

GCP Production Interview Guide

Platform, SRE & DevOps interview guide

Questions	90 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/gcp_interview_guide.html
Printed	2026-10-21

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

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

IAM & identity

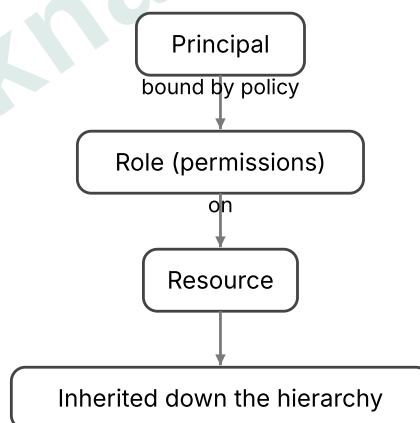
🗣 Spoken answer · ~78 s

GCP IAM answers who can do what on which resource. Principals are the identities: Google accounts, groups, service accounts, Workspace domains, and federated workload or workforce identities. A role is a collection of permissions, where a permission is a single action such as `storage.objects.get`. Roles are basic (owner, editor, viewer, far too broad for production), predefined (curated by Google per service), or custom. An allow policy binds principals to roles at a resource, and the binding is attached at the organisation, folder, project, or an individual resource. Policies are inherited downward through the hierarchy, and the effective permissions are the union of what's granted at each level, so you cannot take away at a child what a parent grants, except with deny policies. Deny policies and IAM conditions add guardrails, for example restricting access by time or resource tag. Best practices: grant roles to groups rather than individuals, use the narrowest predefined role at the lowest practical scope, avoid basic roles, use dedicated service accounts per workload, and audit with Cloud Audit Logs and IAM Recommender. Policy Troubleshooter explains why a principal does or doesn't have access.

Likely follow-ups

- What is the difference between allow and deny policies?
- Why avoid the basic Editor role?

🚩 GCP IAM



GCP IAM controls who (principal) can do what (role) on which resource (policy binding). A policy is a collection of role bindings attached to a resource.

CORE CONCEPTS

```
gcloud projects get-iam-policy <project> # view project-level policy
```

```
gcloud projects add-iam-policy-binding <project> --member="user:x@y.com" --  
role="roles/viewer"
```

```
gcloud iam roles describe roles/container.admin # inspect permissions in a role
```

Principals: Google accounts, service accounts, groups, domains, allUsers/allAuthenticatedUsers. Roles: Basic (Owner/Editor/Viewer — avoid in prod), Predefined (fine-grained per service), Custom (least privilege). IAM policies are additive — no explicit deny. Resource hierarchy: Org → Folder → Project → Resource. Policies inherited downward.

Interview tip: Mention that IAM conditions allow attribute-based access control (restrict by time, IP, resource tag). This is the Principal-level nuance interviewers listen for.

Official docs

- [Google Cloud IAM overview](#)

Q2 A service account has too many permissions — how do you audit and enforce least privilege?

IAM & identity

Spoken answer · ~82 s

I start by finding out what the service account actually uses versus what it's granted. IAM Recommender analyses roughly ninety days of activity and suggests removing unused permissions or replacing a broad role with a narrower one; the Policy Analyzer shows who can access what; and Cloud Audit Logs show the real API calls. I also check where the account is attached and used, whether it's the default Compute Engine service account, which often has the Editor role and should be replaced with dedicated accounts. Remediation: replace basic roles with specific predefined or custom roles, move bindings to the lowest scope such as a single bucket or dataset, and remove unused service accounts and keys. Make the change safely: apply the recommendation in a test environment or use policy simulation first, then monitor for permission denied errors, and keep the previous policy to roll back. Enforcement: organisation policies that disable service account key creation and prevent automatic Editor grants to default accounts; Terraform with reviews so permissions are code; alerts on high-privilege grants; restricting who can impersonate or act as the account; and periodic access reviews. Prefer short-lived credentials via impersonation or workload identity over keys.

Likely follow-ups

- What does IAM Recommender actually analyse?
- Why is the default compute service account risky?

```
gcloud projects get-iam-policy <project> --format=json | jq '.bindings[] | select(.members[] | contains("serviceAccount:"))'
```

```
gcloud policy-intelligence query-activity --project=<project> # what permissions were actually used
```

```
gcloud iam service-accounts list --project=<project>
```

Use Policy Analyzer and IAM Recommender — GCP automatically suggests permission reductions based on 90-day usage history. Recommender shows which permissions are granted but never used. Create custom roles with only the permissions actually needed. Audit with Cloud Asset Inventory for org-wide SA permission scan.

Automated enforcement: Enable IAM Recommender via Security Command Center and set up alerts for high-risk recommendations. This is how platform teams scale least-privilege across hundreds of SAs without manual audits.



Official docs

- [Kubernetes: RBAC](#)
- [Kubernetes: Service accounts](#)
- [AWS IAM: Least privilege \(grant least privilege\)](#)

Q3 What is Workload Identity Federation — how does it replace service account keys?

IAM & identity

 Spoken answer · ~86 s

Workload Identity Federation lets external workloads, such as GitHub Actions, AWS, Azure, or an on-premises system with an OIDC or SAML identity provider, access Google Cloud without a downloaded service account key. You create a workload identity pool and a provider that trusts the external issuer. The workload presents its own token, for example the GitHub OIDC token, to Google's Security Token Service, which validates it against the provider and attribute mappings and conditions, such as the repository name, and returns a short-lived federated token. That token can be used directly, by granting roles to the federated principal, or exchanged to impersonate a service account, which gets a short-lived access token. This replaces long-lived keys, which are the most common source of credential leaks: they can be copied, committed to repositories, never rotate, and are hard to audit. With federation there is nothing static to steal, tokens last about an hour, and access can be restricted to a specific repository, branch, or environment through attribute conditions. Setup involves the pool, provider, attribute mapping, a binding allowing the principal to impersonate the service account, and configuring the client, for example the google-github-actions/auth action. Always restrict conditions tightly so other repositories in the same organisation cannot use the trust.

Likely follow-ups

- Why must you add attribute conditions to the provider?
- What is the difference between direct federation and impersonation?

Workload Identity Federation lets external workloads (GitHub Actions, AWS, Azure, on-prem) authenticate to GCP using their native identity — no service account key files needed.

```
gcloud iam workload-identity-pools create <pool> --location=global
```

```
gcloud iam workload-identity-pools providers create-oidc <provider> --workload-identity-pool=<pool> --issuer-uri=https://token.actions.githubusercontent.com
```

```
gcloud iam service-accounts add-iam-policy-binding <sa> --role=roles/iam.workloadIdentityUser --member="principalSet://..."
```

Flow: external system gets OIDC token → exchanges with GCP STS → gets short-lived access token → calls GCP APIs. No keys to rotate, no keys to leak. For GKE specifically, Workload Identity maps a Kubernetes ServiceAccount to a GCP ServiceAccount — pods call GCP APIs without key files.

Never use SA key files in production. They don't expire by default, are hard to rotate, and are the #1 source of GCP credential leaks. Workload Identity is the production standard for every external integration.

Official docs

- [Kubernetes: Service accounts](#)
- [Azure: Managed identities](#)
- [Google Cloud: Workload Identity Federation](#)

Q4 IAM policy change accidentally removed access — how do you recover?

IAM & identity

Spoken answer · ~83 s

First, stop the damage and restore access, then work out what happened. If I have the previous policy, I can reapply it. Cloud Audit Logs record SetIamPolicy calls in Admin Activity logs, which show who changed what and when, along with the request containing the new policy, so I can compare and identify the removed bindings. If access is fully lost at project level, a principal with higher-level access can fix it: an organisation or folder administrator, since permissions inherit down, or in the worst case the Organization Administrator, which can always regain control. For a locked-out project, the Owner at organisation level can re-add bindings. Use the Policy Troubleshooter to confirm why a principal lacks access. After restoring, make the change controlled: manage IAM through Terraform or another IaC with pull request review, so every change is reviewable and revertible from version control; use the policy simulator to preview the effects of changes before applying; add alerts on SetIamPolicy events for sensitive resources; limit who holds the permissions to change IAM; and keep break-glass accounts with strong protection and monitoring. Finally, document the incident and add guardrails, like organisation-level admin groups, so a single mistake cannot lock everyone out.

Likely follow-ups

- Where would you find who made the change?
- How does Terraform reduce the risk of this happening?

```
gcloud projects get-iam-policy <project> --format=json > backup.json # always backup first
```

```
gcloud projects set-iam-policy <project> backup.json # restore from backup
```

```
gcloud logging read 'protoPayload.methodName="SetIamPolicy"' --limit=5 --format=json #  
audit who changed what
```

If you locked yourself out of a project: use a break-glass account (org-level admin) to restore access. Check Cloud Audit Logs (Admin Activity) for the exact SetIamPolicy call — it records the before/after policy diff. For org-level lockout, contact GCP Support with proof of ownership.

Prevention: Use Terraform for IAM management — all changes go through PR review and state diff. Never make manual IAM changes in production. Store IAM policy backups in GCS with versioning.

Q5 How does GCP Service Account impersonation work and when should you use it? IAM & identity

Spoken answer · ~88 s

Service account impersonation lets one principal temporarily act as a service account and obtain short-lived credentials for it, without anyone having a key. The caller needs the Service Account Token Creator role on the target service account, or `iam.serviceAccounts.getAccessToken`, and the IAM Credentials API issues an access token, or an ID token or signed blob, usually valid for up to an hour. Calls are logged, showing both the caller and the service account used. Use it when a human or a pipeline needs elevated access for a task: developers can run `gcloud` with the `impersonate` flag, or application default credentials with impersonation, to test with the same permissions the production workload has, while their own accounts stay low-privilege. CI/CD systems use it after federated authentication to act as a deploy service account. It supports chaining through delegation and lets you separate duties by having small-scope service accounts. Compared with downloaded keys it is much safer: nothing to leak, automatic expiry, and a clear audit trail. Restrict who may impersonate each account, since the Token Creator role effectively grants all the permissions of that account, so treat it as sensitive. Contrast it with `actAs`, the Service Account User role, which lets a principal attach the account to a resource such as a VM.

Likely follow-ups

- What is the difference between Token Creator and Service Account User?
- How does impersonation improve audit trails?

```
gcloud --impersonate-service-account=<sa>@<project>.iam.gserviceaccount.com compute instances list
```

```
gcloud iam service-accounts add-iam-policy-binding <target-sa> --member="serviceAccount:<caller-sa>" --role="roles/iam.serviceAccountTokenCreator"
```

Impersonation lets one SA act as another SA — short-lived credentials, full audit trail, no key exchange. Use it for: pipeline SA impersonating deployment SA (privilege separation), breaking glass scenarios, testing SA permissions without a key. Requires `roles/iam.serviceAccountTokenCreator` on the target SA.

Security pattern: Human users never get SA keys — they impersonate SAs via their own identity. The audit log shows both the human and the SA identity, giving full traceability. This is the recommended model for developer access to production resources.

Official docs

- [Kubernetes: Service accounts](#)
- [Google Cloud: Best practices for service accounts](#)

Q6 GKE cluster upgrade is failing — how do you troubleshoot and manage it safely?

GKE

Spoken answer · ~85 s

A failing GKE upgrade is troubleshooted by first identifying which part failed: control plane or node pools. Check the operation status with `gcloud container operations list` and `describe`, and the cluster's status message and Cloud Logging for errors. Common causes: insufficient quota or IP addresses for surge nodes; PodDisruptionBudgets that block draining, so nodes cannot be evicted; pods that won't reschedule due to resource constraints, affinity rules, or no capacity; admission webhooks that are unavailable and reject API calls; deprecated API usage that breaks workloads in the new version; and custom node images or startup issues. Fixes: add quota or capacity, temporarily relax PDBs, fix or scale the problematic workloads, and correct webhooks with failure policies. To manage upgrades safely: use release channels and maintenance windows and exclusions; check deprecation insights and recommended upgrade paths beforehand; upgrade a non-production cluster first; configure surge upgrades with sensible max surge and unavailable values, or blue-green node pool upgrades for fast rollback; ensure workloads have multiple replicas, PDBs, and readiness probes; back up state; and monitor during the upgrade. Control plane upgrades are done by Google and node pools follow. Never skip several minor versions. Communicate the schedule and have a rollback plan, like a parallel node pool.

Likely follow-ups

- How do PodDisruptionBudgets interfere with upgrades?
- What is a blue-green node pool upgrade?

```
gcloud container clusters describe <cluster> --zone=<zone> | grep -A5 currentMaster
```

```
gcloud container operations list --filter="status=RUNNING" # any pending operations
```

```
gcloud container clusters upgrade <cluster> --master # upgrade control plane first
```

GKE upgrades: control plane first (automatic with release channels), then node pools. Enable surge upgrade on node pools: adds extra nodes during upgrade so pods are never disrupted. Set `--max-surge=1 --max-unavailable=0` for zero-downtime node upgrades. Use release channels (Rapid/Regular/Stable) instead of manual version pinning — GKE manages the upgrade cadence for you.

Production pattern: Use Regular channel for production. Rapid for dev/staging to catch issues early. Stable only if you have compliance requirements for specific versions.

Official docs

- [Kubernetes: Upgrading kubeadm clusters](#)
- [GKE: Cluster upgrades](#)
- [GKE overview](#)

Q7 **GKE nodes are experiencing high memory pressure — how do you diagnose and fix?**

GKE

Spoken answer · ~82 s

Memory pressure on GKE nodes shows up as pods being OOMKilled or evicted, and nodes with the MemoryPressure condition. To diagnose, check `kubectl describe node` for conditions and allocated resources, and `kubectl top nodes` and `kubectl top pods` to find heavy consumers; use Cloud Monitoring or Prometheus to look at container memory working set against requests and limits over time; check events for evictions and OOM kills; and on the node look at `dmesg`, kubelet logs, and system daemons. Typical causes: containers without limits or with limits much higher than requests, so the node is overcommitted; memory leaks or growing caches; too many pods per node; node-level overhead from system components and the kubelet reserving; and burstable workloads all spiking together. Fixes: set accurate requests and limits, with requests close to real usage and limits that avoid OOM, using the Vertical Pod Autoscaler recommendations; use Guaranteed or higher QoS for critical pods; use priority classes so important pods are evicted last; fix leaks in the application; spread workloads with anti-affinity or topology constraints; move to larger or memory-optimised nodes; and enable the cluster autoscaler or node auto-provisioning. Add alerts on node memory and eviction counts before pressure becomes outages.

Likely follow-ups

- How do requests and limits influence eviction order?
- How can VPA help here?

```
kubectl top nodes
```

current usage

```
kubectl describe node <node> | grep -A10 "Allocated resources"
```

```
kubectl get pods -A --field-selector spec.nodeName=<node> # pods on node
```

Distinguish between: requested memory (scheduling pressure), used memory (actual pressure), and allocatable memory (total minus OS and kubelet reservation). GKE reserves memory for OS and system daemons — subtract this from VM memory. Fix: scale node pool, enable cluster autoscaler, set accurate memory requests, or move memory-heavy workloads to dedicated node pools.

GKE Autopilot advantage: No node management — GCP provisions and manages nodes automatically based on pod requests. Memory pressure becomes a pod scheduling concern, not a node ops concern. Recommend Autopilot for new clusters unless you need node-level customization.

Q8 How do you configure GKE Workload Identity for a pod to access GCP services?

GKE

🗣 Spoken answer · ~78 s

GKE Workload Identity lets a pod authenticate to Google Cloud APIs as a Google identity, without service account keys. Setup: enable Workload Identity on the cluster, which creates the workload identity pool PROJECT.svc.id.goog, and on node pools the GKE metadata server mode. Newer clusters can have it enabled by default. There are two approaches. The recommended direct method uses principal identifiers: grant IAM roles directly to the Kubernetes service account's principal, identified by project, namespace, and name, so no Google service account is needed. The classic method creates a Google service account with the needed roles, grants it the roles/iam.workloadIdentityUser binding for member serviceAccount:PROJECT.svc.id.goog[NAMESPACE/KSA_NAME], and annotates the Kubernetes service account with iam.gke.io/gcp-service-account. Pods then run with that Kubernetes service account, and client libraries using application default credentials fetch tokens automatically from the metadata server. Verify from the pod with gcloud auth list or a call to the metadata endpoint, and watch for errors like permission denied. Best practices: one service account per workload, least privilege, avoid the default Compute account on nodes, and block access to the node's metadata credentials. Audit logs show which workload made the calls.

Likely follow-ups

- What is the benefit of direct principal identifiers?
- How do you verify a pod is using Workload Identity?

```
gcloud container clusters update <cluster> --workload-pool=<project>.svc.id.goog
```

```
gcloud iam service-accounts add-iam-policy-binding <gcp-sa>@<project>.iam.gserviceaccount.com --role=roles/iam.workloadIdentityUser --member="serviceAccount:<project>.svc.id.goog[<namespace>/<ksa>]"
```

```
kubectl annotate serviceaccount <ksa> -n <ns> iam.gke.io/gcp-service-account=<gcp-sa>@<project>.iam.gserviceaccount.com
```

Workload Identity is the recommended way for GKE pods to call GCP APIs. The pod's Kubernetes SA is bound to a GCP SA — the metadata server on the node exchanges the KSA token for a GCP access token transparently. No key files, automatic rotation, full audit trail per pod.

Common mistake: Forgetting to annotate the Kubernetes ServiceAccount OR forgetting the IAM binding on the GCP SA. Both are required. Test with `kubectl exec -- gcloud auth list` from the pod.

Official docs

- [Azure: Managed identities](#)
- [GKE: Workload Identity Federation for GKE](#)
- [GKE overview](#)

Q9 GKE cluster autoscaler is not scaling up — how do you debug?

GKE

Spoken answer · ~82 s

When the cluster autoscaler isn't adding nodes, I look at why pods are pending and what the autoscaler thinks. Start with `kubectl describe pod` to read scheduling events, including `TriggeredScaleUp` or `NotTriggerScaleUp` with reasons, and check the cluster autoscaler visibility logs in Cloud Logging. Common reasons: the node pool is at its maximum size; project quota, such as CPU, IP address, or disk limits, is exhausted, or the zone has no capacity; the pod's requests exceed the largest node's capacity, so no node type could fit it; node selectors, taints, affinities, or topology constraints that no node pool satisfies; the pod lacks resource requests so the autoscaler can't compute needs; `PersistentVolume` zone constraints; the pod is unschedulable for reasons scaling can't fix; autoscaling isn't enabled on the node pool or the autoscaling profile is conservative; and pods with local storage or restrictive annotations. Fixes: raise the maximum node count or quota, add a suitable node pool or enable node auto-provisioning, correct requests and constraints, use multiple zones, and use balanced autoscaling profiles. Verify the autoscaler status `ConfigMap` in `kube-system`. Monitor pending pods and scale-up latency, and use overprovisioning with low-priority placeholder pods when fast scaling matters.

Likely follow-ups

- What does `NotTriggerScaleUp` tell you?
- How do placeholder pods speed up scaling?

```
kubectl describe configmap cluster-autoscaler-status -n kube-system # autoscaler decisions
```

```
kubectl get events -n <ns> | grep -i "FailedScheduling|ScaleUp"
```

```
gcloud container clusters describe <cluster> | grep -A5 autoscaling
```

Autoscaler won't scale if: node pool is at max size (increase `maxNodeCount`), pods have `PodAntiAffinity` preventing placement on a new node, node selector/taint rules prevent the pod from running on any new node type, or the pending pod requests are larger than the node pool's instance type. Check the `cluster-autoscaler-status ConfigMap` — it explains why scale-up was skipped.

