reduced drift and made rollbacks predictable.

Practical commands / snippets:

```
# Package and push chart to OCI registry
helm dependency update charts/base-app
helm lint charts/base-app
helm package charts/base-app

helm registry login harbor.internal.example.com
helm push base-app-1.4.2.tgz oci://harbor.internal.example.com/helm

# Pull and use chart
helm pull oci://harbor.internal.example.com/helm/base-app --version 1.4.2
helm upgrade --install myapp oci://harbor.internal.example.com/helm/base-app \
-n myns -f values.yaml --create-namespace
```

Q10. What is Helm chart testing and how is it done?

Interview-ready answer: Helm chart testing validates chart quality before release. It includes linting, rendering, schema validation, unit tests for templates, and integration tests by installing the chart in a disposable cluster (kind/k3d) and running smoke checks. In CI, I use helm lint + ct (chart-testing) + optional helm-unittest plugin, then install into kind and verify readiness.

Scenario example:

Scenario: A chart change accidentally removed a required selector label, which would have broken Service routing. The template unit test and ct install step failed in CI before merge, so the issue never reached staging.

Practical commands / snippets:

```
# Lint and render
helm lint charts/api
helm template api charts/api -f charts/api/ci-values.yaml > rendered.yaml

# Unit tests (plugin)
helm plugin install https://github.com/helm-unittest/helm-unittest || true
helm unittest charts/api

# Chart Testing (ct)
ct lint --charts charts/api
ct install --charts charts/api

# Integration test in kind (example)
kind create cluster --name helm-ci
helm upgrade --install api charts/api -n test --create-namespace
kubectl rollout status deploy/api -n test --timeout=180s
kubectl get all -n test
```

Docker

5 interview questions with suggested answers, scenarios, and command examples.

Q1. How would you manage Docker workloads across multiple clouds?

Interview-ready answer: I avoid managing raw Docker hosts manually at scale. I standardize on immutable images, push them to a central registry strategy, and run workloads on managed orchestrators (usually Kubernetes on EKS/AKS/GKE). Deployment is driven by Terraform + GitHub Actions + GitOps, with environment parity, observability, and policy controls across clouds.

Scenario example:

Scenario: A platform needed active-active services in AWS and Azure. We built multi-arch images once, pushed to GHCR, and deployed to EKS and AKS with the same Helm chart and cloud-specific values. Terraform modules handled cloud differences while keeping app deployment consistent.

Practical commands / snippets:

```
# Build and push multi-arch image
docker buildx create --use --name multi || true
docker buildx build --platform linux/amd64,linux/arm64 \
  -t ghcr.io/org/api:1.8.0 -t ghcr.io/org/api:latest --push .

# Replicate/tag for cloud-specific registries if needed
docker pull ghcr.io/org/api:1.8.0
docker tag ghcr.io/org/api:1.8.0
<aws_account>.dkr.ecr.ap-south-1.amazonaws.com/api:1.8.0
docker push <aws_account>.dkr.ecr.ap-south-1.amazonaws.com/api:1.8.0
```

Q2. How do you handle image cleanup to prevent disk space issues?

Interview-ready answer: I combine preventive and reactive cleanup. Preventive: smaller images (multi-stage builds), tag discipline, and registry retention policies. Reactive: scheduled prune jobs on build agents/hosts, monitoring disk usage, and alerting before exhaustion. I am careful not to delete images used by running containers.

Scenario example:

Scenario: CI runners were failing builds due to disk-full errors. I added nightly docker system prune for self-hosted runners, enabled build cache cleanup, and moved large layers into reusable base images. Disk-related failures dropped significantly.

Practical commands / snippets:

```
# Inspect disk usage
docker system df
df -h

# Safe cleanup (unused data)
docker image prune -a -f
docker container prune -f
docker volume prune -f
docker builder prune -a -f

# One-shot cleanup (be careful on shared hosts)
docker system prune -a --volumes -f

# Example cron (nightly at 2:30 AM)
# 30 2 * * * /usr/bin/docker system prune -a --volumes -f >/var/log/docker-prune.log 2>&1
```

Q3. How do you manage multi-container dependencies using Docker Compose?

