Interview Question Answers

Q1. Infrastructure as Code (IaC)

Modules help with configuration reproducibility across multiple environments. Working with Terraform modules ensures consistency across staging and production.

For example, I can have a compute module that handles Compute Engine, Managed Instance Groups, Cloud Functions, and related configurations, a GKE module for Kubernetes-related setup, and a networking module for components like VPCs, subnets, routes, and firewall rules.

Both staging and production reuse the same modules, which helps ensure they are set up in exactly the same way. The only difference is the values passed in which are handled using environment-specific variables through separate *tfvars* files.

To avoid drift, Terraform state files act as the source of truth, tracking what's defined in code versus what exists in GCP. Each environment has its own remote state, and state locking ensures only one person or pipeline can make changes at a time, preventing race conditions and conflicting updates.

I also prevent drift by limiting manual changes in the GCP console, running regular terraform plan checks to detect unexpected changes, and importing any manually created resources back into Terraform if needed.

In practice, I always apply changes to staging first, test thoroughly, and then promote the exact same code to production.

Q2. Kubernetes Observability

In this scenario, get the list of pods in the namespace (or across all namespaces, depending on the case):

```
k get pods -n namespace
Or
k get pods -A
```

Identify the pod giving the error and describe the pod

```
k describe pod pod_name -n namespace
```

This will show the events at the bottom (image pull errors, probe failures, OOMKilled hints, etc)

Next is to check the logs to get more information if it's due to application errors or resource constraints

```
k logs pod_name -n namespace
```

If it's an application error, it's usually due to missing configuration, secrets,environment variables, etc.

Fix: entrypoint or config if the error is from there.
If it's a resource constraint, the pod is often OOMKilled due to insufficient memory or CPU.
Fix: check resource requests and limits, review usage, and adjust memory or CPU accordingly.

Q3. Secret Management

In this case, the best approach security-wise is to store your secrets in a vault or secret manager, for example Google Secret Manager. GKE then uses Workload Identity so workloads like pods running inside the cluster can securely access those secrets without using long-lived credentials.
The secrets are injected into the container applications using either the CSI Driver or the External Secrets Operator.

The key thing is that both the CSI Driver and External Secrets Operator solve the same problem securely providing secrets to workloads, but they do it in different ways.

With the CSI Driver, the secret is fetched directly from Secret Manager and mounted into the pod at runtime, without being stored in the cluster.

With External Secrets, the secret is synced from Secret Manager into the cluster as a Kubernetes Secret, which the pod then consumes.

Steps:

- Enable Google Secret Manager, create the secret, and set up Workload Identity on the GKE cluster.
- Create a Google service account, grant it Secret Manager access, create a Kubernetes service account, and bind the Kubernetes service account to the Google service account using Workload Identity.
- Enable (or install via Helm) the Secrets Store CSI Driver and the GCP Secret Manager provider on the cluster.
- Create a SecretProviderClass that defines which Secret Manager secret should be exposed to the workload.
- Configure the pod to use the Kubernetes service account and mount the secret via the CSI Driver, then verify access and rotation inside the container.

Anti-patterns to avoid are:

- committing secrets to Git (.env, Helm values, manifests)
- echoing secrets in CI logs
- baking secrets into Docker images/build args
- storing secrets in Terraform state/outputs
- Share passwords across environments
- Give everyone production access

Q4. Reliability (Redis)

For Redis session management, monitoring availability, memory pressure, latency, and errors, is essential because those are the fastest ways sessions start failing (logouts, slow requests, etc).

What I monitor (and alert on):

- *Redis Availability:*
  Alert if Redis is not reachable / ping fails for 1–2 minutes.

- *Memory usage (most important for sessions)*:
  Warning if memory is >80–85% of maxmemory
  Critical if >90–95%
  If maxmemory isn't set, I still alert if used_memory keeps climbing and there's no headroom.

- *Evictions (sessions being kicked out):*
  Alert if evicted_keys is > 0 and increasing
  For sessions, even small sustained evictions is a big deal because users will get logged out.

- *Latency*:
  Warning if p95 latency > 10–20ms
  Critical if p95 > 50ms for a few minutes
  (Exact numbers depend on traffic)

- *CPU / Blocked commands*:
  Alert if Redis CPU is high and sustained (e.g. >85% for 5–10 mins)
  Alert if blocked_clients is > 0 and not dropping quickly (it means commands are stuck).

Why these metrics?
Because for sessions, the biggest "user impact" signals are: evictions, memory pressure, latency, availability. If Redis is healthy on those, sessions stay stable.