Scale-down issue: Autoscaler won't remove a node if any pod has a local PVC (`emptyDir` with data), `PodDisruptionBudget` blocks eviction, or pod has `cluster-autoscaler.kubernetes.io/safe-to-evict: "false"` annotation.

Official docs

- [Kubernetes Autoscaler: Cluster Autoscaler](#)
- [GKE overview](#)

Q10 How do you set up a private GKE cluster and what are the connectivity implications?

GKE

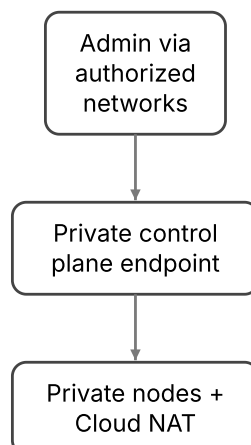
Spoken answer · ~83 s

A private GKE cluster gives nodes only internal IP addresses, so they're not reachable from the internet. The control plane has a private endpoint and optionally a public endpoint. Set it up with private nodes enabled, a VPC-native cluster with secondary ranges for pods and services, a control plane IP range of /28 for peering, and master authorised networks to restrict which CIDRs can reach the API server. You can disable the public endpoint entirely. Connectivity implications: nodes can't pull images from the internet or call external APIs unless you provide Cloud NAT; Google APIs and Artifact Registry are reached through Private Google Access; the control plane lives in a Google-managed VPC connected through peering, or Private Service Connect in newer clusters, so peering routes need to be exported to reach the control plane from on-premises over VPN or Interconnect; engineers or CI systems must reach the private endpoint from within the VPC, via a bastion, Cloud VPN, Identity-Aware Proxy tunnelling, or the Connect gateway; webhooks need firewall rules allowing the control plane to reach nodes on specific ports; and external load balancers still work for exposed services. It reduces the attack surface but needs planned networking and access paths.

Likely follow-ups

- Why do private nodes still need Cloud NAT?
- How would a CI system reach a private control plane?

Private GKE



```
gcloud container clusters create <name> --enable-private-nodes --master-ipv4-cidr=172.16.0.0/28 --enable-ip-alias
```

```
gcloud container clusters update <name> --enable-master-authorized-networks --master-authorized-networks=<cidr>
```

Private clusters: nodes have no public IPs — only RFC1918 addresses. Control plane is in a GCP-managed VPC peered to your VPC. Implications: nodes need Cloud NAT for outbound internet (image pulls, external APIs), kubectl access requires authorized networks or VPN/bastion, and inter-node traffic stays on Google's network. Use Master Authorized Networks to restrict control plane API access to specific CIDRs.

Image pull: Private nodes cannot reach Docker Hub directly. Either mirror images to Artifact Registry, configure Cloud NAT, or use GKE's image streaming feature. This catches teams off-guard the first time.

Official docs

- [GKE overview](#)

Q11 VPC network is misconfigured — traffic between VMs is not working. How do you debug?

Networking

Spoken answer · ~85 s

When VMs can't communicate, I work through the layers. First verify the basics: both VMs are running and in the expected VPC and subnets, with correct internal IP addresses. Then firewall rules: VPC firewalls are stateful and the default denies ingress, so check there is an allow rule matching the protocol, port, source ranges, and target tags or service accounts, and consider rule priorities and any deny rules; hierarchical firewall policies at folder or organisation level may also block. Routes: confirm that routes exist for the destination and that a custom route or a higher-priority route isn't sending traffic elsewhere. Use Connectivity Tests in Network Intelligence Center, which simulates the path and reports which firewall rule or route drops the packet; this is usually the fastest tool. Enable VPC Flow Logs and firewall rules logging to observe real traffic. Also check the guest operating system: local firewalls such as iptables, the service actually listening on the port, and the correct network interface. For cross-VPC traffic, verify peering is active in both directions with route exchange, no overlapping CIDRs, and that peering isn't transitive. Check DNS if names are involved, and network tags typos. Finally test with ping, curl, or traceroute from both sides.

Likely follow-ups

- What does a Connectivity Test show you?
- Why isn't VPC peering transitive an issue here?

```
gcloud compute firewall-rules list --filter="network=<vpc>" # all firewall rules
```

```
gcloud compute networks subnets describe <subnet> --region=<region>
```

```
gcloud compute connectivity-tests create test1 --source-instance=<vm1> --destination-instance=<vm2> --protocol=TCP --destination-port=80
```

Debug layers: (1) Firewall rules — default VPC allows internal traffic, custom VPCs do not. (2) Routes — is there a route to the destination subnet? (3) VM network tags — firewall rules target tags, not IPs. (4) Shared VPC — is the subnet in the correct host project? Use Connectivity Tests (Network Intelligence Center) to trace packet path and identify exactly where traffic is dropped.

Network Intelligence Center: Far faster than manual firewall rule analysis. It simulates the packet path and shows the exact firewall rule accepting or blocking traffic. Always use this before manually scanning rule lists.

 **Start here (official docs)**

- [Google Cloud documentation](#)

Q12 How does Cloud Load Balancing work in GCP — what type do you choose for which scenario?

Networking

 Spoken answer · ~86 s

Cloud Load Balancing is a family of fully managed load balancers, and the choice depends on traffic type, scope, and whether it's external or internal. The global external Application Load Balancer handles HTTP and HTTPS at layer 7, with a single anycast IP across regions, URL maps for content-based routing, SSL termination, Cloud CDN, Cloud Armor, and IAP; use it for public websites and APIs. The regional external Application Load Balancer suits workloads that must stay in a region. The internal Application Load Balancer is for private layer 7 traffic between services within a VPC, including across a service mesh. The external proxy Network Load Balancer handles TCP with SSL offload at global or regional scope. The external passthrough Network Load Balancer is a regional layer 4 pass-through for TCP, UDP, and other protocols, preserving client IP, suited to gaming, VoIP, or non-HTTP apps. The internal passthrough Network Load Balancer is layer 4 for internal services, often used as a next hop or for databases. Backends can be instance groups, network endpoint groups for containers and serverless, or buckets. Health checks control traffic. Decide by protocol, global versus regional reach, public versus private, needed features like WAF and CDN, and the need to preserve source IP.

Likely follow-ups

- When would you choose a passthrough load balancer?
- What is a network endpoint group?

External HTTP(S) LB: global, anycast IP, URL-based routing, Cloud CDN integration

Internal HTTP(S) LB: regional, private IP, for internal microservices

TCP/UDP Network LB: regional, preserves client IP, for non-HTTP protocols

```
gcloud compute backend-services describe <name> --global # inspect LB backend config
```

External HTTPS LB is GKE Ingress default — uses Google's global anycast network, enabling traffic to enter from the nearest PoP worldwide. Backend health is checked via GCP health checks. Premium Tier networking routes traffic on Google's backbone; Standard Tier uses public internet.

GKE Ingress note: Backend pods must pass GCP health checks or they're marked unhealthy in the LB backend. Check pod readiness AND GCP health check path/port alignment. A common source of 502 errors after deployment.

Official docs

- [Cloudflare Learning: Load balancing](#)
- [Cloud Load Balancing](#)

Q13 Cloud NAT is not working — VMs cannot reach the internet. How do you debug?

Networking

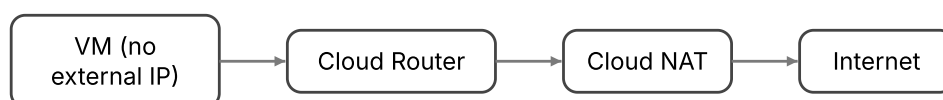
Spoken answer · ~85 s

If VMs with no external IP cannot reach the internet through Cloud NAT, I check each piece in order. Confirm a Cloud NAT gateway exists on a Cloud Router in the same region and VPC as the VM's subnet, and that the NAT configuration covers that subnet's IP ranges, including primary and secondary ranges, for GKE pods. Check there is a route to the internet: the default route 0.0.0.0/0 with the default internet gateway as next hop must exist, because NAT relies on it. Check firewall rules: egress deny rules or a low-priority allow can block traffic. Verify that the VM has no external IP, since VMs with one use it directly. Look at NAT allocation: if ports are exhausted, connections fail or drop, which appears as errors and in the NAT logs; increase the minimum ports per VM, add more NAT IP addresses, or enable dynamic port allocation. Enable Cloud NAT logging to see translation errors and dropped connections. Use Connectivity Tests for path analysis. Check DNS resolution separately, since failures may be name-related. For Google APIs, Private Google Access is needed instead of NAT. Also consider destination issues, like allowlisting of the NAT IPs by third parties, and idle timeouts on long-lived connections.

Likely follow-ups

- What is port exhaustion in Cloud NAT?
- When is Private Google Access enough without NAT?

Cloud NAT path



```
gcloud compute routers describe <router> --region=<region> # NAT router config
```

```
gcloud compute routers get-nat-mapping-info <router> --region=<region> # active NAT mappings
```

```
gcloud logging read 'resource.type="nat_gateway"' --limit=20 # NAT logs
```

Check: Cloud Router exists in the correct region, NAT config covers the correct subnet (AUTO or manual subnet selection), VM has no public IP (NAT only applies to VMs without external IPs), and NAT IP allocation is not exhausted. Enable Cloud NAT logging for dropped connection visibility.

Port exhaustion: Each NAT IP supports ~64K ports. High-connection workloads (like GKE pods doing many outbound calls) can exhaust NAT ports. Allocate additional NAT IPs or increase min-ports-per-VM in the NAT config.

Official docs

- [RFC 3022: Traditional NAT](#)
- [Cloud NAT](#)

Q14 How does VPC peering vs Shared VPC vs VPN differ — when do you use each?

Networking

Spoken answer · ~79 s

These are three different ways to connect networks. VPC peering connects two VPC networks privately over Google's internal network, exchanging routes. It is non-transitive, requires non-overlapping CIDR ranges, and each side keeps separate administration; it suits connecting a handful of VPCs, for example between teams or with a managed service. Shared VPC lets a host project own a central VPC whose subnets are shared with service projects in the same organisation. Resources in different projects use the same network, with central control over firewall rules, routes, and IP addressing, while teams keep project-level billing and IAM; it suits an organisation wanting centralised network governance. Cloud VPN creates encrypted IPsec tunnels over the public internet, in HA configuration with a 99.99 percent SLA, to connect on-premises or other clouds; it's quick to set up but with bandwidth limits per tunnel and variable latency. Interconnect offers dedicated or partner private connectivity for higher bandwidth and consistent latency. My guidance: use Shared VPC for internal organisation-wide design, peering for connecting separate VPCs or third-party services, and VPN or Interconnect for hybrid connectivity. For many VPCs, consider Network Connectivity Center for hub-and-spoke transitive routing.

Likely follow-ups

- Why is VPC peering non-transitive a design constraint?
- When do you move from Cloud VPN to Interconnect?

```
# VPC Peering: direct connectivity, no transitive routing, both projects manage own firewall
```

Shared VPC: central host project owns subnets, service projects use them

Cloud VPN / Interconnect: connect to on-premises or other clouds

gcloud compute networks peerings list # view existing peerings

VPC Peering: peer-to-peer, low latency, no transitive routing (A-B and B-C peering doesn't give A-C connectivity). Shared VPC: org-recommended for multi-project architectures — network team owns subnets, app teams deploy into them. VPN: encrypted tunnel over internet. Dedicated/Partner Interconnect: high-bandwidth on-prem connectivity (99.99% SLA for dedicated).

Enterprise pattern: Shared VPC with a hub-and-spoke topology. Central connectivity project (VPN/Interconnect) peered to the Shared VPC host. App projects are service projects. Network security team controls firewall rules centrally.



Official docs

- [Cloudflare Learning: What is a VPN?](#)
- [Amazon VPC: VPC peering](#)
- [Google Cloud: Shared VPC](#)

Q15 DNS resolution is failing for a GCP resource — how do you troubleshoot Cloud DNS?

Networking

Spoken answer · ~85 s

For DNS failures in GCP, I first work out what kind of name is failing and where the query goes. Reproduce from an affected VM with dig or nslookup, noting the response code: NXDOMAIN means the name doesn't exist, SERVFAIL points to a resolver or upstream problem, and a timeout suggests connectivity. Check which resolver the VM uses; by default it's the metadata server at 169.254.169.254, which resolves internal names and Cloud DNS private zones attached to the VPC. Then check the Cloud DNS side: does the managed zone exist, is it public or private, and is a private zone attached to the correct VPC network? Check the record sets for typos, TTL, and recently changed records that may be cached. For public zones, verify the domain registrar's name servers point to the Cloud DNS name servers. Check DNS policies: forwarding zones and inbound or outbound server policies, whether the forwarding target is reachable through VPN or Interconnect, and that firewall rules allow DNS traffic on port 53 from Google's forwarding ranges. For GKE, check kube-dns or CoreDNS or NodeLocal DNS. Enable Cloud DNS logging to see queries and responses. Also consider split-horizon conflicts and DNSSEC problems. Use Connectivity Tests for network paths.

Likely follow-ups

- What does SERVFAIL suggest compared with NXDOMAIN?
- How do forwarding zones work for hybrid DNS?

```
gcloud dns managed-zones list # list DNS zones
```

```
gcloud dns record-sets list --zone=<zone> # inspect records
```

```
gcloud compute instances ssh <vm> -- "dig +short <hostname> @169.254.169.254"
```

```
gcloud dns policies list # DNS policies on VPC
```

Check: DNS zone is attached to the correct VPC network (private zones must be authorized for the network), record TTL (stale cache), DNS policy enabling/blocking forwarding, and VPC DNS settings. For GKE, CoreDNS queries go to the metadata server (169.254.169.254) which proxies to Cloud DNS.

Hybrid DNS: For on-prem DNS resolution from GCP, use Cloud DNS inbound forwarding (creates forwarding IP in subnet) and configure on-prem DNS to forward GCP domain queries to it.

Official docs

- [RFC 1035: DNS implementation and specification](#)
- [Cloudflare Learning: What is DNS?](#)
- [Cloud DNS](#)

Q16 GCS bucket is returning 403 errors — how do you debug access issues?

Storage & data

Spoken answer · ~86 s

A 403 from Cloud Storage means the request was authenticated but not authorised, or blocked by a policy. Start by identifying who is calling: the user, service account, or federated identity, from the error and audit logs. Check IAM: does the principal have a role with storage.objects.get, list, or create on the bucket or project, for example Storage Object Viewer? Remember the permission is on the object, so listing buckets differs from reading objects. Check the bucket's access control mode: with uniform bucket-level access, only IAM applies and ACLs are ignored; with fine-grained access, object ACLs matter too. Check public access prevention and whether the bucket is meant to be public. Then guardrails that override IAM: organisation policies, IAM deny policies, IAM conditions with expiry or resource restrictions, VPC Service Controls perimeters blocking access from outside the perimeter, which produces a distinct error with a unique identifier, and requester pays buckets needing a billing project. For signed URLs, check expiry, signing account permissions, and clock skew. Use Policy Troubleshooter to evaluate the exact access, and Cloud Audit Logs data access logs, if enabled, for the denial reason. Also verify that you're using the right credentials with gcloud auth list, and the correct bucket name and project.

Likely follow-ups

- How do uniform bucket-level access and ACLs interact?
- Which tool shows why a specific principal is denied?

```
gcloud storage buckets get-iam-policy gs://<bucket> # bucket-level IAM
```

```
gsutil acl get gs://<bucket>/<object> # object-level ACL
```

```
gcloud logging read 'resource.type="gcs_bucket" protoPayload.status.code=403' --limit=10
```

```
gcloud auth application-default print-access-token # verify caller identity
```

403 causes: caller SA lacks roles/storage.objectViewer or storage.objectAdmin, bucket has uniform bucket-level access disabled with conflicting ACLs, VPC Service Controls perimeter blocking access, or bucket CORS config rejecting the origin. Check if Uniform Bucket-Level Access is enabled — it disables ACLs and uses IAM only.

VPC Service Controls: If the bucket is inside a service perimeter, only requests from within the perimeter are allowed. A SA outside the perimeter gets 403 even with correct IAM. Check SC perimeter config first when IAM looks correct.

Official docs

- [Cloud Storage](#)

Q17 **BigQuery query costs are unexpectedly high — how do you investigate and control costs?**

Storage & data

Spoken answer · ~79 s

BigQuery on-demand pricing charges by bytes scanned, so cost spikes usually come from inefficient queries or growing data. To investigate, query `INFORMATION_SCHEMA.JOBS` views or the billing export in BigQuery, grouping `total_bytes_billed` by user, project, query text, and day to find the expensive queries and who runs them. Common culprits: `SELECT *`, scanning unpartitioned tables, repeated dashboards re-running the same query, missing filters on partition columns, cross joins, and scheduled queries gone wrong. Control measures: partition tables, usually by date, and require partition filters; cluster on commonly filtered columns; select only needed columns; preview with dry runs to see the bytes to be processed; use materialised views or summary tables for repeated aggregations; and rely on the query cache. Set guardrails: maximum bytes billed per query, custom quotas per user and project, and budgets with alerts. Consider pricing models: capacity pricing through slot reservations or editions gives predictable cost for heavy workloads, and autoscaling slots handles spikes, while on-demand suits light usage. Control storage costs with table expiration and long-term storage pricing, and choose the physical versus logical storage billing model carefully. Educate analysts, and monitor costs on a dashboard.

Likely follow-ups

- How does partitioning reduce the bytes scanned?
- When does capacity pricing beat on-demand?

```
SELECT * FROM region-us.INFORMATION_SCHEMA.JOBS_BY_PROJECT WHERE total_bytes_billed > 1000000000 ORDER BY creation_time DESC LIMIT 20;
```

```
bq query --dry_run 'SELECT ...' # estimate cost before running
```

Cost controls: partition tables by date and require partition filter (rejects full scans), cluster tables on common filter columns, use column selection (avoid SELECT *), set project-level custom quotas (max bytes per day/user), use flat-rate slots for predictable cost vs on-demand pricing. Audit with INFORMATION_SCHEMA.JOBS to find expensive queries and their authors.

Partitioned tables + partition filter: Setting `require_partition_filter = true` forces all queries to include a WHERE clause on the partition column. This single change can reduce BigQuery costs by 90% for time-series data.

Official docs

- [BigQuery](#)

Q18 Cloud Spanner is showing high latency — how do you diagnose performance issues?

Storage & data

Spoken answer · ~83 s

For Spanner latency, I begin by separating where the time is spent: the client, network, or the database. Cloud Monitoring shows request latencies, CPU utilisation, and node count; the Spanner console's Query Insights and Transaction Insights, plus Key Visualizer, give deeper analysis. Common causes: CPU utilisation too high, since Google recommends keeping it under about 65 percent for regional and 45 percent for multi-region instances, so add nodes or processing units; hotspots, where sequential keys like timestamps or auto-increment IDs concentrate writes on one split, which Key Visualizer shows as bright bands, fixed by using UUIDs, hash-prefixed or bit-reversed keys; poor queries doing full table scans, missing secondary indexes, or inefficient joins, which you inspect with query execution plans and fix by adding indexes or using hints; long read-write transactions holding locks and causing aborts and retries, solved by shorter transactions, read-only transactions, and stale reads where acceptable; and large result sets or many round trips from the client. Also check geographic distance between the app and the instance's leader region, multi-region commit latency, and session pool configuration. Use tools such as query statistics tables and lock statistics, and load test changes. Plan capacity and add autoscaling to manage load.

