DevOps Interview Questions and Answers

# 1. Linux

* How do you limit CPU/memory usage for a process in Linux?

- cgroups (control groups):  
 \* Use systemd or cgcreate/cgset to set CPU/memory limits.  
 \* Example:  
 cgcreate -g memory,cpu:/mygroup  
 cgset -r memory.limit\_in\_bytes=1G mygroup  
 cgset -r cpu.shares=512 mygroup  
 cgexec -g memory,cpu:mygroup <command>  
- ulimit:  
 \* For per-process limits (shell):  
 ulimit -u 1000 # max user processes  
 ulimit -v 1048576 # max virtual memory (KB)  
- systemd:  
 \* In service files:  
 [Service]  
 MemoryLimit=1G  
 CPUQuota=50%

* What is the difference between soft mount and hard mount in NFS?

- Hard mount:  
 \* Default. Retries indefinitely if server is unreachable.  
 \* Good for critical data, but can hang processes.  
- Soft mount:  
 \* Fails after timeout/retries.  
 \* Faster failure, but risk of data corruption/loss.  
- Best practice: Use hard with intr/timeo for critical, soft for non-critical.

* How do you capture and analyze system calls of a process?

- strace: strace -p <pid> or strace <command>  
- ltrace: For library calls.  
- perf, bpftrace, sysdig: For advanced tracing and profiling.

* Your application is randomly freezing. How would you troubleshoot.

1. Check resource usage: top, htop, vmstat, iostat  
2. Check logs: Application, syslog, dmesg  
3. strace/lsof: Attach to process to see where it’s stuck (I/O, network, locks)  
4. Check for deadlocks: Thread dumps, pstack, gdb  
5. Check kernel messages: OOM, hardware errors  
6. Network issues: Packet loss, DNS, firewall

* A server with 64GB RAM is still swapping aggressively. How would you handle this?

- Check memory usage: free -m, vmstat, smem  
- Check for memory leaks: ps aux --sort -rss  
- Swappiness:  
 \* Lower vm.swappiness (default 60, try 10-20):  
 sysctl vm.swappiness=10  
- Check for tmpfs/ramdisk overuse  
- Check for zombie processes  
- If swap is needed: Add more RAM, optimize apps, tune caches.

# 2. AWS

* Your ECS tasks are crashing frequently. How would you troubleshoot?

1. Check task/container logs: CloudWatch, ECS console  
2. Check resource limits: CPU/memory reservation vs. usage  
3. Check image health: ImagePull errors, outdated images  
4. Check IAM roles: Permissions for resources  
5. Check networking: Security groups, ENI, VPC/subnets  
6. Check service events: ECS events tab

* How would you secure ECS tasks that need access to S3 and DynamoDB?

- Use IAM roles for tasks:  
 \* Assign least-privilege role to ECS task definition.  
 \* Avoid using instance roles or static credentials.  
- VPC endpoints:  
 \* Use S3/DynamoDB endpoints to avoid public internet.  
- Encrypt data in transit and at rest.

* How would you ensure compliance and auditability for RDS access?

- Enable RDS logging: Audit, general, slow query logs.  
- CloudTrail: Track API calls.  
- IAM authentication: Use IAM for DB access.  
- Encryption: Enable at rest and in transit.  
- VPC: Restrict access via security groups.

* Explain a scenario where VPC peering is more appropriate than Transit Gateway.

- VPC Peering:  
 \* Simple, direct, low-latency, for few VPCs.  
 \* No transitive routing.  
- Transit Gateway:  
 \* Scalable, hub-and-spoke, transitive routing, multi-account.  
 \* Use for large/multi-region/multi-account networks.

* Your organization needs to grant cross-account access to a partner. How would you implement it securely?

- IAM Roles with External ID:  
 \* Create role in your account, allow partner to assume with external ID.  
 \* Restrict permissions, monitor with CloudTrail.  
- Resource policies:  
 \* For S3, KMS, etc., use bucket/key policies with principal.

* Your application has predictable daily peaks but also unexpected spikes. How would you combine scheduled and dynamic scaling?

- Scheduled scaling:  
 \* Set scaling actions for known peaks (e.g., 9am-6pm).  
- Dynamic scaling:  
 \* Use CloudWatch alarms (CPU, queue length, custom metrics).  
- Combine:  
 \* Scheduled for baseline, dynamic for unexpected spikes.