Interview-ready answer: In Docker Compose, I define explicit service dependencies, healthchecks, shared networks, and startup order. I do not rely only on `depends_on` for application readiness; I add health checks and app-level retry logic. Config and secrets are externalized, and persistent data uses named volumes.

Scenario example:

Scenario: An API container started before Postgres was ready, causing startup failures. I added a database healthcheck, `depends_on` with `service_healthy`, and retry logic in the app startup. Local developer onboarding became much smoother.

Practical commands / snippets:

```
# docker-compose.yml (snippet)
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 3s
      retries: 10

  api:
    build: .
    depends_on:
      db:
        condition: service_healthy
    environment:
      DATABASE_URL: postgres://postgres:secret@db:5432/app

# Lifecycle commands
docker compose up -d --build
docker compose ps
docker compose logs -f api db
docker compose down -v
```

Q4. How do you monitor container performance in production?

Interview-ready answer: I use layered observability: container/node metrics (CPU, memory, filesystem, throttling), application metrics (latency, error rate, throughput), logs, and traces. In Kubernetes, Prometheus + Grafana + Alertmanager are standard; for Docker hosts, I also use cAdvisor/node-exporter and central log shipping. I define SLO-based alerts, not just infrastructure thresholds.

Scenario example:

Scenario: Users reported slowness but pods were not crashing. Metrics showed CPU throttling and GC spikes under peak load. We raised CPU requests, tuned JVM memory flags, and added HPA based on CPU and request rate, which stabilized latency.

Practical commands / snippets:

```
# Quick checks on a Docker host
docker stats
docker inspect <container_id> --format '{{json .State}}' | jq
docker events --since 30m

# Prometheus queries (examples)
# container_cpu_usage_seconds_total
# container_memory_working_set_bytes
# container_cpu_cfs_throttled_seconds_total

# Kubernetes resource metrics
kubectl top pods -A
kubectl top nodes
```

Q5. Wrote a multi-stage Dockerfile during screen sharing.

Interview-ready answer: I would write a multi-stage Dockerfile to keep the final image small, secure, and reproducible. Build dependencies stay in the builder stage, and the runtime stage contains only the compiled app plus required runtime libs. I also run as non-root where possible and pin base images.

Scenario example:

Scenario: During an interview or live debugging session, I usually explain each stage while writing it: build stage, test stage (optional), runtime stage, and image size/security improvements. Then I build and run it to prove it works.

Practical commands / snippets:

```
# Example multi-stage Dockerfile for a Node.js app
FROM node:20-alpine AS deps
WORKDIR /app
COPY package*.json ./
RUN npm ci

FROM node:20-alpine AS build
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
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/package*.json ./
COPY --from=build /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]

# Build and test
docker build -t sample-api:multi .
docker run --rm -p 3000:3000 sample-api:multi
docker images | grep sample-api
```

Terraform

4 interview questions with suggested answers, scenarios, and command examples.

Q1. How do you manage Terraform provider versioning?

Interview-ready answer: I pin provider versions with version constraints in `required_providers`, commit the `.terraform.lock.hcl` file, and upgrade providers intentionally through a tested process. I also standardize Terraform and provider versions in CI to avoid drift between engineer laptops and pipelines.

Scenario example:

Scenario: A provider minor upgrade changed default behavior and caused unexpected plan noise. After that, we locked provider versions, committed the lock file, and introduced a scheduled upgrade branch with review and integration tests.

Practical commands / snippets:

```
terraform {
  required_version = ">= 1.6.0, < 1.9.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.50"
    }
  }
}

# Initialize and create lock file
terraform init
git add .terraform.lock.hcl

# Controlled upgrade
terraform init -upgrade
terraform providers
terraform version
```

Q2. How would you provision infra across 10 AWS regions simultaneously?