Likely follow-ups

- What does a hotspot look like in Key Visualizer?
- Why do sequential primary keys hurt Spanner?

```
gcloud spanner databases execute-sql <db> --instance=<inst> --sql='SELECT * FROM
SPANNER_SYS.QUERY_STATS_TOP_HOUR ORDER BY avg_latency_seconds DESC LIMIT 10'
```

```
gcloud spanner instances describe <inst> # node count
```

Spanner latency causes: hot spots from monotonically increasing keys (use UUID or bit-reversed sequences as primary keys), insufficient nodes (scale up processing units), large read/write transactions, missing secondary indexes causing full scans, or high-priority CPU above 65%. Use Query Execution Plans to identify slow operations.

Hot spot pattern: Using timestamps or auto-increment IDs as Spanner primary keys creates hot spots on the last split. Always use UUID v4 or hash-prefixed keys. This is the most common Spanner performance mistake.

Official docs

- [Cloud Spanner](#)

Q19 Cloud SQL instance is not accepting connections — how do you debug?

Storage & data

Spoken answer · ~87 s

For a Cloud SQL instance refusing connections, I check the instance, then the path, then the credentials. Instance state: is it running, not in maintenance, failover, or stopped, and are there storage-full or max-connections conditions? Check the Cloud SQL logs and metrics for connection counts versus the max_connections setting. Connectivity depends on how you connect. Public IP: the client's address must be in authorised networks, SSL requirements must be satisfied, and the instance must have a public IP at all. Private IP: the client must be in the connected VPC or a peered network, with private services access set up, firewall rules allowing the database port, and routes working from on-premises if relevant. The recommended method is the Cloud SQL Auth Proxy or language connectors, which handle IAM-based authorisation and encryption over port 3307, so check the proxy logs, the instance connection name, and that the service account has the Cloud SQL Client role. Credentials: correct user, password or IAM database authentication, and host restrictions for MySQL users. Also consider organisation policies restricting public IP, DNS problems, exhausted connection pools in the application, and failover causing temporary refusal. Test with the proxy and psql or mysql from a VM in the same VPC to isolate network from application issues.

Likely follow-ups

- Why is the Cloud SQL Auth Proxy recommended?
- How would you tell if you've hit max_connections?

```
gcloud sql instances describe <instance> | grep -E "state|ipAddresses"
```

```
gcloud sql connect <instance> --user=root # direct test
```

```
./cloud_sql_proxy -instances=<project>:<region>:<instance>=tcp:5432
```

Connection issues: Cloud SQL Auth Proxy not running (recommended connection method — handles IAM auth and TLS), authorized networks not including the client IP (for public IP instances), SA missing roles/cloudsql.client, instance is stopped or in maintenance, max connections reached. Use PgBouncer or Cloud SQL's built-in connection pooling for high-concurrency apps.

GKE pattern: Run Cloud SQL Auth Proxy as a sidecar container in each pod, or as a Deployment with a ClusterIP Service. The sidecar pattern is simpler; the shared proxy pattern reduces total connections to Cloud SQL.

Official docs

- [Cloud SQL](#)

Q20 Compute Engine VM is unreachable via SSH — how do you recover access?

Compute

Spoken answer · ~88 s

If SSH to a Compute Engine VM fails, I first determine whether it's a network, authentication, or guest problem. Check the VM is running and look at the serial console output for boot problems, enabled via the interactive serial console if permitted. Network: the firewall must allow TCP 22 from your source range, the default-allow-ssh rule may have been removed; for VMs without external IPs, use Identity-Aware Proxy tunnelling, whose source range 35.235.240.0/20 must be allowed, or a bastion or Cloud VPN. Run Connectivity Tests. Authentication: with OS Login, check your IAM roles such as Compute OS Login or OS Admin Login, and 2FA; with metadata-based keys, check the key is in project or instance metadata and the format is correct. Guest issues: sshd not running or misconfigured, a full disk, a corrupted authorized_keys file, high load, or a broken firewall inside the OS. Recovery options: use the serial console to log in with a local account, run a startup script via metadata to fix sshd or add a user, or detach the boot disk and attach it as a secondary disk to a rescue VM to repair files, then reattach. As a last resort, snapshot the disk and rebuild. Afterwards, standardise on OS Login and IAP, and monitor the agent.

Likely follow-ups

- How do you recover a VM when SSH is completely broken?
- Why use IAP for SSH?

```
gcloud compute instances describe <vm> --zone=<zone> | grep -E "status|networkInterfaces"
```

```
gcloud compute instances get-serial-port-output <vm> --zone=<zone> # boot/OS logs
```

```
gcloud compute ssh <vm> --zone=<zone> --tunnel-through-iap # SSH via IAP (no public IP needed)
```

Recovery options: IAP TCP tunneling bypasses firewall rules and public IP requirement — just needs `roles/iap.tunnelResourceAccessor`. For SSH key issues: use OS Login (recommended) which ties SSH access to GCP IAM identity. Serial console output shows OS-level boot errors. For disk issues, detach disk, attach to recovery VM, repair, reattach.

OS Login vs metadata SSH keys: OS Login is the recommended approach — maps Linux user accounts to Google identities, respects IAM, audit logs every login. Metadata SSH keys are unmanaged and hard to audit at scale.

Official docs

- [Compute Engine: Managed instance groups](#)

Q21 Cloud Run service is returning 500 errors — how do you debug?

Compute

Spoken answer · ~84 s

For Cloud Run returning 500 errors, I look at the logs and the revision first. Cloud Logging shows the request logs with the status codes and the container's stdout and stderr, where stack traces usually appear. Determine whether the 500 comes from the application or from Cloud Run itself: a 500 with application logs is a code problem; a 503 or a 'container failed to start' message is a platform-level problem. Common platform causes: the container doesn't listen on the port in the `PORT` environment variable, or on `0.0.0.0`, so startup fails; startup takes too long and the probe times out; the container crashes due to out-of-memory, visible in a memory limit exceeded log, so raise memory or fix leaks; request timeout exceeded; CPU being throttled outside request processing for background work; max instances limit reached, leading to queued requests and 429s; or cold start latency. Application causes: dependency failures, such as database connection limits, missing environment variables or secrets, wrong IAM permissions for the service account calling other services, VPC connector or egress misconfiguration, and bad deployments. Compare the latest revision with the previous one, and roll traffic back to the last good revision immediately if a release is at fault.

Likely follow-ups

- How do you tell platform errors from application errors?
- What is your fastest mitigation after a bad release?

```
gcloud run services describe <service> --region=<region> # configuration
```

```
gcloud logging read 'resource.type="cloud_run_revision" severity=ERROR' --limit=20
```

```
gcloud run services update-traffic <service> --to-revisions=LATEST=0,<prev-rev>=100 --region=<region> # rollback traffic
```

Cloud Run 500 causes: application panic/crash (check container logs), container fails to start within 4 minutes (startup probe/timeout), container port mismatch (must listen on \$PORT env var), exceeded concurrency limit causing queue buildup, cold start timeout, or dependency unavailable. Roll back traffic to previous revision while debugging.

PORT env var: Cloud Run injects the \$PORT environment variable. Your container must listen on this port — hardcoding port 8080 works only if you explicitly set it. Many teams hardcode the port and then are confused when the service returns 502.

Official docs

- [Kubernetes: Debug Services](#)
- [Cloud Run](#)

Q22 VM instance group is not autoscaling as expected — how do you debug MIG autoscaling?

Compute

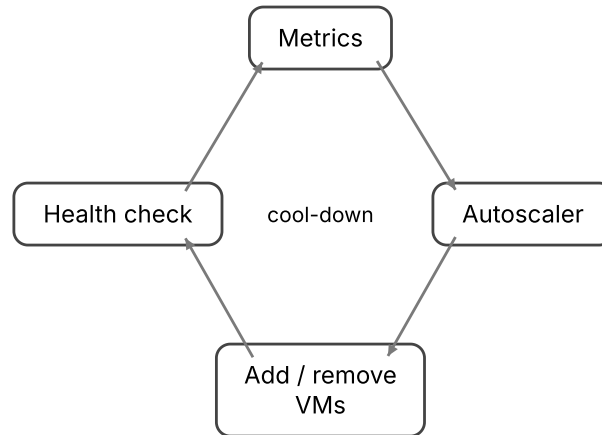
Spoken answer · ~88 s

A managed instance group autoscaler decides the number of instances from a utilisation target, so I first confirm what it's configured to do and what the data says. Check the autoscaling policy: minimum and maximum instances, the metric used, such as average CPU utilisation, load balancing utilisation, or a custom Cloud Monitoring metric, and the target value. Reasons it doesn't scale: the metric isn't reaching the target, because the metric averages across all instances; the instances are already at the maximum; the cool-down period, which is the time instances need to initialise, delays decisions, and a long period or slow startup means new VMs are not counted yet; the scale-in controls limit how quickly it removes instances; the instance health check is failing, so instances are being recreated; quota or capacity limits in the zone or region prevent creation; the template has errors such as an invalid image or missing permissions; or metrics are missing for custom metrics, in which case the autoscaler may do nothing. Check the MIG's autoscaler status and errors in the console, the instance group's activity in Cloud Logging, and the chosen metric in Monitoring. Fixes: tune the target, use predictive autoscaling, shorten initialisation, request more quota, use multiple zones, and choose a metric that reflects real load.

Likely follow-ups

- What does the cool-down period do?
- Why might scale-in be slower than scale-out?

🚧 MIG autoscaling



```
gcloud compute instance-groups managed describe <mig> --zone=<zone> # autoscaler config
```

```
gcloud compute instance-groups managed list-instances <mig> --zone=<zone>
```

```
gcloud compute instance-groups managed describe-autoscaler <mig> --zone=<zone> # scaling decisions
```

Autoscaler decisions are based on: CPU utilization, HTTP LB serving capacity, Cloud Monitoring metrics, or schedule. Debug: check autoscaler status (ACTIVE/ERROR), verify metric is being emitted, check min/max instance bounds, and confirm instance template is valid. Cooldown period (default 60s) prevents rapid scale oscillation.

Predictive autoscaling: GCP MIG autoscaler supports predictive mode — it learns traffic patterns and scales proactively before load spikes. Enable with `--autoscaling-mode=ONLY_SCALE_OUT` initially to avoid unexpected scale-in.

📖 Official docs

- [Compute Engine: Managed instance groups](#)

Q23 **How do you set up comprehensive observability in GCP — metrics, logs, and traces?**

Observability

 Spoken answer · ~81 s

On GCP, comprehensive observability uses the Cloud Operations suite plus open standards. Metrics: Cloud Monitoring collects system metrics from Google services automatically; for applications and GKE, use Managed Service for Prometheus or the Ops Agent on VMs, and custom metrics from OpenTelemetry. Build dashboards, define SLOs on services, and alert on symptoms and burn rates. Logs: Cloud Logging captures platform and application logs; emit structured JSON, include trace IDs, use sinks to route to BigQuery or Cloud Storage for retention and analysis, apply exclusion filters to control cost, and create log-based metrics for important events. Traces: instrument applications with OpenTelemetry and export to Cloud Trace, with sampling configured, and propagate context with W3C headers. Link them: logs with trace and span IDs appear alongside traces, and Error Reporting groups exceptions, with Cloud Profiler adding continuous profiling for CPU and memory. Add uptime checks and synthetic monitoring for external availability. Organise with metrics scopes to view multiple projects together. Notification channels send alerts to PagerDuty, Slack, or email, with runbook links. Use Terraform for dashboards and alert policies. Control cost through retention settings and sampling, and set IAM so teams see only what they should.

Likely follow-ups

- How do you view several projects in one monitoring scope?
- How do logs get linked to traces?

```
gcloud monitoring alerting-policies list    # existing alerts
```

```
gcloud logging sinks list                  # log routing destinations
```

```
gcloud trace list --project=<project>    # recent traces
```

Key setup: install Ops Agent on VMs for system metrics and logs, configure log-based metrics for business events, create alerting policies on SLO burn rates (not just thresholds), export logs to BigQuery for long-term analysis, and use Cloud Trace with OpenTelemetry SDK for distributed tracing. Create Uptime Checks for external endpoint monitoring.

SLO-based alerting: Alert on error budget burn rate, not raw error rate. GCP Cloud Monitoring supports SLO burn rate alerting natively — use it over simple threshold alerts.

 Official docs

- [OpenTelemetry: Signals \(traces, metrics, logs\)](#).

Q24 Cloud Logging is missing expected logs — how do you investigate log routing issues?

Observability

 Spoken answer · ~82 s

When expected logs aren't appearing, I trace the path: generation, collection, routing, and viewing. First, is the application writing to stdout or stderr, or to a file that isn't collected? On VMs, is the Ops Agent installed, running, and configured with the right log source, and does its service account have the Logs Writer role? On GKE, check that logging is enabled for workloads and that the logging agent is healthy. Then routing: Cloud Logging uses sinks; the _Default sink sends most logs to the _Default bucket, but exclusion filters on sinks can drop entries, and organisation or folder-level sinks may intercept logs; review all sinks and their inclusion and exclusion filters. Check the log bucket retention, the project and region, and whether log views restrict what you see. Check that the query is right: time range, resource type, severity, and the log name; some logs, like Data Access audit logs, are disabled by default and must be enabled. Look for quotas and rate limits: entries too large or too many requests can be dropped, so check the logging metrics for dropped entries. Permissions are another cause: the viewer needs Logs Viewer on the right project.

Likely follow-ups

- Which audit logs are not enabled by default?
- How can an exclusion filter hide logs?

```
gcloud logging sinks list --project=<project> # log sinks and filters
```

```
gcloud logging read 'resource.type="gce_instance"' --limit=5 # test query
```

```
gcloud logging write test-log "test message" --severity=INFO # inject test log
```

Missing logs causes: log exclusion filter on the default sink dropping logs before they reach Log Bucket, application writing to stdout but log agent not configured, resource not exporting logs to Cloud Logging, log retention period expired (default 30 days), or log sink misconfigured. Check the _Default sink's exclusion filters first.

Cost vs retention: Logs are expensive at scale. Use log exclusions to drop high-volume low-value logs. Route important logs to BigQuery for long-term analysis. Set log-based alerts for critical errors instead of keeping all logs hot.

Official docs

- [Cloud Logging](#)

Q25 How do you create custom dashboards and alerting policies in Cloud Monitoring?

Observability

 Spoken answer · ~85 s

In Cloud Monitoring, custom dashboards are built from charts using the Metrics Explorer, with the metric chosen from system, agent, Prometheus, or log-based metrics, filtered and grouped by labels, aggregated by function such as rate or percentile, and arranged into widgets. You can use the query builder, MQL, or PromQL, add scorecards, gauges, tables, and logs panels, and use dashboard variables for filters. Create them in the console for exploration, then manage final versions as code in JSON with Terraform or the API, so they're versioned and consistent across projects. Alerting policies define a condition, notification channels, and documentation.

Condition types: a metric threshold, such as latency above a value for five minutes; metric absence, when data stops arriving; forecasted breach; log-based conditions on specific log entries; and SLO burn rate for services with defined SLOs. Set the alignment period and duration to avoid flapping, and group by resource. Notification channels include email, Slack, PagerDuty, SMS, Pub/Sub, and webhooks. Document each alert with what it means and a runbook link, using variables for context. Set severity, and use snooze for maintenance. Alert on user-facing symptoms, tune thresholds from real baselines, and review noisy alerts regularly. Uptime checks and synthetic monitors add external visibility.

Likely follow-ups

- When would you use a metric-absence condition?
- How do you manage alert policies as code?

```
gcloud monitoring dashboards create --config-from-file=dashboard.json
```

```
gcloud monitoring alerting-policies create --policy-from-file=alert.yaml
```

```
kubernetes.io/container/cpu/request_utilization # pod CPU vs request
```

```
loadbalancing.googleapis.com/https/backend_latencies # LB latency
```

Build dashboards around golden signals: latency (p50/p99), traffic (RPS), errors (5xx rate), saturation (CPU/memory). Use Monitoring Query Language (MQL) for complex aggregations. Set up notification channels (PagerDuty, OpsGenie, Slack) before creating alerts.

Dashboard as code: Export dashboard JSON, store in Git, deploy via Terraform `google_monitoring_dashboard` resource. All observability config should be IaC — dashboards deleted in UI get recreated automatically.

Official docs

- [Cloud Monitoring](#)
- [Grafana documentation](#)

 Spoken answer · ~84 s

VPC Service Controls creates a security perimeter around Google Cloud resources and services to reduce the risk of data exfiltration, even from callers with valid IAM credentials. A perimeter contains projects and the restricted services, such as Cloud Storage, BigQuery, and others; API calls crossing the perimeter boundary are blocked unless allowed. This protects against stolen credentials, misconfigured IAM, or an insider copying data to a bucket outside the perimeter. Access levels, defined in Access Context Manager, allow requests from certain IP ranges, device policies, or identities. Ingress and egress rules define controlled exceptions for specific identities, resources, and methods, and perimeter bridges or rules connect perimeters. Configuring it: define the access policy at organisation level, identify the projects and services to protect, create the perimeter in dry-run mode first, review violation logs in Cloud Audit Logs to find legitimate flows that would break, add the necessary access levels and rules, and then enforce. Pay attention to Cloud Build, CI systems, Google-managed services, on-premises access through Private Google Access and restricted VIPs, and users who need console access. Roll out gradually and monitor. It adds defence in depth on top of IAM, but needs careful design because it can break integrations.

Likely follow-ups

- Why start in dry-run mode?
- How does VPC Service Controls differ from IAM?

```
gcloud access-context-manager perimeters list --policy=<policy>
```

```
gcloud access-context-manager perimeters create <name> --policy=<policy> --  
resources=projects/<num> --restricted-services=storage.googleapis.com
```

VPC Service Controls creates a security perimeter around GCP services — requests from outside the perimeter are denied even with valid IAM credentials. Use for: data exfiltration prevention, compliance (HIPAA, PCI-DSS), restricting BigQuery/GCS to internal access only. Configure access levels for exceptions. Dry-run mode logs violations without enforcing — use this first to identify legitimate traffic to allowlist.

Breakage risk: VPC-SC can break service integrations. Always run in dry-run mode for 2+ weeks before enforcing. Audit the violation logs thoroughly. A wrong VPC-SC config can cause widespread service outages.

 Start here (official docs)

- [Google Cloud documentation](#)

Q27 Security Command Center is showing high-severity findings — how do you triage and remediate?