# 3. Docker

* How do you secure Docker against kernel privilege escalation exploits?

- Run as non-root user in containers  
- Drop capabilities: --cap-drop=ALL and add only needed ones.  
- Use seccomp, AppArmor, SELinux profiles  
- Keep Docker and kernel updated  
- Disable privileged mode

* You want to run thousands of containers per host. What Linux tuning do you apply?

- Increase ulimits: File descriptors, processes, network connections.  
- Tune kernel params:  
 \* /etc/sysctl.conf:  
 fs.file-max, vm.max\_map\_count, net.core.somaxconn, net.ipv4.ip\_local\_port\_range  
- Storage driver: Use overlay2, optimize disk I/O.  
- Monitor CPU/memory pressure

* You must ensure data at rest encryption for container volumes. How?

- Use encrypted storage backend:  
 \* LUKS, dm-crypt, EBS encryption, Azure Disk encryption.  
- Docker volume plugins:  
 \* e.g., Portworx, StorageOS with encryption.  
- Kubernetes: Use CSI drivers with encryption.

* How do you handle multi-cloud Docker deployments with compliance restrictions?

- Use orchestrators: Kubernetes, ECS, Nomad.  
- Centralized secrets management: HashiCorp Vault, AWS Secrets Manager.  
- Image scanning: Trivy, Clair, AWS ECR scan.  
- Network policies: Calico, Cilium.  
- Compliance: CIS Docker/K8s Benchmarks, audit logging.

* Write a complete multi-stage Dockerfile from scratch.

Example:  
FROM golang:1.21 AS builder  
WORKDIR /app  
COPY . .  
RUN go build -o myapp  
  
FROM alpine:3.18  
WORKDIR /app  
COPY --from=builder /app/myapp .  
CMD ["./myapp"]

# 4. Kubernetes (K8s)

* The Kubernetes control plane becomes completely unreachable. How do you recover the cluster?

- Check etcd, API server, networking  
- Access master nodes via SSH  
- Restore etcd from backup if needed  
- Check certificates, kubelet status  
- Review cloud provider control plane health

* etcd data corruption occurs, and your cluster won’t start. What’s your recovery plan?

- Restore etcd from latest backup  
- If no backup: Try etcdctl snapshot restore if partial data available.  
- Document and automate regular etcd backups!

* A Pod is stuck in ImagePullBackOff. How do you troubleshoot?

- Check image name/tag  
- Check registry credentials  
- Check network/DNS  
- Check image exists in registry  
- Describe pod: kubectl describe pod <pod>

* During high traffic, Ingress controller becomes a bottleneck. How do you fix this?

- Scale ingress controller pods  
- Use LoadBalancer service in front  
- Enable keepalives, tune timeouts  
- Use multiple ingress controllers  
- Optimize backend app performance

* A StatefulSet upgrade failed and left half the pods in crashloop. How do you roll back safely?

- Rollback StatefulSet to previous version  
- Check PVCs/data consistency  
- Use kubectl rollout undo statefulset/<name>  
- Manually delete/recreate pods if needed

# 5. Jenkins

* How do you reduce Jenkins pipeline execution time in large projects?

- Parallelize stages  
- Use agent pools/executors  
- Cache dependencies/artifacts  
- Use lightweight checkout  
- Optimize test suites

* How do you manage pipelines for multiple microservices with interdependencies?

- Use shared libraries  
- Pipeline as code (Jenkinsfile per repo)  
- Trigger downstream/upstream jobs  
- Use dependency graph plugins

# 6. Terraform

* Rotate secrets used by Terraform-managed resources periodically. How?

- Use external secret managers: Vault, AWS Secrets Manager.  
- Automate rotation: Use terraform taint or apply after secret change.  
- Avoid storing secrets in state files.

* How to destroy only a specific resource safely?

- terraform destroy -target=resource.type.name

* Difference between count and for\_each.

- count: Integer, creates N identical resources.  
- for\_each: Map/set, creates resources per key/value, more flexible.

* How to handle Terraform provider versioning?

- Specify version in required\_providers block.  
- Use version constraints (>=, ~>, etc.)  
- Pin versions in CI/CD.

* Terraform keeps recreating a resource unnecessarily. How do you debug it?

- Check for drift: Manual changes outside Terraform.  
- Check for computed values: Dynamic fields in config.  
- Check provider bugs or resource dependencies.  
- Use terraform plan and terraform state show.