Interview-ready answer: I use a reusable module with aliased AWS providers per region and drive region deployments with `for_each` from a region map. For speed and safety, I run plans/applies in parallel at the pipeline layer per region/workspace, with state isolation per region. I avoid a single giant apply for all regions when blast radius is high.

Scenario example:

Scenario: We needed S3, CloudWatch, and Route53 health checks in 10 regions. I built a regional module, instantiated it with provider aliases, and ran applies in parallel jobs. Failures in one region did not block or corrupt the others because each region had isolated backend state.

Practical commands / snippets:

```
# providers.tf (conceptual)
provider "aws" { alias = "mumbai"    region = "ap-south-1" }
provider "aws" { alias = "frankfurt" region = "eu-central-1" }
# ...repeat or generate via code conventions/Terragrunt

# Example plan/apply per region (CI jobs run in parallel)
terraform workspace select ap-south-1 || terraform workspace new ap-south-1
terraform plan -var='aws_region=ap-south-1' -out=tfplan
terraform apply -auto-approve tfplan

# Parallel orchestration example (GNU parallel)
parallel -j 5 ::: \
  "terraform -chdir=envs/ap-south-1 apply -auto-approve" \
  "terraform -chdir=envs/eu-central-1 apply -auto-approve" \
  "terraform -chdir=envs/us-east-1 apply -auto-approve"
```

Q3. What to do when your Terraform state file becomes too large?

Interview-ready answer: Large state usually means too many resources in one state or too much churn. I split monolithic state into smaller domains (network, data, app platform, per-region/per-env), use remote backend with locking, and move resources gradually using `terraform state mv/import`. I also remove stale resources and avoid storing unnecessary computed outputs.

Scenario example:

Scenario: A single state file for shared networking plus app infra caused slow plans and lock contention. We split state by domain and environment, moved resources with state mv, and reduced plan time significantly while improving team parallelism.

Practical commands / snippets:

```
# Inspect state size and contents
ls -lh terraform.tfstate
terraform state list | wc -l
terraform state list | head -20

# Move resources to a new state (example workflow)
terraform state mv aws_security_group.app module.platform.aws_security_group.app

# Remove orphaned resources from state (does not destroy infra)
terraform state rm aws_instance.legacy

# Pull/push remote state carefully (for backup/migration)
terraform state pull > backup-$(date +%F).tfstate
```

Q4. Terraform plan shows destroy + recreate for a critical DB - how to prevent downtime?

Interview-ready answer: I stop and investigate before applying. For critical DBs, I verify which attribute forces replacement, check provider docs, and compare against intentional changes. I use lifecycle prevent_destroy, ignore_changes only when justified, staged changes, snapshots, and blue/green or replica promotion strategies. If replacement is unavoidable, I plan a migration with cutover, not a direct apply in business hours.

Scenario example:

Scenario: A parameter change on an RDS resource triggered replacement in plan. We used prevent_destroy so apply would fail safely, created a read replica / new instance with desired settings, synced data, switched traffic during a maintenance window, then cleaned up old resources after validation.

Practical commands / snippets:

```
# Review exact replacement reasons
terraform plan
terraform show -json plan.out | jq '.resource_changes[] | select(.change.actions | index("delete"))'

# Protect critical resources
resource "aws_db_instance" "main" {
  # ...
  lifecycle {
    prevent_destroy = true
  }
}

# Take backup / snapshot before any change (AWS example)
aws rds create-db-snapshot --db-instance-identifier prod-db --db-snapshot-identifier prod-db-prechange-$(date +%F)

# Targeted plan for safer review (use carefully)
terraform plan -target=aws_db_parameter_group.main
```

GitHub Actions

5 interview questions with suggested answers, scenarios, and command examples.

Q1. How do you reuse workflows across repositories?

Interview-ready answer: I use reusable workflows with `workflow_call` for standardized CI/CD logic and composite actions for smaller step bundles. Teams call centrally maintained workflows from a platform repo, passing inputs and secrets. This improves consistency, security hardening, and maintenance compared to duplicating YAML in every repo.