Security

 Spoken answer · ~85 s

Security Command Center aggregates findings from vulnerability, threat, and misconfiguration detectors, so triage is about prioritising and routing. First, filter to high-severity findings and group by category, resource, and project. Assess each finding's real risk using context: is the resource internet-facing, does it contain sensitive data, and is there an attack path, which SCC Premium's attack exposure scores and attack path simulation highlight? Distinguish threats, such as active exploitation or anomalous behaviour from Event Threat Detection, which need immediate incident response, from posture issues, like a public bucket or open firewall, which need prompt remediation, and vulnerabilities in images or VMs, which go to patch workflows. For urgent threats: contain by isolating the resource, revoking credentials, preserving evidence, and following the incident process. For misconfigurations: identify the owner through labels and project ownership, create a ticket with the details and a deadline, and fix the root cause, ideally in Terraform or via an organisation policy so it can't recur. Use mute rules for accepted risks or false positives, with documented justification. Automate: export findings through Pub/Sub to a SIEM or ticketing system, and use Cloud Functions for auto-remediation of well-understood cases. Track metrics such as time to remediate, and review trends with the teams.

Likely follow-ups

- How do you tell a threat from a posture issue?
- What belongs in automated remediation?

```
gcloud scc findings list --organization=<org> --filter="severity=HIGH AND state=ACTIVE" --format=json
```

```
gcloud scc findings update <finding> --organization=<org> --state=INACTIVE # mark resolved
```

SCC finding categories: vulnerabilities (CVEs in container images), misconfigurations (public GCS buckets, open firewall rules, SA key age), threats (anomalous API calls, cryptomining, data exfiltration), and compliance violations. Prioritize: CRITICAL > HIGH, then by asset criticality. Integrate SCC with SIEM for SOC workflows.

Quick wins: SCC often flags public GCS buckets, overly permissive firewall rules (allow all ingress), and stale SA keys (>90 days). Fix these first — they're high impact, low effort. Set up SCC notifications to PagerDuty for CRITICAL findings.

Official docs

- [Security Command Center](#)

Q28 How do you implement Binary Authorization for GKE — ensuring only verified images run in production?

Security

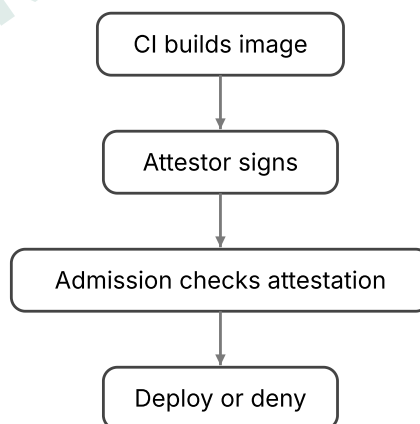
🗣 Spoken answer · ~86 s

Binary Authorization is a deploy-time control that ensures only trusted container images run on GKE. Enable the Binary Authorization API and turn on enforcement for the cluster. Define a policy specifying the default rule and cluster-specific rules: allow all, deny all, or require attestations from named attestors. An attestor is an identity that verifies a claim about an image, backed by a public key from Cloud KMS. In the pipeline, after the build, tests, and vulnerability scans pass, a step creates an attestation, a signature over the image digest, and stores it. Images are referenced by digest, not tag, so the signature applies to exact content. At deployment, the admission controller checks the policy: if the image has the required attestations, it's admitted; otherwise it's rejected. Policies can allow-list system images or trusted registries via exemption patterns. Use dry-run mode first, which logs violations without blocking, then switch to enforced. Provide an emergency break-glass mechanism using an annotation, with audit logging and alerts. Combine with Artifact Registry vulnerability scanning, so an attestation is only created when there are no critical findings, and with build provenance from Cloud Build. Continuous validation re-checks running pods against the policy. Separate who builds from who attests, and protect the signing keys.

Likely follow-ups

- Why use digests instead of tags with Binary Authorization?
- What is the break-glass annotation for?

🔧 Binary Authorization



```
gcloud container binauthz policy export > policy.yaml # export current policy
```

```
gcloud container binauthz attestations sign-and-create --artifact-url=<digest-url> --  
attestor=<attestor> --keyversion=<kms-key>
```

```
gcloud container clusters update <cluster> --binauthz-evaluation-  
mode=PROJECT_SINGLETON_POLICY_ENFORCE
```

Binary Authorization enforces that only cryptographically attested images run on GKE. CI pipeline: build image → scan with Trivy/Container Analysis → if no critical CVEs, sign the image digest with KMS → attestation stored. GKE admission controller verifies attestation before allowing pod creation. Unsigned images are rejected.

For Interview context: Binary Authorization is the gold standard for supply chain security in GKE. Being able to describe this end-to-end at a Principal level demonstrates mature GCP security thinking.

Official docs

- [Binary Authorization](#)
- [GKE overview](#)

Q29 A GCP service account key was accidentally committed to GitHub — what do you do?

Security

Spoken answer · ~84 s

Treat it as a compromised credential immediately, since scanners find exposed keys within minutes. First, disable the key right away in IAM so it cannot be used, then delete it once confirmed safe; disabling is quick and reversible. Don't just delete the commit: the key remains in Git history, forks, and caches, so assume it's compromised. Next, assess the blast radius. Review Cloud Audit Logs for the service account, filtering by the key ID, to see calls made with it, from which IP addresses, and look for unfamiliar activity, new resources such as crypto-mining instances, new keys, IAM changes, or data access. Check what permissions the account had, as that bounds the possible damage. Remediate anything suspicious: delete unauthorised resources, revoke created credentials, rotate secrets accessible to the account, and review billing. Then fix the cause: remove the secret from history using git filter-repo or BFG and force push, though this is secondary to revocation; enable GitHub secret scanning and push protection; add pre-commit scanners. Prevent recurrence: avoid service account keys entirely using Workload Identity Federation or impersonation; enforce the organisation policy to disable key creation; apply least privilege. Notify the security team, document the incident, and monitor for further activity.

Likely follow-ups

- Why is deleting the commit not enough?
- How do you find out what the key was used for?

```
gcloud iam service-accounts keys disable <key-id> --iam-account=<sa> # disable immediately
```

```
gcloud iam service-accounts keys delete <key-id> --iam-account=<sa> # then delete
```

```
gcloud logging read 'protoPayload.authenticationInfo.serviceAccountKeyName="<key-id>"' --limit=50 # audit usage
```

Incident response: (1) Disable the key immediately. (2) Audit Cloud Audit Logs for any usage of the compromised key. (3) Delete the key. (4) Rotate all secrets that the SA had access to. (5) Remove from Git history and force-push. (6) Enable Secret Scanning in GitHub. (7) Migrate to Workload Identity to eliminate keys entirely.

Never rely on Git history deletion alone. Assume the key was already scraped by automated bots. The key is compromised — disable it first, investigate second.

Official docs

- [Kubernetes: Service accounts](#)
- [Google Cloud: Workload Identity Federation](#)
- [Google Cloud: Best practices for service accounts](#)

Q30 How do you implement a production-grade CI/CD pipeline on GCP using Cloud Build?

CI/CD & IaC

Spoken answer · ~83 s

A production-grade Cloud Build pipeline is defined in `cloudbuild.yaml` and triggered from the repository: pull request triggers run validation, and merges to main run the full release. Steps: run linting and unit tests; build the container with caching; scan for vulnerabilities; push to Artifact Registry tagged with the commit SHA; and generate provenance. Use a dedicated, least-privilege user-specified service account for builds, not the default, and fetch secrets from Secret Manager. Run on private pools if builds need VPC access. For deployment, either apply manifests to GKE via GitOps by updating a config repository, which Argo CD or Config Sync reconciles, or use Cloud Deploy to manage delivery pipelines across dev, staging, and production targets, with approvals, canary or blue-green strategies, automated verification, and rollbacks. Enforce Binary Authorization so only attested images deploy. Promote the same immutable image digest through environments, with environment-specific configuration separate. Add post-deployment smoke tests and monitoring with automated rollback on alert. Control access: restrict who can approve production releases, and who can edit triggers. Optimise with parallel steps and caching, filter triggers by path, and set timeouts. Keep builds reproducible, store logs centrally, and measure DORA metrics. Manage pipelines and triggers as code with Terraform.

Likely follow-ups

- What does Cloud Deploy add beyond Cloud Build?
- Why use a dedicated build service account?

```
steps:
```

```
- name: gcr.io/cloud-builders/docker
```

```
  args: [build, -t, IMAGE, .]
```

```
- name: gcr.io/cloud-builders/gke-deploy
```

```
  args: [run, --filename=k8s/, --cluster=<c>, --location=<z>]
```

```
gcloud builds triggers create github --repo-name=<repo> --branch-pattern=main --build-  
config=cloudbuild.yaml
```

Production pipeline: PR trigger → lint + test → build image → push to Artifact Registry → scan with Container Analysis → run integration tests in staging → deploy to staging → gate on test results → deploy to production. Use Cloud Build substitution variables for environment-specific config. Separate SA per environment with minimal permissions.

GitOps alternative: Use Cloud Build to update the image tag in a GitOps repo → ArgoCD picks up the change and syncs to GKE. This separates CI (Cloud Build) from CD (ArgoCD) cleanly — the recommended pattern for production GKE deployments.



Official docs

- [Cloud Build](#)

Q31 **Terraform is being used to manage GCP infrastructure — what GCP-specific best practices apply?**

CI/CD & IaC

 Spoken answer · ~83 s

For Terraform on GCP, several practices matter. State: store it remotely in a versioned Cloud Storage bucket with locking, restrict access tightly, and separate state per environment or component to limit blast radius. Authentication: use service account impersonation or Workload Identity Federation from CI, never downloaded keys, with a dedicated deploy service account of minimal permissions. Structure: use modules for repeatable patterns, such as the Cloud Foundation Toolkit, pinned provider and module versions, and separate environments by directory or workspace with different projects. Project and organisation: manage the folder hierarchy, projects, IAM, and organisation policies as code. Be careful with IAM resources: `google_project_iam_member` is additive and safe, `google_project_iam_binding` is authoritative per role, and `google_project_iam_policy` is authoritative for everything and can lock you out. Handle API enablement explicitly with `google_project_service`. Use labels consistently for cost allocation. Watch for resources with long creation times, eventual consistency, and fields that force replacement, reviewing plans carefully. Use lifecycle `prevent_destroy` for critical resources such as databases and buckets. Run plan in pull requests, apply from the pipeline after approval, add policy checks with Sentinel or OPA, scan with tfsec, and detect drift on a schedule. Never put secrets in state or variables in plain text.

Likely follow-ups

- Why are authoritative IAM resources dangerous?
- How do you reduce the blast radius of a bad apply?

```
terraform {
```

```
  backend "gcs" {
```

```
    bucket = "tf-state-<project>"
```

```
    prefix = "prod"
```

```
  }
```

```
}
```

```
gcloud iam roles create terraform_deployer --project=<project> --  
permissions=compute.instances.create,...
```

GCP-specific Terraform best practices: store state in GCS with object versioning. Use Workload Identity for CI/CD — no SA key files. Enable GCS state locking (native, no DynamoDB equivalent needed). Organize by resource hierarchy: one root module per project/environment. Use `google_project_service` to manage API enablement.

API enablement: Add `google_project_service` resources for every API your Terraform uses, with `disable_on_destroy = false` to avoid breaking dependent resources during teardown.

 **Start here (official docs)**

- [Google Cloud documentation](#)

Q32 How do you manage GCP project structure and resource hierarchy at an enterprise scale?

CI/CD & IaC

 Spoken answer · ~85 s

At enterprise scale, the resource hierarchy is organisation, folders, projects, and resources, and it's the foundation for IAM inheritance, organisation policies, billing, and governance. Design principles: the organisation node is tied to the Cloud Identity or Workspace domain; folders group projects by function and environment, a common pattern being top-level folders for production, non-production, and shared services, or by business unit with environments below; avoid deeply nested trees since policies inherit and complexity grows. Projects are the unit of isolation, quota, billing, and API enablement, so use separate projects per application and environment, and separate projects for shared networking via Shared VPC host, logging, monitoring, security tooling, and billing exports. Apply IAM to groups at folder level and organisation policies for guardrails such as restricting regions, disabling key creation, and enforcing uniform bucket access. Use labels and tags for cost allocation and policy conditions. Automate the structure with Terraform and a landing zone blueprint, like the Cloud Foundation Toolkit, and a project factory so new projects are created consistently with budgets, labels, APIs, and network attachments. Plan break-glass access, centralised audit log sinks, and naming conventions. Review the hierarchy as the organisation grows, and avoid overly granular folders that mirror the org chart too tightly.

Likely follow-ups

- Why separate projects per environment?
- What is a project factory?

```
gcloud resource-manager folders list --organization=<org-id>
```

```
gcloud organizations get-iam-policy <org-id> # org-wide policies
```

```
gcloud org-policies set-policy --organization=<org-id>  
constraints/storage.publicAccessPrevention.yaml
```

Recommended structure: Org → [Bootstrap, Common, Environments] folders → [network, security, prod, staging, dev] projects. One project per environment per application (project as blast radius boundary). Shared VPC host project for networking. Use Organization Policies (constraints) for guardrails: disable SA key creation, enforce resource location, require OS Login.

Landing Zone pattern: Use Google's Cloud Foundation Toolkit (Terraform modules) for bootstrapping the org hierarchy. Don't design from scratch — the CFT encodes GCP best practices accumulated from thousands of enterprise deployments.

 Official docs

- [Google Cloud: Resource hierarchy](#)

Q33 **GCP bill is unexpectedly high — how do you investigate and optimize costs?**

Cost

 Spoken answer · ~82 s

An unexpected bill needs a quick, structured investigation. First look at the Billing reports in the console, filtering by project, service, SKU, and location, and compare with the previous period to find which service jumped. For detail, export billing data to BigQuery and query by labels, SKU, and day. Check the usual suspects: egress and network charges, especially cross-region or internet traffic; BigQuery scanning; idle or oversized Compute Engine VMs and disks; forgotten resources like unattached disks, old snapshots, and unused IPs; GKE node pool growth or autoscaling gone wrong; logging and monitoring ingestion volume; Cloud Storage operations and retrieval fees; and potential compromise, such as crypto-mining VMs created with a leaked credential. Verify against Cloud Audit Logs to see who created what. Immediate actions: shut down or scale down waste, stop runaway jobs, and set quotas. Optimisation: right-size using recommenders, use committed use discounts and spot VMs where fit, schedule dev environments off outside work hours, apply storage lifecycle rules, reduce data movement, and tune log retention. Prevent surprises: set budgets with alert thresholds, anomaly detection alerts, label resources consistently for showback, and review cost reports regularly with owners. Apply programmatic budget actions only with care.

Likely follow-ups

- Where would you look first for a sudden cost spike?
- How would you detect a compromised project through billing?

```
gcloud billing budgets list --billing-account=<account> # existing budget alerts
```

```
SELECT service.description, SUM(cost) as total
```

```
FROM `billing_dataset.gcp_billing_export`
```

```
WHERE DATE(usage_start_time) = CURRENT_DATE()-1
```

```
GROUP BY 1 ORDER BY 2 DESC
```

Investigation: export billing data to BigQuery (enable billing export first), query by service/project/label/SKU. Common culprits: undeleted persistent disks after VM deletion, unused static IPs, GKE nodes with low utilization, inter-region egress traffic, BigQuery on-demand scan costs. Use committed use discounts (CUDs)

— 1-year CUD saves ~37%, 3-year ~55%.

Label-based cost attribution: Tag all resources with team, env, application labels. Filter billing export by label to charge back costs to teams. Without labels, cost attribution becomes a guessing game at scale.

 **Start here (official docs)**

- [Google Cloud documentation](#)

Q34 GKE cluster costs are too high — what optimization strategies do you apply?

Cost

 Spoken answer · ~75 s

GKE cost optimisation starts with visibility, then right-sizing, then purchasing choices. Visibility: enable GKE cost allocation and usage metering, label namespaces and workloads, and review per-team costs in the billing export. Right-size workloads: set resource requests based on actual usage, using the Vertical Pod Autoscaler recommendations, because oversized requests waste node capacity; remove idle or over-provisioned deployments. Scale efficiently: use the Horizontal Pod Autoscaler, the cluster autoscaler with an optimise-utilisation profile, and node auto-provisioning so nodes fit pods; consider Autopilot, which bills per pod resource requests, for workloads where operations overhead matters. Choose the right machine types and bin pack well, perhaps using E2 or Arm-based Tau instances where compatible. Use Spot VMs for fault-tolerant, stateless, batch workloads, with node taints and graceful termination handling, and committed use discounts for the steady baseline. Reduce other costs: scale down non-production clusters overnight, consolidate small clusters, use regional storage appropriately, reduce cross-zone traffic with topology-aware routing, tune logging and monitoring volume, and clean unused persistent disks and load balancers. Set budgets and alerts, and review utilisation dashboards regularly, as optimisation is continuous.

Likely follow-ups

- What workloads suit Spot VMs?
- How does oversized requests waste money?

```
kubectl top pods -A --sort-by=memory # find over-provisioned pods
```

```
gcloud container node-pools create spot-pool --cluster=<cluster> --spot --machine-type=e2-standard-4
```

GKE cost optimization: (1) Spot/Preemptible VMs for batch/stateless workloads (60-90% cheaper), (2) Right-size node pools using VPA recommendations, (3) Enable cluster autoscaler to scale down idle nodes, (4) Use E2 machine family (cost-optimized), (5) Schedule dev/staging clusters to scale to zero overnight, (6) Enable GKE usage metering to attribute costs per namespace/team.

Autopilot cost model: GKE Autopilot charges per pod CPU/memory request — no idle node cost. For variable workloads with low off-peak usage, Autopilot is often significantly cheaper than Standard mode

clusters with persistent node pools.

Official docs

- [GKE overview](#)

Q35 How do you implement cost governance and FinOps practices on GCP?

Cost

Spoken answer · ~83 s

FinOps is the practice of bringing financial accountability to cloud spending through collaboration between engineering, finance, and business. On GCP I'd work across three phases: inform, optimise, and operate. Inform: enforce consistent labels and project structure for cost allocation; export billing data to BigQuery and build dashboards in Looker Studio showing cost by team, service, and environment; provide showback or chargeback so teams see their own spend; and track unit economics, like cost per customer or per transaction, not just total spend. Optimise: act on recommenders for idle and oversized resources; use committed use discounts and Spot VMs; apply storage lifecycle policies; and tune architecture for cost. Operate: set budgets with alerts per project and team, anomaly detection, and quotas; add guardrails with organisation policies, such as restricting expensive machine types or regions; include cost estimates in pull requests with Terraform tooling like Infracost; and hold regular reviews with owners. Establish governance: a cloud centre of excellence, clear ownership, tagging policy with enforcement, and a process for approving large spend. Culture matters most: make costs visible, celebrate savings, and give engineers the tools and responsibility. Measure progress with metrics such as percentage of spend allocated, commitment coverage, and waste reduction.

Likely follow-ups

- What is the difference between showback and chargeback?
- What unit-cost metric would you track?

```
gcloud billing budgets create --billing-account=<account> --display-name="Prod Budget" --
budget-amount=5000USD --threshold-rule=percent=0.5 --threshold-rule=percent=0.9
```

```
gcloud recommender recommendations list --project=<project> --
recommender=google.compute.instance.MachineTypeRecommender --location=<region>
```

FinOps on GCP: (1) Budget alerts per project/team, (2) Billing export to BigQuery + Looker Studio dashboards for real-time spend visibility, (3) Enforce resource labels via Organization Policy, (4) Use GCP Recommender API for automatic right-sizing suggestions, (5) Committed Use Discounts for baseline compute, (6) Monthly cost review with team leads.