* How do you test Terraform code before production?

- Use terraform plan and terraform validate  
- Use Terratest, kitchen-terraform, or checkov for automated tests  
- Use separate staging environment

# 7. SRE

* How would you handle a high-severity outage in production?

- Incident response:  
 \* Declare incident, assemble team, assign roles.  
 \* Communicate status, update stakeholders.  
 \* Mitigate impact, restore service, root cause later.  
 \* Document timeline.

* Hot to calculate error budgets? and Difference between SLI vs SLO vs SLA.

- Error budget:  
 \* Error Budget = 1 - SLO  
 \* If SLO is 99.9%, error budget is 0.1%.  
- SLI: Service Level Indicator (measured metric, e.g., latency < 100ms)  
- SLO: Service Level Objective (target, e.g., 99.9% uptime)  
- SLA: Service Level Agreement (contractual, with penalties)

* How do you reduce toil in incident response?

- Automate repetitive tasks  
- Improve runbooks  
- Use chatops/bots  
- Invest in self-healing systems

* How would you perform capacity planning for a rapidly growing service?

- Monitor usage trends  
- Forecast growth  
- Test with load testing  
- Plan for headroom (buffer)  
- Automate scaling

* How do you implement blameless postmortems?

- Focus on process, not people  
- Document facts, timeline, impact  
- Identify systemic issues  
- Share learnings, track action items

# DevOps Interview Q&A

## Pipeline Running Very Slow: Troubleshooting Approach

* Check Pipeline Stages: Identify which stage(s) are slow (build, test, deploy, etc.).
* Resource Utilization: Examine runner/executor CPU, memory, and disk I/O usage.
* Job Logs: Review logs for bottlenecks (e.g., long dependency downloads, slow tests).
* Parallelization: Are jobs running sequentially that could be parallelized?
* Artifacts & Caching: Check if large artifacts are being uploaded/downloaded unnecessarily.
* External Dependencies: Are there slow external calls (e.g., Docker pulls, API calls)?
* Network Latency: Especially if using cloud runners or accessing remote resources.
* Runner Location: Are runners close to your source and artifact repositories?
* Recent Changes: Did the slowness start after a recent pipeline or infra change?
* Tools: GitLab CI/CD metrics, runner monitoring, custom logging, and profiling.

## Pipeline Failing Inconsistently: Troubleshooting Approach

* Flaky Tests: Identify if failures are due to non-deterministic tests.
* Resource Contention: Are jobs competing for limited resources?
* External Dependencies: Are failures due to unstable external services or APIs?
* Environment Differences: Are jobs running in different environments or with different variables?
* Logs & Artifacts: Collect and compare logs/artifacts from successful and failed runs.
* Retry Logic: Use `retry` in GitLab jobs to confirm if failures are transient.
* Isolation: Run jobs in isolated containers/VMs to avoid cross-job interference.

## Handling Multi-Environment Pipelines

* Environment Variables: Use GitLab’s environment-specific variables.
* Dynamic Environments: Use review apps for feature branches.
* Approval Gates: Implement manual approval for production deployments.
* Promotion Pipelines: Use artifacts to promote builds from lower to higher environments.
* Config as Code: Parameterize environment-specific configs.
* Rollback Strategy: Ensure rollback steps are defined per environment.

## Biggest Challenge as a DevOps Engineer

* Designing a secure, multi-tenant Kubernetes platform for a fintech client.
* Balancing compliance (PCI DSS), automation, and developer self-service.
* Ensuring isolation, auditability, and rapid delivery required deep collaboration.
* Implemented custom admission controllers and advanced IaC patterns.

## Agreeing/Disagreeing with a Lead

* Lead suggested using a single shared Kubernetes namespace for all microservices.
* Disagreed due to risks around security, resource contention, and blast radius.
* Proposed a namespace-per-team model with documentation and proof-of-concept.
* Adopted a hybrid approach after discussion.

## Admitting Mistakes Made by a Junior in Pipelines

* Junior engineer pushed a misconfigured pipeline that deleted staging resources.
* Took responsibility as the reviewer and explained the root cause to the team.
* Implemented better review and approval checks.
* Turned it into a learning opportunity.