Scenario example:

Scenario: We had 30 repos with duplicated build-and-deploy YAML. I moved shared logic into an org platform repository as reusable workflows, then each repo consumed it with a small wrapper workflow. Updating one central workflow improved all repos quickly.

Practical commands / snippets:

```
# Caller workflow (repo A)
name: CI
on: [push, pull_request]
jobs:
  call-shared-ci:
    uses: org/platform-workflows/.github/workflows/ci.yml@v2
    with:
      run_tests: true
      node_version: '20'
      secrets: inherit

# Reusable workflow (platform repo)
on:
  workflow_call:
    inputs:
      run_tests:
        required: false
        type: boolean
        default: true
```

Q2. How to manage large workflow files efficiently?

Interview-ready answer: I reduce workflow size by splitting logic into reusable workflows and composite actions, use matrix strategy for repeated jobs, and keep environment-specific values in variables/secrets instead of hardcoding. I also standardize naming, add comments for non-obvious logic, and lint workflows in CI.

Scenario example:

Scenario: One monolithic workflow grew to 900+ lines and became hard to debug. I extracted build, security scan, and deploy phases into reusable pieces, then created a slim orchestrator workflow. Review time and failure triage improved a lot.

Practical commands / snippets:

```
# Lint workflow files
npm i -g actionlint
actionlint

# Example matrix to reduce duplication
strategy:
  matrix:
    service: [api, worker, cron]

# Call a composite action
- uses: ../.github/actions/setup-build-env
```

Q3. What's the difference between public and private workflow repositories?

Interview-ready answer: Public workflow repositories are easier to share broadly but must never contain sensitive defaults, internal URLs, or privileged assumptions. Private workflow repositories are restricted to authorized repos/org members and are better for internal deployment logic. In both cases, access to secrets is controlled by the caller workflow and repository/org settings, not by the reusable workflow file alone.

Scenario example:

Scenario: We kept generic lint/test reusable workflows in a public repo for open-source projects, but deployment and release workflows stayed in a private org repo because they referenced internal tooling and protected environments.

Practical commands / snippets:

```
# Example caller references
uses: org/public-workflows/.github/workflows/lint.yml@v1
# or internal/private
uses: org/platform-workflows/.github/workflows/deploy.yml@v3

# Check repo visibility via gh CLI
gh repo view org/platform-workflows --json name,visibility,defaultBranchRef
```

Q4. How to implement workflow concurrency?

Interview-ready answer: I use the concurrency key to prevent duplicate runs for the same branch, environment, or deployment target. For CI on pull requests, I usually cancel in-progress runs to save runners. For production deploys, I serialize by environment and often do not cancel in-progress if a deployment already started.

Scenario example:

Scenario: Multiple pushes to the same branch triggered overlapping deploys and caused race conditions. I added concurrency groups by app and environment, which ensured only one deployment to production ran at a time.

Practical commands / snippets:

```
# Branch-level CI concurrency
concurrency:
  group: ci-${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

# Environment-level deploy concurrency
concurrency:
  group: deploy-prod-api
  cancel-in-progress: false
```

Q5. How do you handle failed workflows?

Interview-ready answer: I treat failed workflows as either code failures, infra/transient failures, or pipeline design issues. First I inspect the failing job logs and artifacts, rerun failed jobs for transient issues, and correlate with external outages (registry, cloud API, runner). Then I fix root cause and add guardrails like retries, caching fixes, improved observability, or flaky-test quarantine if appropriate.

Scenario example:

Scenario: A deployment workflow failed intermittently during image push because of registry rate limits. I confirmed via logs, added authenticated pulls/pushes, exponential retry for the push step, and scheduled cleanup of stale tags. Failures dropped sharply.

Practical commands / snippets:

```
# List recent runs
gh run list --limit 20

# Inspect a specific run
gh run view <run-id> --log-failed
gh run download <run-id> --dir artifacts/

# Re-run failed jobs only
gh run rerun <run-id> --failed

# Enable debug logs (repo/org secrets/variables)
# ACTIONS_STEP_DEBUG=true
# ACTIONS_RUNNER_DEBUG=true
```