Principal-level ownership: A Platform Engineer builds the cost visibility tooling and governance framework. Attribute costs to teams, surface anomalies automatically, make the data self-serve. The goal is teams making cost-aware decisions independently.

Official docs

- [Cloud Billing budgets and alerts](#)
- [AWS Well-Architected: Cost optimization pillar](#)

Q36 How do you manage Organization Policies to enforce guardrails across all GCP projects?

IAM & identity

Spoken answer · ~87 s

Organisation Policies are centrally defined constraints that restrict how resources can be configured across the hierarchy, independent of IAM: IAM controls who can act, while organisation policies control what is allowed. They're set at the organisation, folder, or project level and inherited downward, with the option to merge with or override parent policies. Constraints are either boolean, such as disabling service account key creation or requiring uniform bucket-level access, or list-based, such as the allowed locations for resources, allowed external IPs, or permitted services. Useful guardrails: restrict resource locations for data residency; disable service account key creation and upload; prevent public IPs on VMs and Cloud SQL; skip default network creation; enforce shielded VMs; restrict domain-sharing in IAM to your own domain; and disable automatic IAM grants for default service accounts. Custom constraints let you define rules on specific resource fields using CEL. Rollout: start with dry-run mode or at a test folder, check the impact on existing resources since policies apply to new changes and not retroactively, document exceptions with a documented process using tags to scope them, and then enforce wider. Manage them in Terraform with review. Monitor violations with audit logs and Security Command Center, and review regularly. Grant the Organization Policy Administrator role sparingly.

Likely follow-ups

- How do organisation policies differ from IAM?
- How do you handle legitimate exceptions?

```
gcloud org-policies list --organization=<org-id> # all active policies
```

```
gcloud org-policies describe constraints/iam.disableServiceAccountKeyCreation --organization=<org-id>
```

```
gcloud org-policies set-policy policy.yaml --organization=<org-id>
```

Key Organization Policies for security: iam.disableServiceAccountKeyCreation (enforce Workload Identity), storage.publicAccessPrevention (no public GCS buckets), compute.requireOsLogin, compute.restrictCloudSQLInstances (private-only), gcp.resourceLocations (restrict to specific regions). Policies inherit downward and can be overridden at lower levels unless enforced.

Policy inheritance override: Use inheritFromParent: false with an empty allow list to enforce a deny-all

at org level that cannot be overridden by child resources.

Official docs

- [AWS Control Tower](#)
- [Google Cloud: Organization policy constraints](#)

Q37 Cloud Interconnect is experiencing packet loss — how do you diagnose hybrid connectivity issues?

Networking

Spoken answer · ~90 s

For packet loss on Cloud Interconnect, I split the path into segments and find where loss begins. Start with the Interconnect and VLAN attachment status in the console and Cloud Monitoring: link state, light levels for dedicated connections, errors and drops, utilisation, and BGP session status through Cloud Router. High utilisation is a common cause: if the attachment is saturated, packets are dropped, so check bandwidth against capacity and consider adding capacity or more attachments. Check for physical issues on dedicated links, such as faulty optics, dirty fibre, or LACP bundle problems, with the colocation provider or the partner provider, who can test their segment. Check MTU mismatches: VLAN attachments support 1440 or 1500 MTU, and a mismatch causes fragmentation or drops for larger packets; test with ping using don't-fragment sizes. Check BGP flaps, route advertisement problems causing asymmetric routing, and Cloud Router limits. Look at the on-premises side: router interface counters, QoS policies, firewalls or stateful devices that drop asymmetric flows, and CPU. Use mtr or traceroute and Connectivity Tests, and run iperf tests in both directions to quantify loss. Use redundancy per Google's 99.99 percent topology, with two connections in separate metro areas, so one fault doesn't cause an outage. Open a support case with data if the issue is on Google's side.

Likely follow-ups

- How does MTU mismatch cause packet loss?
- What topology gives 99.99 percent availability?

```
gcloud compute interconnects describe <interconnect> # operational state
```

```
gcloud compute interconnects attachments describe <attachment> --region=<region>
```

```
gcloud compute routers get-status <router> --region=<region> # BGP session status
```

Interconnect issues: BGP session down (check advertised routes and ASN config), VLAN attachment misconfiguration, physical layer issues (contact GCP Support), or route advertisement missing. For Partner Interconnect, the ISP manages the physical layer — open ticket with both GCP Support and the ISP simultaneously.

Redundancy: Always provision 2 VLAN attachments in different metropolitan areas for Dedicated

Interconnect (99.99% SLA requires this). A single attachment gives only 99.9% SLA.

Official docs

- [Cloud Interconnect](#)

Q38 How do you implement multi-cluster GKE architecture for high availability?

GKE

Spoken answer · ~85 s

A multi-cluster GKE architecture improves availability, isolates failures, and supports regional proximity. Design: run clusters in at least two regions, each regional with nodes spread across zones, so zone failure is tolerated within a cluster and region failure across clusters. Traffic: use a global external Application Load Balancer with multi-cluster Ingress or the Gateway API multi-cluster gateway, which routes users to the nearest healthy cluster using health checks and fails over automatically; for internal traffic, multi-cluster services or a service mesh like Cloud Service Mesh provides cross-cluster service discovery and mTLS. Configuration consistency: manage clusters as cattle with GitOps through Config Sync or Argo CD from a single repository, using fleet management to group clusters, apply policies uniformly with Policy Controller, and roll out changes progressively cluster by cluster. State is the hard part: stateless services replicate easily, but data needs a multi-region database such as Spanner, or a replicated Cloud SQL and an explicit failover plan; avoid cross-region latency on the hot path. Also plan for capacity so one region can absorb the other's load, DNS and certificates, observability aggregated across clusters, and regular failover testing. Control costs, since each cluster adds overhead, and weigh whether the availability gain justifies the complexity.

Likely follow-ups

- What makes state the hardest part of multi-cluster?
- How does multi-cluster Ingress route traffic?

```
gcloud container fleet memberships list    # GKE Fleet registered clusters
```

```
gcloud container fleet ingress enable --config-membership=<cluster> # Multi-cluster Ingress
```

```
gcloud container fleet multi-cluster-services enable
```

Multi-cluster patterns: Multi-cluster Ingress with a Global Load Balancer routes traffic to the nearest healthy cluster (active-active), GKE Fleet for unified management and policy enforcement across clusters, Config Sync (ACM) for GitOps-driven config consistency. Deploy stateless workloads active-active across regions; stateful workloads with primary-replica across zones within a region.

GKE Fleet: Registers clusters under a common management plane. Enables Fleet-wide policy enforcement, Config Sync across all clusters, unified observability, and multi-cluster service mesh (Anthos Service Mesh). This is the production pattern for 5+ cluster environments.

Q39 How do you implement a secure software supply chain on GCP?

Security

 Spoken answer · ~85 s

A secure software supply chain on GCP protects every step from source to production. Source: protect branches with reviews and signed commits, scan for secrets, and limit who can change the pipeline. Build: use Cloud Build on hardened, ephemeral workers with a least-privilege service account, hermetic and reproducible builds where possible, and dependencies pinned and pulled from trusted sources, perhaps through Artifact Registry remote or virtual repositories with scanning. Generate build provenance, which Cloud Build produces at SLSA levels, and SBOMs. Artefacts: store images in Artifact Registry with vulnerability scanning on push, immutability, and restricted access via IAM; sign images with Cosign or Cloud KMS keys. Policy: use Binary Authorization with attestors, so only images built by your pipeline and passing scans can deploy, and continuous validation for running workloads. Platform: use private GKE clusters, Workload Identity, minimal base images such as distroless, Pod Security standards, and VPC Service Controls around artefact and build projects. Monitoring: Security Command Center, audit logs, and alerts on policy violations or unexpected pushes. Process: rotate and protect keys in KMS, track vulnerabilities to remediation with SLAs, and regularly review. The Software Delivery Shield offering bundles several of these. Verify provenance at deployment, not just at build time.

Likely follow-ups

- What does SLSA provenance prove?
- Where does Binary Authorization sit in the chain?

```
gcloud artifacts repositories create <repo> --repository-format=docker --location=<region>
```

```
gcloud artifacts docker images list <region>-docker.pkg.dev/<project>/<repo> --show-occurrences
```

```
gcloud builds submit --config=cloudbuild-sign.yaml # build, scan, attest in pipeline
```

Supply chain security layers: (1) Source — branch protection, code review, secret scanning. (2) Build — Cloud Build with hermetic builds, provenance attestation (SLSA). (3) Artifact — Artifact Registry with Container Analysis, vulnerability policy gates. (4) Deploy — Binary Authorization enforces only attested images. (5) Runtime — Falco/GKE security posture for anomaly detection.

SLSA framework: GCP Cloud Build supports SLSA Level 3 provenance — cryptographic proof that a specific source commit produced a specific image via a specific build process. This is the industry direction for supply chain security and a strong Principal-level talking point.

Official docs

- [GitHub: Artifact attestations \(build provenance\)](#)
- [SLSA specification](#)

Q40 How would you design a complete GCP Platform Engineering architecture for a growing product company?

CI/CD & IaC

Spoken answer · ~87 s

For a growing product company, I'd design a platform that gives teams self-service with guardrails. Foundation: an organisation with a folder hierarchy by environment, projects per team and environment, created by a Terraform project factory with budgets, labels, APIs, and baseline IAM; organisation policies and centralised logging, monitoring, and security projects; identity through Cloud Identity with groups and SSO. Networking: Shared VPC with a central network team, hub-and-spoke routing, Cloud NAT, private access, and hybrid connectivity if needed. Compute: GKE, standard or Autopilot, as the main runtime for services, with Cloud Run for simple stateless workloads, using fleets and Config Sync for consistent configuration. Delivery: Git-based workflows, Cloud Build or GitHub Actions with federation, Artifact Registry, Binary Authorization, and Cloud Deploy or Argo CD for promotion, with golden path templates so a new service is running in production in hours. Data: managed databases such as Cloud SQL, Spanner, and BigQuery with private connectivity. Security: Security Command Center, VPC Service Controls for sensitive data, Secret Manager, and least-privilege Workload Identity. Observability: Managed Prometheus, Cloud Logging, Trace, SLOs, and standard dashboards. Cost: FinOps with labels and budgets. Treat the platform as a product: a small team, documented paths, feedback loops, and metrics such as adoption and lead time. Evolve incrementally.

Likely follow-ups

- Why treat the platform as a product?
- What would you build first with a small team?

This is the end-to-end system design question. A Principal Platform Engineer owns this entire picture.

Foundation: Org hierarchy via Cloud Foundation Toolkit (Terraform)

Identity: Workload Identity everywhere, OS Login for VMs, no SA keys

Networking: Shared VPC + Cloud NAT + Private GKE + Cloud Interconnect

Compute: GKE Autopilot for apps, Cloud Run for stateless APIs, Cloud SQL + Spanner for data

CI/CD: Cloud Build + Artifact Registry + ArgoCD + Binary Authorization

Observability: Cloud Monitoring + Cloud Logging + Cloud Trace + SLO alerting

Security: VPC-SC + SCC + Org Policies + KMS + Secret Manager

Cost: Billing export → BigQuery → Looker dashboards + team labels + CUDs

The Platform team's output is a paved road: developers deploy applications without knowing GCP internals. Golden paths (IaC modules, CI/CD templates, observability standards) make the right thing the easy thing. Platform team measures success by developer deployment frequency and MTTR — not by individual infrastructure components.

Interview framing: A Principal Platform Engineer at a security-focused company also owns the security posture of the platform itself. Tie every architectural decision back to: least privilege, auditability, data protection, and supply chain integrity. That combination — platform reliability + security-first thinking — is exactly what the role requires.

Official docs

- [Google Cloud Architecture Framework](#)

Q41 Basic, predefined and custom roles in GCP IAM: which should you use and why avoid basic roles?

IAM & identity

Spoken answer · ~62 s

IAM roles are collections of permissions. Basic roles, Owner, Editor and Viewer, predate the current model and grant hundreds of permissions across nearly all services, so they violate least privilege. Editor, for example, can create resources and service accounts in the project. Predefined roles are maintained by Google for a specific product and task, such as Storage Object Viewer or Cloud SQL Client, and update automatically when new permissions are needed. Custom roles let me assemble an exact permission set, which is useful for tight boundaries, but I must maintain them as APIs change and they have limits on the number of permissions and scope. My approach is to grant predefined roles to groups, never to individual users, at the lowest practical level in the hierarchy, use IAM Recommender to remove unused permissions, add conditions for time or resource constraints, and keep basic roles out of production projects.

Likely follow-ups

- What does IAM Recommender do?
- How do you review role bindings at scale?

Basic roles (Owner, Editor, Viewer) are very broad and span almost every service. Predefined roles are curated per service and job. Custom roles let you pick exact permissions but need maintenance. Prefer predefined, and custom only when none fit.

```
gcloud iam roles describe roles/storage.objectViewer
```

```
gcloud projects add-iam-policy-binding my-proj --member=group:sre@corp.com --
role=roles/logging.viewer
```

```
gcloud iam roles create deployer --project=my-proj --file=role.yaml
```

Interview tip: Say Editor can modify almost everything, including creating service accounts, so it is a privilege escalation path.

Official docs

- [Google Cloud IAM overview](#)

Q42 What are IAM Conditions and IAM deny policies, and how do they differ from allow policies?

IAM & identity

Spoken answer · ~59 s

A normal allow policy binds principals to roles on a resource. An IAM Condition is an attribute-based restriction attached to a binding, so a role applies only when the expression is true, for example only until a date, only on resources with a given name prefix or tag, or only during working hours. That makes temporary or scoped access safer without creating many custom roles. Deny policies are separate: they list principals and permissions that are blocked, optionally with exceptions, and are evaluated before allow policies. Even if someone has a broad role granted lower in the hierarchy, a deny at the organization or folder wins. I use deny policies for guardrails such as preventing everyone except a break-glass group from creating service account keys or disabling audit logs, and I use conditions for just-in-time access combined with approval workflows and logging.

Likely follow-ups

- In what order are policies evaluated?
- How would you grant two hours of temporary admin access?

Allow policies grant access. IAM Conditions add attributes such as time, resource name or tag to a binding. Deny policies explicitly block permissions for principals, evaluated before allow, and apply at the organization, folder or project level.

```
gcloud projects add-iam-policy-binding my-proj --member=user:a@corp.com --
role=roles/compute.admin --condition='expression=request.time < timestamp("2026-12-
31T00:00:00Z"),title=temp'
```

```
gcloud iam policies create no-key-creation --attachment-
point=cloudresourcemanager.googleapis.com/organizations/123 --kind=denypolicies --policy-
file=deny.json
```

Interview tip: Deny always wins over allow, so deny policies are a strong guardrail against mistakes in lower-level grants.

 **Start here (official docs)**

- [Google Cloud documentation](#)

Q43 What are the risks of the default service accounts in GCP, and how do you remove them from your design?

IAM & identity

 **Spoken answer** · ~59 s

When the Compute Engine or App Engine API is enabled, Google creates default service accounts, and in many existing projects they hold the Editor role. If a VM is created without specifying a service account, it uses this identity, so any code running there, including a compromised web app, can read and change most project resources. That is a serious privilege escalation risk. The better design is one dedicated service account per workload or per tier, with only the specific predefined roles it needs, attached explicitly at creation, and with access scopes set broadly while IAM does the real restriction. I strip Editor from default accounts, enforce an organization policy so new projects do not auto-grant it, disable service account key creation, prefer workload identity or impersonation over keys, and use IAM Recommender and audit logs to confirm and tune the permissions.

Likely follow-ups

- Why do access scopes still matter on VMs?
- How do you migrate a running VM to a new service account?

The Compute Engine default service account is created with the Editor role in older projects and is attached to VMs by default. Any process on the VM inherits it. Use dedicated, least-privilege service accounts per workload and enforce with org policy.

```
gcloud iam service-accounts list --project my-proj
```

```
gcloud compute instances create vm1 --service-account=app-sa@my-proj.iam.gserviceaccount.com --scopes=cloud-platform
```

```
gcloud resource-manager org-policies enable-enforce  
iam.automaticIamGrantsForDefaultServiceAccounts --project my-proj
```

Interview tip: Name the org policy `iam.automaticIamGrantsForDefaultServiceAccounts`, which stops new projects from granting Editor to default accounts.

Official docs

- [Kubernetes: Service accounts](#)
- [Google Cloud: Best practices for service accounts](#)
- [Google Cloud Architecture Framework](#)

Q44 How do you use IAM Recommender and Policy Intelligence to reach least privilege?

IAM & identity

Spoken answer · ~59 s

Policy Intelligence is a set of tools. IAM Recommender looks at what permissions each principal has actually used over the past ninety days and recommends smaller roles or removal of unused bindings, with a count of excess permissions. Policy Troubleshooter takes a principal, resource and permission and shows exactly which bindings, conditions or deny policies produced the result, which saves a lot of time on 403 problems. Policy Analyzer answers audit questions such as who can access this bucket, or what can this service account touch, including inherited access. My process is to export recommendations regularly, review them with owners, apply changes through Terraform pull requests so they are tracked, and keep exceptions documented for rare but important access, like disaster recovery roles that are unused most of the year. I measure progress by the number of excess permissions over time.

Likely follow-ups

- What does Policy Troubleshooter show?
- Why can recommendations be unsafe to auto-apply?

IAM Recommender analyses 90 days of permission usage and suggests removing or replacing roles. Policy Troubleshooter explains why access was granted or denied. Policy Analyzer shows who can access what. Together they support audits and cleanup.

```
gcloud recommender recommendations list --project=my-proj --location=global --  
recommender=google.iam.policy.Recommender --format=json
```

```
gcloud policy-troubleshoot iam //storage.googleapis.com/projects/_/buckets/b --principal-  
email=a@corp.com --permission=storage.objects.get
```

Interview tip: Apply recommendations through code review, and test in a lower environment, because rarely used permissions such as disaster recovery may not appear in the usage window.

Official docs

- [AWS IAM: Least privilege \(grant least privilege\)](#)

 Spoken answer · ~59 s

In Autopilot mode, Google manages the nodes: provisioning, upgrades, scaling, security hardening and bin-packing, and you are billed for the CPU, memory and storage that your pods request rather than for idle node capacity. It enforces good defaults such as Workload Identity, shielded nodes and restrictions on privileged containers, which lowers operational effort and risk. Standard mode gives you full control over node pools, machine types, GPUs, local SSDs, custom kernel parameters, privileged workloads and DaemonSets, but you manage capacity and pay for nodes whether they are full or not. I choose Autopilot for most general application workloads and teams with limited platform capacity. I choose Standard when workloads need special hardware or node level access, third-party agents that need privileges, or when predictable large steady usage makes committed-use node pricing cheaper. Compute classes in Autopilot cover many special cases now.

Likely follow-ups

- How are Autopilot pods billed?
- What can't you do in Autopilot?

Autopilot manages nodes, scaling and hardening, and bills per pod resource request. Standard gives control over node pools, machine types, DaemonSets and kernel settings, and bills per node. Choose Autopilot unless you need that control.

```
gcloud container clusters create-auto prod --region europe-west1
```

```
gcloud container clusters create std --region europe-west1 --num-nodes 1 --enable-autoscaling --min-nodes 1 --max-nodes 5
```

```
kubectl describe pod <p> | grep -i 'autopilot\|cpu\|memory'
```

Interview tip: In Autopilot, requests are what you pay for, so accurate sizing directly cuts the bill.