## Major Loss Due to Your Actions

* Misconfigured a deployment script causing a brief production outage.
* Owned up, led the incident response, and documented lessons learned.
* Championed automated testing, canary deployments, and rollback mechanisms

DevOps, Cloud, and Linux Interview Q&A Cheat Sheet

That's a fantastic and comprehensive set of questions across Linux, CI/CD, Terraform, AWS, Docker, and Kubernetes — clearly aligned with your advanced DevOps and cloud engineering goals. Let's break this down into sections with concise answers and real-world relevance:

# 🔹 Linux Basics & Administration

## 1. Check running processes

ps aux # Detailed list of all processes

top # Interactive view of CPU/memory usage

htop # Enhanced version of top (if installed)

## 2. Schedule a cron job every 15 mins

\*/15 \* \* \* \* /path/to/script.sh

## 3. Hard vs Soft Links

* Hard Link: Points directly to inode. Cannot link directories. Survives original file deletion.
* Soft Link (Symlink): Points to file path. Can link directories. Breaks if target is deleted.

## 4. find vs locate

* find: Searches in real-time, slower but accurate.
* locate: Uses a prebuilt database (updatedb), faster but may be outdated.

## 5. Find processes using most CPU & memory

ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head

## 6. Log rotation

* Managed by logrotate. Rotates, compresses, and deletes old logs based on config in /etc/logrotate.d/.

## 7. Inode

* Metadata structure storing file attributes (not name). Each file has a unique inode.

## 8. Runlevels

* Legacy concept (SysVinit). Defines system state (e.g., 3 = multi-user, 5 = GUI). Replaced by systemd targets.

## 9. chmod 755

* Owner: read/write/execute
* Group & Others: read/execute

## 10. Set default file & directory permissions

umask 022 # Files: 644, Directories: 755

## 11. Check if a port is open/listening

netstat -tuln | grep <port>

ss -tuln # Modern alternative

## 12. Configure static IP

Edit /etc/network/interfaces or use nmcli/nmtui for NetworkManager systems.

## 13. Mounting & Unmounting

mount /dev/sdX /mnt

umount /mnt

## 14. Check disk partitions & usage

lsblk

df -h

fdisk -l

## 15. Sticky bit, setuid, setgid

* Sticky bit: Only owner can delete files in shared dir (/tmp)
* setuid: Run file with owner's privileges
* setgid: Run with group privileges or set group on new files in dir

# 🔹 CI/CD

## 1. Migrate pipelines

* Map stages/jobs from old tool to new (e.g., Jenkins → GitHub Actions)
* Convert syntax (Groovy → YAML)
* Validate secrets, triggers, and runners

## 2. Store secrets securely

* Use vaults (HashiCorp Vault, AWS Secrets Manager)
* CI/CD native secret stores (GitHub → Encrypted Secrets)

## 3. GitHub Actions popularity

* Native GitHub integration
* Easy YAML syntax
* Free runners for public repos
* Marketplace for reusable actions

## 4. Docker containers as CI/CD agents

Yes. Tools like GitLab CI, Jenkins use Docker containers as isolated build environments.

## 5. Enforce pipeline security

* Signed commits
* Secrets scanning
* Least privilege IAM roles
* Artifact integrity checks
* Use OPA/Gatekeeper for policy enforcement

# 🔹 Terraform

## 1. Secure state file

* Store in remote backend (S3 + DynamoDB for locking)
* Enable encryption (SSE)
* Restrict access via IAM

## 2. Avoid storing secrets

* Use vault, SSM, or Secrets Manager
* Avoid hardcoding in .tf or .tfvars

## 3. Restrict resource deletion

lifecycle {

prevent\_destroy = true

}

## 4. count vs for\_each

* count: Index-based, good for identical resources
* for\_each: Key-based, better for unique resources

## 5. Lifecycle blocks

* create\_before\_destroy: Avoid downtime
* prevent\_destroy: Protect critical infra
* ignore\_changes: Avoid unnecessary updates

## 6. terraform init/plan/apply

* init: Initialize backend, plugins
* plan: Preview changes
* apply: Execute changes

# 🔹 AWS

## 1. Public vs Private subnets

* Public: Route to Internet Gateway
* Private: No direct internet access (via NAT)

## 2. CloudFront & Elastic IP

* CloudFront: CDN for caching and global delivery
* Elastic IP: Static public IP for EC2

## 3. VPC Peering vs Direct Connect

* Peering: VPC-to-VPC, same region/account
* Direct Connect: Dedicated line to AWS, low latency, hybrid cloud

## 4. Route53 routing policies

* Simple, Weighted, Latency-based, Failover, Geolocation, Multi-value

## 5. Monitor EC2 with CloudWatch

* Use CloudWatch Agent
* Create dashboards
* Set alarms for thresholds

## 6. Lost private key & SSM disabled

* Use EC2 Instance Recovery
* Create AMI → Launch new instance
* Use EBS volume reattachment

# 🔹 Docker

## 1. CMD vs ENTRYPOINT vs RUN

* RUN: Build-time command
* CMD: Default runtime command
* ENTRYPOINT: Fixed runtime command

## 2. Volumes vs Bind Mounts

* Volumes: Managed by Docker, portable
* Bind Mounts: Host path, useful for dev

## 3. Cgroups & Dangling objects

* Cgroups: Resource limits (CPU, memory)
* Dangling: Unused images/volumes

## 4. Cache Docker layers

* Order Dockerfile smartly (dependencies first)
* Use .dockerignore

## 5. Delete all Docker resources

docker system prune -a

## 6. ADD vs COPY

* ADD: Can extract archives, fetch URLs
* COPY: Simple file copy, preferred for clarity

# 🔹 Kubernetes

## 1. K8s Architecture

* Master: API Server, Scheduler, Controller Manager
* Worker: Kubelet, Kube-proxy, Container Runtime

## 2. Kubelet, Kube-apiserver, Kube-proxy

* Kubelet: Manages pods on node
* API Server: Central control plane
* Kube-proxy: Handles networking

## 3. StatefulSet vs Deployment

* StatefulSet: Stable identity, persistent storage
* Deployment: Stateless apps

## 4. Troubleshooting 503 errors

* Case 1: Check DNS, service selectors
* Case 2: Check LB health checks, target groups

## 5. Secrets & Certificates in EKS

* Use AWS Secrets Manager, KMS
* Integrate with CSI driver

## 6. Restrict Pod communication

* Use NetworkPolicies

## 7. StatefulSet Pods sharing volume

* Use volumeClaimTemplates for unique PVCs

## 8. Fix OOMKilled Pods

* Increase memory limits
* Optimize app memory usage

## 9. Probes

* Liveness: Restart if unhealthy
* Readiness: Traffic only if ready
* Startup: Delay checks during boot

## 10. Update cluster version

* Use EKS console or CLI
* Upgrade node groups after control plane

## 11. Firewall breakage

* Use VPC Flow Logs, Security Groups
* Restore connectivity via bastion or SSM

## 12. Centralized logging

* Stack: Fluentd/Fluent Bit + Elasticsearch/OpenSearch + Kibana/Grafana
* Use EFK or Loki stack

Would you like this formatted into a cheat sheet PDF, or want to dive deeper into any specific section (e.g., Terraform lifecycle or Kubernetes probes)?

Terraform Interview Q&A

## Terraform Interview Q&A

## 🔹 How do you manage Terraform state in a team environment, and what challenges can arise with remote backends?

* Use remote backends like S3 (with DynamoDB for locking), Terraform Cloud, or Azure Storage.
* Challenges:
* State locking issues (e.g., if DynamoDB isn't configured properly).
* Concurrent changes causing race conditions.
* Access control—ensure only authorized users can modify state.
* State file corruption—requires versioning and backups.

## 🔹 Explain a scenario where you would use workspaces. How do they differ from multiple state files?

* Workspaces isolate state within the same backend (e.g., dev, prod).
* Multiple state files use different backends or paths.
* Use case: Workspaces are ideal for same infra across environments, while separate state files are better for different infra or accounts.

## 🔹 You have multiple AWS accounts (Dev, QA, Prod). How would you structure your Terraform code to handle cross-account deployments?

* Use separate workspaces or directories per account.
* Structure:

modules/

environments/

dev/

qa/

prod/

* Use assume\_role in provider blocks:

provider "aws" {

alias = "prod"

region = "us-east-1"

assume\_role {

role\_arn = "arn:aws:iam::PROD\_ACCOUNT\_ID:role/TerraformRole"

}

}

## 🔹 How do you implement modularization in Terraform? Share an example where poor module design caused issues.

* Break infra into reusable modules (e.g., VPC, EC2, RDS).
* Poor design example: tightly coupled modules with hardcoded values or no input validation, causing inflexibility and duplication.