 Official docs

- [GKE Autopilot](#)
- [GKE overview](#)

Q46 **What are GKE release channels and maintenance windows, and how do they keep clusters current safely?**

GKE

 Spoken answer · ~57 s

A release channel subscribes a cluster to a stream of Kubernetes versions that Google qualifies. Rapid gets new versions first, Regular is the default balance, and Stable receives versions that have run longer, with the most conservative cadence. Clusters on a channel are auto-upgraded, both control plane and nodes, so they never fall out of support. Maintenance windows define when automatic maintenance may start, and maintenance exclusions let me block upgrades during critical periods like a sales event, for a limited number of days, while still allowing security patches in some cases. A good strategy is to place development clusters on Rapid or Regular and production on Stable, so issues appear earlier in lower environments. I pair it with surge upgrade settings on node pools, PodDisruptionBudgets, and deprecation insights to catch removed APIs before they bite.

Likely follow-ups

- What are surge upgrades?
- How do you handle a deprecated API warning?

 GKE release channels

Rapid / Regular	Stable
Newer versions first	Slower, tested versions
More change	Fewer surprises

Rapid, Regular and Stable channels deliver Kubernetes versions at different cadences with different soak time. Cluster and node auto-upgrade follow the channel. Maintenance windows and exclusions control when upgrades happen.

```
gcloud container clusters update prod --region europe-west1 --release-channel regular
```

```
gcloud container clusters update prod --region europe-west1 --maintenance-window-start 2026-10-25T02:00:00Z --maintenance-window-end 2026-10-25T06:00:00Z --maintenance-window-recurrence 'FREQ=WEEKLY;BYDAY=SU'
```

```
gcloud container get-server-config --region europe-west1 --format='yaml(channels)'
```

Interview tip: Run dev on Rapid or Regular and production on Stable, so a version has been tested before it reaches production.

Official docs

- [GKE: Cluster upgrades](#)
- [GKE overview](#)

Q47 How does the GKE Gateway API work compared with classic Ingress?

GKE

Spoken answer · ~55 s

The classic Ingress resource is simple but limited, relying on annotations for advanced features and mixing infrastructure and application concerns in one object. The Gateway API splits this into roles. A GatewayClass identifies the implementation, for example the GKE global external managed load balancer. The platform team creates a Gateway that defines listeners, addresses and certificates. Application teams then create HTTPRoute objects in their own namespaces, which attach to the Gateway and define host and path matching, header-based routing, redirects, and weighted traffic splits for canary releases. GKE also offers multi-cluster Gateways that front services in several clusters, and policy resources for health checks, timeouts, session affinity, Cloud CDN and Cloud Armor. It gives better delegation and portability than annotations, so I use it for new designs and migrate Ingress gradually.

Likely follow-ups

- How do you do a canary with HTTPRoute?
- What is a multi-cluster Gateway?

Gateway API separates roles: platform teams define a Gateway tied to a load balancer, app teams attach HTTPRoutes. It supports header routing, traffic splitting, multi-cluster gateways and policies, with Google Cloud load balancers behind it.

```
kubectl get gatewayclass
```

```
kubectl apply -f gateway.yaml # gatewayClassName: gke-l7-global-external-managed
```

```
kubectl describe httproute web-route
```

Interview tip: Mention backend policies for health checks, timeouts and Cloud Armor attached through GCPBackendPolicy.

Official docs

- [Kubernetes: Ingress](#)
- [Kubernetes: Gateway API](#)
- [GKE overview](#)

Q48 **What are GKE node pools, node auto-provisioning and compute classes, and when do you use each?**

GKE

 Spoken answer · ~57 s

A node pool is a set of nodes with the same machine type, image and settings, and a cluster usually has several, for example general purpose, memory-heavy, GPU, and Spot. Each pool can use the cluster autoscaler within minimum and maximum bounds. Node auto-provisioning goes further: when pods are pending and no existing pool fits, GKE creates a new pool with a suitable machine shape, and removes it when it is empty, within the CPU, memory and GPU limits I configure. Custom compute classes let developers state in the pod spec what they prefer in priority order, like Spot on an N2D machine, then on-demand N2, and GKE will try each in turn, which balances cost and availability. I use fixed pools for stable baseline capacity and auto-provisioning or compute classes for variable and specialised workloads.

Likely follow-ups

- What limits would you set?
- How do compute classes express fallback?

Node pools are groups of identical nodes you define. Node auto-provisioning creates new pools automatically to fit pending pods within limits. Custom compute classes let workloads request ordered preferences, such as Spot first then on-demand, or specific machine families.

```
gcloud container node-pools create highmem --cluster prod --region europe-west1 --machine-type n2-highmem-8 --enable-autoscaling --min-nodes 0 --max-nodes 10
```

```
gcloud container clusters update prod --region europe-west1 --enable-autoprovisioning --max-cpu 500 --max-memory 2000
```

Interview tip: Always set resource limits for autoprovisioning so a bad deployment cannot create unlimited nodes.

 Official docs

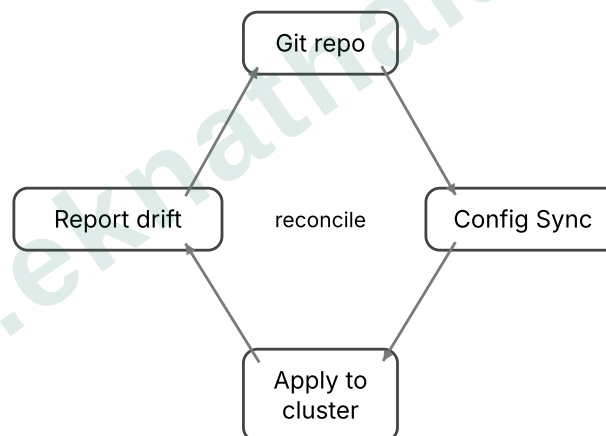
- [GKE overview](#)

Spoken answer · ~60 s

Config Sync is part of GKE Enterprise and fleet management. It watches a Git repository, an OCI image or a Helm chart source and applies the manifests to one or many clusters, with hierarchical namespaces support, and it reverts manual changes so the cluster matches Git. This gives audited, repeatable configuration and easy rollback via git revert. Config Connector is an add-on that installs custom resource definitions for Google Cloud resources. I can declare a storage bucket, Pub/Sub topic or Cloud SQL instance as YAML, and the controller creates and reconciles it using a Google service account, so application teams manage infrastructure beside their app manifests. Alternatives are Argo CD and Flux for syncing, and Terraform or Crossplane for infrastructure. I decide by where the team already works, but the principles of Git as truth, pull-based reconciliation, and drift correction stay the same.

Likely follow-ups

- Config Connector versus Terraform?
- How do you handle drift?

Config Sync

Config Sync continuously applies Kubernetes manifests from a Git repo to clusters and reverts drift. Config Connector exposes Google Cloud resources, such as buckets and Pub/Sub topics, as Kubernetes custom resources so they can be managed the same way.

```
gcloud beta container fleet config-management apply --membership=prod --config=apm.yaml
```

```
kubectl get storagebucket,pubsubtopic -n infra
```

```
nomos status
```

Interview tip: Alternatives to mention: Argo CD or Flux for sync, and Terraform for cloud resources. The principle is Git as the source of truth.

Official docs

- [OpenGitOps principles](#)
- [GKE overview](#)

Q50 How would you design a Shared VPC for a multi-team organisation on GCP?

Networking

Spoken answer · ~57 s

In a Shared VPC, one host project contains the VPC network, subnets, routes, firewall policies and Cloud NAT or interconnect attachments. Service projects owned by teams or environments are attached, and their resources such as VMs and GKE clusters take IP addresses from the shared subnets. A central network team manages connectivity and security, and delegates access by granting the Compute Network User role on specific subnets to the right groups. This keeps IP address planning consistent, avoids duplicate NATs and VPNs, and allows hierarchical firewall policies at the folder or organization level, while billing and quotas remain per service project. I typically create separate shared VPCs for production and non-production, plan secondary ranges for GKE pods and services, use private Google access, and monitor with VPC flow logs. Peering is used where networks must stay separate.

Likely follow-ups

- Shared VPC versus VPC peering?
- How do GKE secondary ranges fit in?

A host project owns the VPC and subnets, and service projects attach to it. Network admins control the network while project teams use subnets via the `compute.networkUser` role. This centralises firewalling and IP planning while keeping project-level billing and IAM.

```
gcloud compute shared-vpc enable host-proj
```

```
gcloud compute shared-vpc associated-projects add svc-proj --host-project host-proj
```

```
gcloud projects add-iam-policy-binding host-proj --member=group:team-a@corp.com --role=roles/compute.networkUser
```

Interview tip: Grant `networkUser` at the subnet level, not the whole host project, so teams can only use their own subnets.

Official docs

- [Google Cloud: Shared VPC](#)
- [Google Cloud VPC overview](#)

What are Private Google Access and Private Service Connect, and how do they keep traffic off the internet?

 Spoken answer · ~60 s

By default a VM without an external IP cannot reach Google APIs. Enabling Private Google Access on a subnet lets such VMs reach the public API addresses of Google services using internal routing. It is a simple switch, but traffic still uses Google's public API endpoints. Private Service Connect goes further. For Google APIs, I create an endpoint with a private IP inside my VPC that forwards to an API bundle, and I can point custom DNS names to it, so on-premises networks connected via VPN or Interconnect reach APIs without any internet path. For services, PSC lets a producer publish a service attachment, and consumers create an endpoint with an IP in their VPC. The two sides do not need to share or avoid overlapping address ranges, and the connection is one-directional, which is much simpler than VPC peering for third-party services.

Likely follow-ups

- PSC versus VPC peering?
- How do on-prem hosts use a PSC endpoint for APIs?

Private Google Access lets VMs with only internal IPs reach Google APIs. Private Service Connect creates a private endpoint with an internal IP in your VPC for Google APIs, or for services published by others, with no peering or shared address space.

```
gcloud compute networks subnets update app-subnet --region europe-west1 --enable-private-ip-google-access
```

```
gcloud compute addresses create psc-apis --global --purpose=PRIVATE_SERVICE_CONNECT --addresses=10.10.0.5 --network=prod
```

```
gcloud compute forwarding-rules create pscapis --global --network=prod --address=psc-apis --target-google-apis-bundle=all-apis
```

Interview tip: PSC is the answer when someone asks for private access to a SaaS or a producer's service across organisations without overlapping IP problems.

Official docs

- [Google Cloud: Private Service Connect](#)

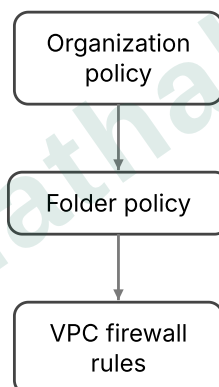
🗣 Spoken answer · ~57 s

GCP evaluates firewall policies in a fixed order: organization policy first, then folder policies, then VPC firewall rules and network-level policies, with implied rules at the end that allow all egress and deny all ingress. Inside hierarchical policies each rule can allow, deny, or use `goto_next` to pass the decision down to the next level. That means a central security team can enforce non-negotiable rules, such as blocking SSH and RDP from the internet or allowing health checker ranges, while delegating the rest to project owners. Rules apply to targets selected by network tags, service accounts, or secure tags. I prefer service accounts or secure tags because network tags can be changed by anyone who can edit the instance. Firewall Insights and logs show rule hit counts, shadowed rules and overly permissive rules to help with cleanup.

Likely follow-ups

- What is `goto_next`?
- How do you find unused rules?

🚩 Firewall hierarchy



Firewall policies can be attached to the organization, folders or networks and are evaluated top-down before VPC rules. Rules can allow, deny or delegate to the next level with `goto_next`. Use tags or service accounts to target workloads.

```
gcloud compute network-firewall-policies create corp-baseline --global
```

```
gcloud compute network-firewall-policies rules create 100 --firewall-policy corp-baseline -  
-global-firewall-policy --action deny --direction INGRESS --src-ip-ranges 0.0.0.0/0 --  
layer4-configs tcp:22,tcp:3389
```

```
gcloud compute firewall-rules list --filter='network:prod' --sort-by=priority
```

Interview tip: Prefer secure tags or service account targets over network tags, since network tags can be set by anyone with VM edit rights.

Official docs

- [Google Cloud: VPC firewall rules](#)

Q53 How do Cloud DNS private zones, forwarding and peering work for hybrid name resolution?

Networking

Spoken answer · ~55 s

Cloud DNS provides several mechanisms for private name resolution. A private managed zone is visible only from the VPC networks I attach to it, which is how internal service names are resolved. A forwarding zone sends queries for a specific domain to target name servers, typically on-premises resolvers over VPN or Interconnect, and the traffic comes from a special range that the on-premises firewall must allow. For the opposite direction, an inbound server policy exposes an IP address in my VPC that on-premises resolvers can forward queries to, so they can resolve Cloud DNS private zones. DNS peering lets one VPC use the zones of another without any network peering, which supports a central DNS project with spokes. I document zone ownership, avoid overlapping namespaces, and monitor query logs when diagnosing resolution problems.

Likely follow-ups

- Why does the on-prem firewall need to allow a special range?
- When would you use DNS peering?

Private zones answer names only for attached VPCs. Forwarding zones send queries for a domain to on-premises DNS. Inbound server policies let on-premises resolvers query Cloud DNS. DNS peering shares zones between VPCs without full network peering.

```
gcloud dns managed-zones create corp-private --dns-name=corp.internal. --visibility=private
--networks=prod --description=internal
```

```
gcloud dns managed-zones create onprem-fwd --dns-name=onprem.example.com. --
visibility=private --networks=prod --forwarding-targets=10.200.0.2
```

```
gcloud dns policies create inbound --networks=prod --enable-inbound-forwarding
```

Interview tip: Hub-and-spoke DNS: host private zones in a central project and use DNS peering from spokes.

Official docs

- [RFC 1035: DNS implementation and specification](#)
- [Cloud DNS](#)

 Spoken answer · ~58 s

All storage classes offer low-latency access, but they differ in price structure. Standard is for hot data. Nearline, Coldline and Archive have lower storage prices, higher retrieval fees and minimum storage durations of thirty, ninety and three hundred sixty-five days respectively, so deleting early still incurs the charge. Lifecycle rules automate transitions and deletions, for example move to Nearline after thirty days, Coldline after ninety, and delete after a year, or remove old noncurrent versions. Autoclass is for unpredictable access patterns: it monitors object access and moves objects between classes automatically, with no retrieval fees for transitions, so I do not have to guess. I choose lifecycle rules when I know the data pattern, like logs and backups, and Autoclass for mixed data lakes. I also use regional versus multi-region buckets according to latency, availability and egress cost needs.

Likely follow-ups

- What is the early deletion fee?
- When would Autoclass cost more?

Standard, Nearline, Coldline and Archive trade lower storage price for higher retrieval cost and minimum storage duration, all with the same millisecond access. Lifecycle rules transition or delete by age. Autoclass moves objects automatically based on access.

```
gsutil lifecycle set lifecycle.json gs://my-bucket
```

```
gcloud storage buckets update gs://my-bucket --enable-autoclass
```

```
gcloud storage objects update gs://my-bucket/old.log --storage-class=COLDLINE
```

Interview tip: Remember early deletion fees: Nearline 30 days, Coldline 90, Archive 365.

 Official docs

- [Kubernetes: StorageClasses](#)
- [Amazon S3: Lifecycle management](#)
- [Cloud Storage](#)

Q55 **Signed URLs and signed policy documents in Cloud Storage: how do you give temporary access safely?**

Storage & data

 Spoken answer · ~60 s

A signed URL embeds a signature that proves a principal authorised a particular request, for a specific method, object and expiry time. Anyone with the URL can perform that action until it expires, so I keep the lifetime short and limit it to the minimum scope. The signature is created using the credentials of a service account, and that account must itself have permission on the object. To avoid creating and storing a long-lived private key, I use service account impersonation with the IAM Credentials API `signBlob`, granting the application the Token Creator role on a dedicated signer account. For browser uploads, a signed policy document or resumable upload URL can restrict content type and size. Alternatives for broader audiences are Cloud CDN signed URLs or signed cookies. I log access, keep buckets private with uniform bucket-level access, and rotate approaches if a URL leaks.

Likely follow-ups

- Why avoid service account key files?
- Signed URL versus signed cookie?

A signed URL grants time-limited access to one object using the signer's credentials. Prefer V4 signing with a service account that has minimal access, via impersonation instead of key files. Policy documents control browser uploads.

```
gcloud storage sign-url gs://bucket/report.pdf --duration=10m --impersonate-service-account=signer@proj.iam.gserviceaccount.com
```

```
gcloud iam service-accounts add-iam-policy-binding signer@proj.iam.gserviceaccount.com --member=user:a@corp.com --role=roles/iam.serviceAccountTokenCreator
```

Interview tip: Avoid downloading service account keys just to sign. IAM Credentials `signBlob` works with impersonation.

 Official docs

- [Cloud Storage](#)

 Spoken answer · ~59 s

For block storage attached to a VM or pod, Persistent Disk is the standard, with balanced as the general default, SSD for higher performance, and standard for cheap throughput. Performance scales with size, so a small disk may be slow. Hyperdisk is the newer family where I provision capacity, IOPS and throughput separately, which suits databases and other workloads that need predictable high performance without oversizing the disk, and it is required by some newer machine series. Regional disks replicate synchronously to a second zone to support failover of stateful workloads. When many nodes need the same files, such as shared content, build caches or home directories, Filestore provides managed NFS in several tiers, including enterprise with regional availability, and the Filestore CSI driver lets GKE pods use it as ReadWriteMany volumes. For object data and analytics, Cloud Storage is more suitable.

Likely follow-ups

- Regional versus zonal persistent disk?
- Why does disk size affect performance?

Persistent Disk (balanced, SSD, standard) is the default block storage. Hyperdisk lets you provision IOPS and throughput independently for demanding databases. Filestore provides managed NFS for shared access, including by GKE pods.

```
gcloud compute disks create data1 --size=500GB --type=pd-balanced --zone=europe-west1-b
```

```
gcloud compute disks create fast1 --size=1TB --type=hyperdisk-balanced --provisioned-iops=20000 --provisioned-throughput=800 --zone=europe-west1-b
```

```
gcloud filestore instances create shared --zone=europe-west1-b --tier=ENTERPRISE --file-share=name=vol1,capacity=1TB --network=name=prod
```

Interview tip: Regional persistent disks replicate synchronously across two zones, which helps stateful workloads fail over without data loss.

 Start here (official docs)

- [Google Cloud documentation](#)

Q57 **How do managed instance group rolling updates, autohealing and health checks work together?**

Compute

 Spoken answer · ~58 s

A managed instance group maintains identical VMs from an instance template. For updates, the rolling update policy replaces instances in batches, with `maxSurge` letting extra temporary instances be created and `maxUnavailable` limiting how many can be down, so setting `maxUnavailable` to zero gives zero downtime. A proactive update replaces VMs, while opportunistic updates apply only when instances are recreated for other reasons. Autohealing uses a health check to detect unresponsive applications and recreates the instance, but it only starts checking after the initial delay, which must be longer than the application's boot time. Load balancer health checks are separate: they decide whether to send traffic and do not recreate VMs, so I commonly configure both. I use regional MIGs spread across zones, and canary updates by running two template versions with a target size split for the new one.