## 🔹 Your team is facing state file corruption. How would you troubleshoot and recover safely?

* Steps:

1. 1. Restore from S3 versioning or backup.
2. 2. Use terraform state list and terraform state rm to clean up.
3. 3. Validate with terraform plan.
4. 4. Lock state access during recovery.

## 🔹 How would you implement Terraform in a CI/CD pipeline while ensuring security of sensitive variables?

* Use tools like Vault, AWS Secrets Manager, or SOPS.
* Avoid storing secrets in .tfvars or state.
* Use CI/CD secrets injection (e.g., GitHub Actions secrets, GitLab CI variables).
* Example:

variable "db\_password" {

type = string

sensitive = true

}

## 🔹 Scenario: You deployed infra with Terraform, but someone made manual changes in the cloud console. How do you detect and fix drift?

* Use terraform plan regularly.
* Use terraform refresh cautiously.
* Tools: Driftctl, Terraform Cloud drift detection.
* Fix: Reconcile manually or re-apply Terraform config.

## 🔹 In a large enterprise setup, how do you organize Terraform projects for 100+ microservices across multiple environments?

* Use mono-repo or multi-repo strategy.
* Structure by service + environment:

services/

service-a/

dev/

prod/

* Use Terragrunt or Terraform Cloud workspaces for orchestration.

## 🔹 How do you use lifecycle rules like prevent\_destroy and create\_before\_destroy for safe production rollouts?

* prevent\_destroy: Prevent accidental deletion.
* create\_before\_destroy: Ensure zero downtime during replacement.
* Example:

lifecycle {

prevent\_destroy = true

create\_before\_destroy = true

}

## 🔹 How do you handle Terraform upgrades (e.g., 0.11 → 0.12 or 0.13 → 1.x) in production environments without downtime?

* Use terraform 0.Xupgrade tool.
* Test in non-prod first.
* Validate with terraform plan.
* Use version constraints in modules and providers.

## 🔹 Can you explain how to design a multi-region highly available infrastructure using Terraform?

* Use modules with region as a variable.
* Deploy resources like ALBs, RDS replicas, Route53 failover.
* Use count or for\_each for region-specific resources.

## 🔹 How do you manage provider versioning to avoid breaking changes in long-lived projects?

* Lock versions in required\_providers block:

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 5.0"

}

}

}

* Use terraform init -upgrade cautiously.

## 🔹 What’s the difference between using depends\_on and implicit dependency resolution in Terraform? Share a real-world scenario.

* Implicit: Terraform infers from resource references.
* Explicit: Use depends\_on when dependency isn't obvious.
* Example:

resource "aws\_instance" "app" {

depends\_on = [aws\_security\_group.sg]

}

## 🔹 How do you handle secret management in Terraform without exposing sensitive data in state files?

* Avoid hardcoding secrets.
* Use sensitive = true for variables.
* Use external secret managers or encrypted files (e.g., SOPS + KMS).

## 🔹 How would you use data sources to reference existing infrastructure? Give a practical example.

* Reference existing infra:

data "aws\_vpc" "default" {

default = true

}

* Use case: Integrating with shared VPC or existing IAM roles.

## 🔹 How do you implement blue-green or canary deployments using Terraform?

* Use count or for\_each to deploy two environments.
* Use Route53 or ALB to switch traffic.
* Canary: Gradually increase traffic to new version.

## 🔹 You need to provision resources across multiple cloud providers (AWS, Azure, GCP). How would you structure your code?

* Use separate provider blocks:

provider "aws" { ... }

provider "azurerm" { ... }

provider "google" { ... }

* Structure code per cloud or use Terragrunt for orchestration.

## 🔹 What’s the impact of terraform refresh and when should you avoid using it?

* Updates state with real infra.
* Avoid if manual changes are not yet reconciled.
* Use cautiously in CI/CD.

## 🔹 How do you test Terraform code before applying changes in production? (unit tests, integration tests, policy checks, etc.)

* Unit tests: terraform validate, tflint.
* Integration tests: terratest, kitchen-terraform.
* Policy checks: OPA, Sentinel, Conftest.

## 🔹 Share a scenario where combining Terraform with Ansible (or another config mgmt tool) was necessary.

* Terraform provisions infra.
* Ansible configures OS/apps.
* Example: Terraform creates EC2, Ansible installs Nginx.