Likely follow-ups

- What is the initial delay for?
- How do you canary a template?

A MIG replaces instances gradually to a new template using `maxSurge` and `maxUnavailable`. Autohealing recreates VMs that fail an application-level health check after an initial delay. Load balancer health checks are separate and only control traffic.

```
gcloud compute instance-groups managed rolling-action start-update web-mig --  
version=template=web-v2 --max-surge=3 --max-unavailable=0 --region=europe-west1
```

```
gcloud compute instance-groups managed update web-mig --health-check=app-hc --initial-  
delay=300 --region=europe-west1
```

```
gcloud compute instance-groups managed describe web-mig --region=europe-west1
```

Interview tip: Set the autohealing initial delay longer than your slowest boot, or healthy-but-booting VMs will be killed in a loop.

 Official docs

- [Kubernetes: Deployments](#)
- [Compute Engine: Managed instance groups](#)

Spoken answer · ~57 s

Spot VMs use spare Compute Engine capacity at a large discount, but Google may preempt them whenever it needs the resources, giving about thirty seconds of warning through the shutdown signal and a metadata flag. Unlike the older preemptible type, they have no twenty-four hour limit. Because capacity can disappear, they suit batch processing, CI workers, rendering, data pipelines and stateless web tiers behind a load balancer, provided the application checkpoints, retries or is replicated. I mix them with regular instances so the baseline is always there, spread them across zones and machine types in a managed instance group or a GKE Spot node pool, and handle the shutdown script to drain gracefully. I would not use them for databases or single points of failure. Costs and availability vary by region, so I measure the preemption rate.

Likely follow-ups

- Spot versus committed use discounts?
- How does GKE handle Spot node termination?

Spot VM

Spot VMs are heavily discounted instances that Google can stop at any time with a 30-second notice and have no maximum runtime. Use them for stateless, batch or fault-tolerant work with checkpointing, and mix with on-demand capacity for baseline.

```
gcloud compute instances create worker1 --provisioning-model=SPOT --instance-termination-action=STOP --machine-type=e2-standard-4
```

```
curl -s -H 'Metadata-Flavor: Google' http://metadata.google.internal/computeMetadata/v1/instance/preempted
```

```
gcloud container node-pools create spot-pool --cluster prod --spot --enable-autoscaling --min-nodes 0 --max-nodes 20
```

Interview tip: Poll the preempted metadata flag or handle the ACPI shutdown signal so work can checkpoint within the 30 seconds.

Start here (official docs)

- [Google Cloud documentation](#)

Q59 **What are OS Login and IAP TCP forwarding, and why use them instead of SSH keys and public IPs?**

Compute

 Spoken answer · ~57 s

Traditionally, SSH access is managed by adding public keys to instance or project metadata, which scatters keys, makes revocation hard and gives little audit trail. OS Login maps the Linux account to a Google identity. Access is granted with IAM roles, such as OS Login for normal users and OS Admin Login for sudo, keys are managed centrally and can be tied to two-step verification, and offboarding a person from the group removes their access everywhere. Identity-Aware Proxy TCP forwarding addresses the network side. Instead of giving VMs public IP addresses and opening port 22 to the internet, users connect through the IAP tunnel after authentication and authorisation, so VMs can be private. A firewall rule must allow IAP's address range. Together they give private VMs, identity-based access control, and audit logs of who connected when.

Likely follow-ups

- Which IAM roles does OS Login use?
- What firewall rule does IAP need?

OS Login ties SSH access to IAM identities, so access and sudo are granted by roles and logged, with no key distribution. IAP TCP forwarding tunnels SSH or RDP through Google's identity-aware proxy to VMs that have no external IP.

```
gcloud compute project-info add-metadata --metadata enable-oslogin=TRUE
```

```
gcloud projects add-iam-policy-binding my-proj --member=group:ops@corp.com --  
role=roles/compute.osAdminLogin
```

```
gcloud compute ssh vm1 --tunnel-through-iap --zone=europe-west1-b
```

Interview tip: IAP needs a firewall rule allowing 35.235.240.0/20 on port 22, a detail interviewers like to hear.

 Official docs

- [OpenSSH manual pages](#)

Q60 **How do you define SLIs and SLOs in Cloud Monitoring and alert on error budget burn?**

Observability

 Spoken answer · ~60 s

In Cloud Monitoring, I model a service, for example by selecting a GKE workload, Cloud Run service or custom service, and then create an SLI. For availability, a request-based SLI is the ratio of good requests to total requests from the load balancer or application metrics. For latency, it is the proportion of requests faster than a threshold. I set an SLO target such as ninety-nine point nine percent over a rolling twenty-eight day window, which gives an error budget of the allowed bad requests. Instead of alerting on raw error rates, I create burn-rate alerts: a fast one, such as a fourteen times burn over an hour, pages immediately, and a slower one, such as a six times burn over six hours, creates a ticket. The dashboard shows remaining error budget, which informs release decisions, and I review SLOs with the product owners regularly.

Likely follow-ups

- What is a burn rate?
- Request-based versus window-based SLIs?

Create a service, define a request-based or window-based SLI from a metric such as load balancer request count and latency, set an SLO target over a rolling or calendar window, and add burn-rate alerting policies on the error budget.

```
gcloud monitoring slos create --service=checkout --display-name='99.9% availability' --goal=0.999 --rolling-period=28d --request-based-sli=...
```

```
gcloud alpha monitoring policies create --policy-from-file=burn-rate.yaml
```

Interview tip: Use two burn-rate windows, fast and slow, to catch sudden outages and slow leaks without constant noise.

 Official docs

- [Cloud Monitoring](#)
- [Google SRE Book: Service Level Objectives](#)
- [Google SRE Book: Embracing Risk \(error budgets\)](#)

Q61 How do Cloud Logging sinks, log-based metrics and exclusion filters work together?

Observability

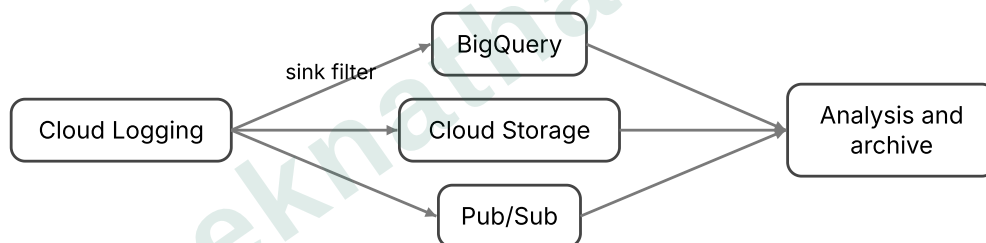
Spoken answer · ~61 s

Every log entry arrives at the Logging API and is routed through sinks. By default the Required bucket keeps audit logs and the Default bucket keeps the rest, each with its own retention. I add sinks to export filtered logs to BigQuery for analysis, Cloud Storage for cheap archive, Pub/Sub for streaming to a SIEM, or to a central logging project through an aggregated sink at the folder or organization level. The sink has a writer identity that I must grant permission on the destination. Log-based metrics count matching entries or extract values such as latency, producing counters and distributions I can chart and alert on, which is handy for application events not exposed as metrics. Exclusion filters on the Default sink drop high-volume low-value logs, for example health check requests, before they are stored. I also set retention by bucket to control cost and meet compliance.

Likely follow-ups

- Why does the sink need a writer identity?
- How do you cut logging cost?

Log sinks



Sinks route matching logs to BigQuery, Cloud Storage, Pub/Sub or another bucket. Log-based metrics turn log patterns into metrics for dashboards and alerts. Exclusion filters drop logs before storage to cut cost. The `_Required` bucket cannot be disabled.

```
gcloud logging sinks create audit-to-bq bigquery.googleapis.com/projects/sec-  
proj/datasets/audit --log-filter='logName:"cloudaudit.googleapis.com"' --include-children -  
-organization=123
```

```
gcloud logging metrics create http_5xx --log-filter='resource.type="k8s_container" AND  
jsonPayload.status>=500'
```

```
gcloud logging buckets update _Default --location=global --retention-days=30
```

Interview tip: Remember to grant the sink's writer identity access on the destination, a step people often forget.

Official docs

- [Cloud Logging](#)

Q62 What is Google Cloud Managed Service for Prometheus, and when is it better than self-hosting Prometheus?

Observability

Spoken answer · ~60 s

Managed Service for Prometheus provides collection and storage for Prometheus metrics without operating the Prometheus servers' scaling and long-term storage. On GKE, managed collection runs lightweight collectors as a DaemonSet, and I define scrape targets with PodMonitoring or ClusterPodMonitoring custom resources. Data is stored in Google's planet-scale Monarch backend, retained for twenty-four months, queryable across clusters and projects with PromQL from Grafana or the Cloud Monitoring UI, and can combine with Google Cloud system metrics. It is better than self-hosting when I run many clusters, need global queries, long retention, or high availability without sharding and Thanos engineering. The trade-offs are cost per ingested sample, less control over internals, and the need to manage cardinality carefully. For small, single-cluster setups, a self-managed Prometheus can still be cheaper and simpler. I always test that dashboards and alerts written in PromQL behave the same as before migrating.

Likely follow-ups

- How would you reduce sample volume?
- How does it handle multi-cluster queries?

It is a managed, Prometheus-compatible backend (Monarch) that stores metrics globally, queried with PromQL. You keep exporters and PodMonitoring resources, but avoid running storage, sharding and retention yourself. Charges are per sample ingested.

```
kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/prometheus-engine/main/examples/pod-monitoring.yaml
```

```
kubectl get podmonitoring -A
```

```
gcloud container clusters update prod --region europe-west1 --enable-managed-prometheus
```

Interview tip: Cost is driven by sample count, so control cardinality and scrape interval.

Official docs

- [Google Cloud Managed Service for Prometheus](#)
- [Prometheus: Overview](#)

Q63 **How do you use Secret Manager safely from applications on GCP, and handle rotation?**

Security

 Spoken answer · ~61 s

Secret Manager stores sensitive values as versions of a secret, encrypted at rest, with fine-grained IAM and audit logging of every access. I grant the Secret Accessor role on each individual secret to the workload's own service account, rather than on the whole project. Applications read the secret at startup or on demand using client libraries, Cloud Run and Cloud Functions can inject secrets as environment variables or mounted volumes, and on GKE the Secret Manager add-on or CSI driver mounts them as files, using Workload Identity so no keys are stored. For rotation, a secret can have a rotation schedule that publishes a Pub/Sub message, which triggers a function that generates a new credential in the target system and adds a new version. Applications either reload periodically or restart. I disable or destroy old versions after a grace period, and alert on unusual access patterns.

Likely follow-ups

- Pinned version versus latest?
- How does Cloud Run mount a secret?

Store secrets in Secret Manager with versions and IAM at the secret level. Applications use their workload identity to access the latest or a pinned version. Rotation publishes Pub/Sub notifications that trigger a function to create a new version.

```
gcloud secrets create db-password --replication-policy=automatic
```

```
printf 's3cr3t' | gcloud secrets versions add db-password --data-file=-
```

```
gcloud secrets add-iam-policy-binding db-password --  
member=serviceAccount:app@proj.iam.gserviceaccount.com --  
role=roles/secretmanager.secretAccessor
```

Interview tip: Pin to a version number in production and move it deliberately, rather than reading latest, to avoid surprise breakage during rotation.

 Official docs

- [Secret Manager](#)

Q64 **What are customer-managed encryption keys (CMEK) in Cloud KMS, and when are they worth the overhead?**

Security

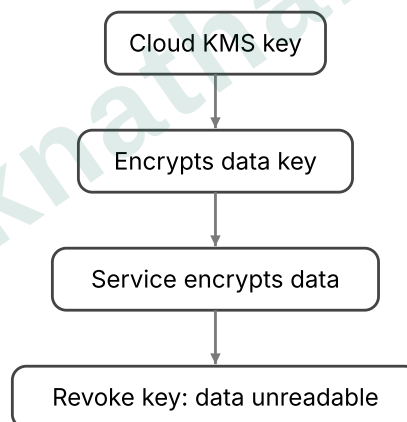
 Spoken answer · ~62 s

All data in Google Cloud is encrypted at rest by default with Google-managed keys. With CMEK, supported services such as Cloud Storage, BigQuery, Compute Engine disks and Cloud SQL encrypt data with a key I own in Cloud KMS, which lives in a keyring in a chosen location. The service agent receives the Encrypter Decrypter role on that key. This lets me set rotation periods, define who can administer versus use the key, audit every use in Cloud Audit Logs, and revoke access instantly by disabling the key. It is worth it for regulatory or contractual requirements, data sovereignty and separation of duties. The costs are operational: key location must match the data, a disabled key makes data unavailable, and key management adds permissions and monitoring. For stricter needs, there are Cloud HSM and External Key Manager options. I automate keys with Terraform and organization policies requiring CMEK.

Likely follow-ups

- CMEK versus CSEK?
- What is Cloud EKM?

 CMEK



By default Google encrypts data with Google-managed keys. CMEK lets you control a Cloud KMS key used to encrypt supported services, giving you rotation control, the ability to disable or destroy the key to cut access, and separate duties. It adds operational responsibility.

```
gcloud kms keyrings create prod --location=europe-west1
```

```
gcloud kms keys create bucket-key --keyring=prod --location=europe-west1 --  
purpose=encryption --rotation-period=90d --next-rotation-time=2026-12-01T00:00:00Z
```

```
gcloud storage buckets update gs://secure-bucket --default-encryption-  
key=projects/p/locations/europe-west1/keyRings/prod/cryptoKeys/bucket-key
```

Interview tip: If the key is disabled or destroyed, the data becomes unreadable. Protect key admin access and enable key destruction scheduling delays.

 Official docs

- [AWS KMS](#)
- [Cloud KMS](#)

Q65 **Which Cloud Audit Logs types exist, and which must you enable explicitly?**

Security

 Spoken answer · ~62 s

Cloud Audit Logs has four categories. Admin Activity logs record API calls that modify configuration or metadata, such as creating a VM or changing IAM, they are always on, cannot be disabled and are free. System Event logs record Google-initiated changes, like live migration. Data Access logs record reads of configuration and user data, and data reads or writes, and they are disabled by default for most services because of volume and cost, so I enable them through audit configuration for sensitive services like Cloud Storage, Secret Manager and KMS, choosing admin read, data read and data write as needed. Policy Denied logs record when a request is blocked by a security policy such as VPC Service Controls. I route audit logs through an aggregated sink to a central, locked-down project, apply long retention, and build alerts on sensitive actions such as IAM changes and service account key creation.

Likely follow-ups

- Why are Data Access logs off by default?
- How do you export audit logs organisation-wide?

Admin Activity logs are always on and free. System Event logs are always on. Data Access logs record reads and data writes but are off by default for most services (except BigQuery) and must be enabled. Policy Denied logs record security policy denials.

```
gcloud projects get-iam-policy my-proj --format='yaml(auditConfigs)'
```

```
gcloud logging read 'logName:"cloudaudit.googleapis.com%2Factivity" AND  
protoPayload.methodName="SetIamPolicy"' --limit 10
```

```
gcloud projects set-iam-policy my-proj policy-with-auditconfigs.yaml
```

Interview tip: Data Access logs can be voluminous and costly, so enable them for sensitive services and exempt noisy service accounts.

 Start here (official docs)

- [Google Cloud documentation](#)

Q66 **What does Cloud Deploy add on top of Cloud Build for delivery to GKE and Cloud Run?**

CI/CD & IaC

 Spoken answer · ~60 s

Cloud Build is a general CI service that runs build steps, runs tests, builds container images and pushes them to Artifact Registry. Cloud Deploy is a managed continuous delivery service that takes those artifacts and moves them through a defined progression of targets, such as dev, staging and production. A delivery pipeline lists the targets in order, a release captures a specific set of images and manifests rendered by Skaffold, and rollouts apply them to each target. It supports approvals before production, canary and blue-green strategies for GKE and Cloud Run, automated verification steps, parallel deployments to multiple clusters, easy rollback to an earlier release, and a clear audit trail of who promoted what. This enforces the principle of building once and promoting the same immutable artifact. I use IAM so developers can create releases but only the release team can approve production.

Likely follow-ups

- Why build once and promote?
- How does Cloud Deploy do a canary?

Cloud Build builds and tests artifacts. Cloud Deploy manages delivery pipelines across targets with promotion, approvals, canary and blue-green strategies through Skaffold, rollback, and a release audit trail.

```
gcloud deploy apply --file=clouddeploy.yaml --region=europe-west1
```

```
gcloud deploy releases create rel-042 --delivery-pipeline=web --region=europe-west1 --  
images=web=europe-west1-docker.pkg.dev/proj/repo/web:42
```

```
gcloud deploy rollouts promote rel-042 --delivery-pipeline=web --region=europe-west1
```

Interview tip: The key sentence: Build creates the artifact once, Deploy promotes the same artifact through environments.

 Official docs

- [Cloud Build](#)
- [Cloud Deploy](#)
- [Cloud Run](#)

Q67 **How do Cloud Build private pools and triggers work, and why use them for builds that touch private resources?**

CI/CD & IaC

 Spoken answer · ~60 s

A Cloud Build trigger starts a build when something happens, such as a push to a branch, a pull request, a tag or a Pub/Sub message, and runs the steps defined in a build configuration file. By default, builds run on Google-managed workers outside my network, so they cannot reach a private GKE API endpoint, an internal artifact repository or a database. A private pool solves this by running workers in a Google-managed VPC that is peered to my VPC, so builds can reach private resources, and it supports custom machine sizes, regions and VPC Service Controls. For security I use a dedicated service account per trigger with only the roles it needs, instead of the broad default account, restrict who can edit triggers, use approval gates for sensitive triggers, and keep secrets in Secret Manager referenced by the build, not in the config.

Likely follow-ups

- Why avoid the default Cloud Build service account?
- What are the limits of private pool peering?

Default workers run in a Google-managed network with no access to your VPC. Private pools run in a peered network, giving builds access to private registries, databases or GKE control planes, with custom machine sizes and VPC Service Controls support.

```
gcloud builds worker-pools create private-pool --region=europe-west1 --peered-  
network=projects/host/global/networks/prod --worker-machine-type=e2-standard-4
```

```
gcloud builds triggers create github --name=main-build --repo-name=app --repo-owner=corp --  
branch-pattern='^main$' --build-config=cloudbuild.yaml --service-  
account=projects/p/serviceAccounts/builder@p.iam.gserviceaccount.com
```

Interview tip: Use a dedicated user-specified service account per trigger instead of the legacy default Cloud Build account.

 Official docs

- [Cloud Build](#)

Q68 **How do committed use discounts work on GCP, and how do they differ from sustained use discounts?**

Cost

 Spoken answer · ~60 s

Committed use discounts reduce price in return for committing to a level of usage for one or three years. Resource-based commitments cover a quantity of vCPU and memory in a region, and apply automatically to matching usage, while spend-based or flexible commitments commit to a dollar amount per hour on eligible services and give more flexibility across machine families, though with a somewhat smaller discount. Sustained use discounts were an automatic discount for running certain instances for much of the month, requiring no commitment, but they apply only to some older machine series, and newer series rely on committed use instead. My approach is to analyse the stable baseline from billing export in BigQuery, commit conservatively, use Spot for interruptible capacity, shut down non-production environments out of hours, and track commitment utilisation and coverage so that money is not wasted on unused commitments.

Likely follow-ups

- Resource-based versus spend-based?
- How do you size a commitment?

Committed use discounts trade a one or three year commitment for a lower price, either resource-based (vCPU, memory) or spend-based (flexible). Sustained use discounts are automatic for some machine types and run-time, and are limited to older series. Combine with Spot for variable work.

```
gcloud compute commitments create web-cud --region=europe-west1 --plan=12-month --resources=vcpu=64,memory=256GB
```

```
gcloud billing accounts list
```

```
bq query 'SELECT service.description, SUM(cost) FROM `billing.gcp_billing_export_v1_XXXX` GROUP BY 1 ORDER BY 2 DESC'
```

Interview tip: Buy commitments for the steady baseline only, and review utilisation of the commitment monthly.

 Official docs

- [Committed use discounts](#)

Q69 **How do you set up billing export to BigQuery and labels to attribute GCP costs to teams?**

Cost

 Spoken answer · ~56 s

I enable Cloud Billing export to BigQuery, the standard export for invoice-level cost and the detailed export for resource-level data, which becomes the source for all analysis. To attribute cost, the best boundary is the project, organised under folders by team or environment, which needs no tagging. Within shared projects I use labels such as team, service and environment, and enforce them with organization policy constraints, Terraform modules and checks in CI, since labels are not applied retroactively. For shared resources like GKE clusters, I enable GKE cost allocation or use namespace labels to split cost, and for shared networking I allocate proportionally. Looker Studio dashboards on the export give showback by team, and budgets with alerts to Pub/Sub or email notify owners. I also review the recommender for idle resources and rightsizing regularly.

Likely follow-ups

- Why aren't labels retroactive?
- How do you attribute shared GKE costs?

Enable standard and detailed usage cost export to a BigQuery dataset, standardise resource labels such as team and env enforced by policy, use folders and projects as the main boundary, and build dashboards with Looker Studio on the export data.

```
bq query --use_legacy_sql=false 'SELECT labels.value AS team, SUM(cost) FROM
`billing.gcp_billing_export_v1_X`, UNNEST(labels) AS labels WHERE labels.key="team" GROUP
BY 1'
```

```
gcloud billing budgets create --billing-account=ABC --display-name=team-a --budget-
amount=5000USD --threshold-rule=percent=0.8
```

Interview tip: Labels apply only from when they are set. They are not retroactive, so enforce them at resource creation.

 Official docs

- [BigQuery](#)
- [Cloud Billing budgets and alerts](#)

Q70 BigQuery on-demand versus capacity (slot) pricing: how do you choose and control query cost?

Cost

 Spoken answer · ~58 s

With on-demand pricing, I pay for the bytes each query processes, so cost depends on how well queries limit data scanned, and I get access to a shared pool of slots. With capacity pricing, I purchase slots in an edition, with autoscaling and optional commitments, and pay for slot-hours regardless of how much data is scanned, which suits steady heavy analytics and gives predictable cost and workload isolation through reservations and assignments. To control costs on either model, I partition tables by date and cluster by frequently filtered columns, select only needed columns rather than select star, require partition filters, use dry runs to estimate bytes, set maximum bytes billed on queries, apply project and user-level quotas, use materialised views or summary tables for repeated dashboards, and monitor the INFORMATION_SCHEMA jobs views to find expensive queries and users.

Likely follow-ups

- What does clustering do?
- How does a dry run help?

BigQuery pricing

On-demand	Capacity slots
Pay per TB scanned	Pay for reserved compute
Bursty, light use	Steady heavy use

On-demand charges per TB scanned and shares a pool of slots. Capacity pricing buys dedicated slots with editions and autoscaling, giving predictable cost for heavy workloads. Control cost with partitioning, clustering, selecting only needed columns, and quotas.

```
bq query --dry_run --use_legacy_sql=false 'SELECT * FROM `proj.ds.events` WHERE dt = "2026-10-20" '
```

```
bq query --maximum_bytes_billed=100000000000 --use_legacy_sql=false '...'
```

```
bq mk --reservation --location=EU --edition=ENTERPRISE --slot_capacity=100 prod-res
```

Interview tip: Always partition large tables on a date column and require a partition filter to prevent accidental full scans.

Official docs

- [BigQuery](#)

Q71 **GKE pods are Pending with 'Insufficient' errors or IP allocation failures. How do you diagnose it?**

GKE

 Spoken answer · ~63 s

The pod events say whether the scheduler failed because no node has enough CPU or memory, or because of taints and affinity, or whether the node cannot assign an IP. For resource shortage, I check requested versus allocatable resources, whether the cluster autoscaler is enabled, and whether it is blocked by quota or maximum node limits. For IP problems, in a VPC-native cluster the pod secondary range is carved into per-node blocks, by default a /24 per node for one hundred ten pods, so a small pod range limits the number of nodes, and the primary subnet range limits the node count as well. Fixes include adding discontinuous multi-pod CIDR ranges, reducing max pods per node so smaller blocks are used, enlarging subnets in a new node pool, or creating a new cluster with a larger plan. I prevent it by calculating the IP budget during design and monitoring utilisation.

Likely follow-ups

- What is a discontinuous multi-pod CIDR?
- How does max pods per node affect IP use?

Distinguish resource shortage from IP exhaustion. Check pod events, node allocatable versus requests, autoscaler status, and the pod secondary range utilisation. VPC-native clusters allocate a /24 slice of the pod range per node, which can run out.

```
kubectl describe pod <p> | tail -15
```

```
kubectl get nodes -o custom-columns=NAME:.metadata.name,PODCIDR:.spec.podCIDR
```

```
gcloud container clusters describe prod --region europe-west1 --  
format='yaml(ipAllocationPolicy,defaultMaxPodsConstraint)'
```

Interview tip: Lower max pods per node or add a discontinuous pod range to extend a cluster that ran out of pod IPs.

Official docs

- [Kubernetes: Debug Pods](#)
- [GKE overview](#)

Q72 **A GKE Ingress or Gateway returns 502 errors and the backend shows as unhealthy. What do you check?**

GKE

 Spoken answer · ~64 s

I check the backend health first, which shows which endpoints are unhealthy. The Google Cloud load balancer probes from specific ranges, one thirty dot two eleven dot zero dot zero slash twenty-two and thirty-five dot one ninety-one dot zero dot zero slash sixteen, so a firewall rule must allow them to reach the pods or nodes on the serving port. Next the health check configuration: by default GKE infers it from the pod's readiness probe, but if the path returns a redirect or non-200 status, or the probe is on a different port, the check fails. A BackendConfig or HealthCheckPolicy lets me set the path, port and thresholds explicitly. I confirm the app listens on the right port and interface, that container-native load balancing with NEGs is used so traffic goes directly to pods, and that pods are ready. For intermittent 502s, I examine connection draining, keep-alive timeouts, and the backend timeout setting.

Likely follow-ups

- Why are NEGs better than instance groups?
- Which ranges do Google health checks use?

The Google load balancer health check must succeed against the pod. Check that firewall rules allow Google health check ranges, the readiness probe and health check path match, container-native NEGs are in use, and the app listens on the target port.

```
kubectl describe ingress web
```

```
gcloud compute backend-services get-health k8s1-abc-web-80 --global
```

```
gcloud compute firewall-rules list --filter='sourceRanges:(130.211.0.0/22 35.191.0.0/16)'
```

Interview tip: The load balancer health check is derived from the readiness probe by default, but only if the path and port are compatible.

 Official docs

- [Kubernetes: Ingress](#)
- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [GKE overview](#)

Q73 **A GKE node auto-upgrade broke workloads. How do you reduce the blast radius next time?**

GKE

 Spoken answer · ~61 s

First I identify exactly what changed by reading the upgrade operation, release notes for the new version, and cluster events. Common causes are a removed or changed API that a controller still used, a different default such as a container runtime or kernel behaviour, a node image update that changed a driver, or workloads with single replicas and no disruption budget being evicted at the same time. To recover, I fix the workload or roll traffic to a new node pool built from the previous version, since control planes cannot be downgraded. To reduce the blast radius in future, I run staging clusters on a faster channel so I see the version first, use maintenance windows and exclusions, set surge upgrade with zero max unavailable, ensure PodDisruptionBudgets and at least two replicas, check deprecation insights before each upgrade, and use blue-green node pool upgrades for critical clusters.

Likely follow-ups

- Can you downgrade a GKE control plane?
- What is blue-green node pool upgrade?

Identify what changed: a Kubernetes version removing an API or changing behaviour, node image changes, or workloads that cannot tolerate disruption. Use release channels with soak, maintenance windows and exclusions, surge settings, PDBs, and test in staging.

```
gcloud container operations list --filter='operationType:UPGRADE_NODES' --limit 5
```

```
kubectl get events -A --sort-by=.lastTimestamp | tail -30
```

```
gcloud container node-pools update pool1 --cluster prod --region europe-west1 --max-surge-upgrade 2 --max-unavailable-upgrade 0
```

Interview tip: For a risky window, use a maintenance exclusion, and for a rollback create a new pool on the old version if the old minor is still supported.

 Official docs

- [AWS: Disaster recovery & availability concepts \(Fault isolation boundaries\)](#)
- [GKE: Cluster upgrades](#)
- [AWS Builders' Library: Workload isolation using shuffle sharding](#)

Q74 **A pod using Workload Identity cannot access Google APIs and gets 403 or metadata server errors. How do you fix it?**

GKE

 Spoken answer · ~59 s

I run a curl against the metadata server from inside the pod to see which identity it receives. If it shows the node's default service account or fails, Workload Identity is not active for that pod. I check that the cluster has a workload pool configured and the node pool uses the GKE metadata server, since older pools may not. Then I confirm the Kubernetes service account has the iam.gke.io slash gcp-service-account annotation pointing to the right Google service account. On the Google side, that account must grant the Workload Identity User role to the member string with the project's workload pool, namespace and service account name, exactly matched. Finally, the Google service account itself needs the API permissions, such as Storage Object Viewer. Newer clusters can bind IAM directly to the Kubernetes service account principal without annotation, which simplifies the setup.

Likely follow-ups

- What is the principal format for direct binding?
- Why does the metadata server matter?

Check that Workload Identity is enabled on the cluster and node pool, the Kubernetes service account is annotated with the Google service account, the IAM binding grants roles/iam.workloadIdentityUser to the right namespace and name, and the GSA has the needed permissions.

```
kubectl get sa app -n prod -o yaml | grep iam.gke.io
```

```
gcloud iam service-accounts get-iam-policy app-gsa@proj.iam.gserviceaccount.com
```

```
kubectl exec <p> -- curl -s -H 'Metadata-Flavor: Google'  
http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email
```

Interview tip: The member format is serviceAccount:PROJECT.svc.id.goog[NAMESPACE/KSA]. A typo there is the most common cause.

 Official docs

- [Azure: Managed identities](#)

Q75 **The GKE cluster autoscaler is not adding nodes though pods are Pending. What are the usual causes on GCP?**

GKE

 Spoken answer · ~57 s

The pod's NotTriggerScaleUp event explains why existing pools were rejected. If the pod requests more resources than the largest machine in any pool, no scale-up can help, so I fix the request or add a larger pool. If a node selector, taint or affinity matches no pool, autoscaling cannot satisfy it. If the pool is at its maximum size, I raise it. Then I check Google Cloud causes: project quotas for CPUs, in-use IP addresses or persistent disk capacity, and zonal resource exhaustion where the machine type is temporarily unavailable, which the autoscaler logs as a scale-up failure. A PersistentVolume bound to a specific zone limits where the pod can go. Remedies are raising quotas, spreading pools across zones and using several machine families, enabling node auto-provisioning, and adding alerts on pending pods and autoscaler errors.

Likely follow-ups

- How does a zonal stockout appear?
- What does NotTriggerScaleUp mean?

Quota limits (CPUs, IPs, disks), max node count reached, no node pool that fits the pod's requests, selectors or zone constraints, a PVC bound to a zone with no capacity, or stockouts of a machine type in a zone.

```
kubectl get events -A --field-selector reason=NotTriggerScaleUp
```

```
gcloud container clusters describe prod --region europe-west1 --  
format='yaml(autoscaling,nodePools[].autoscaling)'
```

```
gcloud compute regions describe europe-west1 --format='yaml(quotas)' | grep -B1 -A1  
'CPUS\|IN_USE_ADDRESSES'
```

Interview tip: Use multiple zones and machine types, or node auto-provisioning, to reduce zonal stockout failures.

Official docs

- [Kubernetes: Debug Pods](#)
- [Kubernetes Autoscaler: Cluster Autoscaler](#)
- [GKE overview](#)

Q76 **Cloud Run cannot connect to Cloud SQL or a private service in your VPC. How do you set up connectivity?**

GKE

 Spoken answer · ~62 s

For Cloud SQL, I can attach the instance to the Cloud Run service, which exposes a Unix socket at slash cloudsql, and the application connects through it using the connection name. The service account needs the Cloud SQL Client role, and for private IP instances the service also needs VPC connectivity. For other private resources, such as internal load balancers, Memorystore or VMs, I configure Direct VPC egress, which assigns the service addresses from a subnet, or a Serverless VPC Access connector. I choose whether all egress or only private ranges go through the VPC, because sending all traffic through requires Cloud NAT for internet access. Firewall rules must allow traffic from the subnet or connector range to the destination. If connections fail, I check the service account roles, the egress setting, DNS resolution for private names, connection limits, and that the instance has the right IP configuration.

Likely follow-ups

- Direct VPC egress versus connector?
- Why might all-traffic egress break internet calls?

Cloud Run is serverless and outside your VPC by default. Use the Cloud SQL connection (unix socket via the Auth proxy built-in) for Cloud SQL, and Direct VPC egress or a Serverless VPC Access connector for private IPs, with correct firewall rules and egress setting.

```
gcloud run deploy api --image=... --add-cloudsql-instances=proj:euope-west1:db
```

```
gcloud run deploy api --image=... --network=prod --subnet=run-subnet --vpc-egress=private-ranges-only
```

```
gcloud run services describe api --region europe-west1 --format='yaml(spec.template.metadata.annotations)'
```

Interview tip: The Cloud Run service account needs the Cloud SQL Client role, a frequent missing piece.

 Official docs

- [Cloud SQL](#)
- [Cloud Run](#)

Q77 **A Pub/Sub subscription backlog keeps growing. How do you investigate and fix it?**

Observability

 Spoken answer · ~59 s

I start with the metrics: undelivered message count, oldest unacked message age, delivery attempts, and ack and nack rates compared with the publish rate. If subscribers are healthy but too few, I scale them out, or switch to streaming pull with larger flow control settings. If messages are repeatedly redelivered, the ack deadline may be shorter than the processing time, so I extend it or use lease extensions in the client library. A poison message that always fails can block ordered delivery or consume retries, so I add a dead-letter topic with a maximum delivery attempt count and investigate the dead letters. Downstream bottlenecks, such as a slow database, are also a common cause, so I check dependency latency and apply backpressure. I set alerts on message age and dead-letter volume, and make handlers idempotent since delivery is at least once.

Likely follow-ups

- Why is delivery at least once?
- How does ordering affect the backlog?

Compare publish rate to ack rate, check oldest unacked message age, subscriber errors and ack deadline expirations, then scale subscribers, tune flow control, fix poison messages with dead-letter topics, and look at downstream dependencies.

```
gcloud monitoring time-series list --  
filter='metric.type="pubsub.googleapis.com/subscription/num_undelivered_messages"'
```

```
gcloud pubsub subscriptions describe orders-sub --  
format='yaml(ackDeadlineSeconds,deadLetterPolicy,retryPolicy)'
```

```
gcloud pubsub subscriptions update orders-sub --dead-letter-topic=orders-dlq --max-  
delivery-attempts=5
```

Interview tip: Alert on `oldest_unacked_message_age`, not just message count, since age directly maps to user-visible delay.

 Official docs

- [Pub/Sub](#)
- [RHEL 9 documentation](#)

 Spoken answer · ~61 s

Cloud NAT translates private addresses to external IPs by allocating source ports. A NAT IP address has around sixty-four thousand ports, and with a configured minimum ports per VM, each VM gets a block. Many simultaneous connections to the same destination IP, port and protocol can consume all of a VM's ports, and then new connections are dropped, which shows in NAT logs as dropped allocation status and in the dropped-sent-packets metric. Solutions are to enable dynamic port allocation so busy VMs can get more ports, raise minimum ports per VM, add more NAT IP addresses or use automatic allocation, shorten timeouts for idle connections, and most importantly use connection pooling and keep-alive in clients so they don't open a new connection per request. For GKE, I also consider whether many pods behind one node share the port range. Monitoring the allocation metrics provides early warning.

Likely follow-ups

- What is dynamic port allocation?
- Why do many connections to one destination cause the problem?

Each NAT IP provides about 64,512 ports, divided among VMs. When a VM exhausts its allocated ports to the same destination, new connections fail. Raise minimum ports per VM, add NAT IPs, enable dynamic port allocation, and reuse connections.

```
gcloud compute routers nats describe nat1 --router=rtr1 --region=europe-west1
```

```
gcloud compute routers nats update nat1 --router=rtr1 --region=europe-west1 --enable-dynamic-port-allocation --min-ports-per-vm=128 --max-ports-per-vm=4096
```

```
gcloud logging read 'resource.type="nat_gateway" AND jsonPayload.allocation_status="DROPPED"' --limit 10
```

Interview tip: Connection pooling and keep-alive in the application is the cheapest fix, since it removes the need for so many ports.

 Official docs

- [RFC 3022: Traditional NAT](#)
- [Cloud NAT](#)

Q79 **A request is denied by VPC Service Controls and the team cannot reach a service from their CI system. How do you investigate and fix it safely?**

Security

🗣 Spoken answer · ~64 s

VPC Service Controls draw a perimeter around projects and services to block data exfiltration, so a request from outside, such as a CI runner in another project or from an unknown network, gets a 403 with a unique violation identifier. I search the audit logs for that ID, which names the caller identity, the source project or IP, the target service and method, and the reason, for example not in the same perimeter or no matching access level. Then I choose the narrowest fix: add the CI project into the perimeter, create an ingress rule allowing that service account to call specific methods on specific services, or an egress rule for legitimate outbound access, or an access level based on IP or device. I test the change in dry-run mode, review the logs, then enforce. I avoid disabling the perimeter or adding broad allowances, and I document each exception with an owner and expiry.

Likely follow-ups

- Ingress versus egress rules?
- What is dry-run mode?

🚧 VPC Service Controls



Look up the violation in audit logs (Policy Denied) for the unique ID, which shows the source identity, network, service and which rule it breached. Fix with an ingress or egress rule scoped to the specific identity, project and method, or an access level, rather than loosening the perimeter.

```
gcloud logging read  
'protoPayload.metadata.violationReason="RESOURCES_NOT_IN_SAME_SERVICE_PERIMETER"' --limit 5  
--format=json
```

```
gcloud access-context-manager perimeters describe prod-perimeter --policy=POLICY_ID
```

```
gcloud access-context-manager perimeters dry-run update prod-perimeter --policy=POLICY_ID -  
-add-ingress-policies=ingress.yaml
```

Interview tip: Test changes in dry-run mode first, which logs would-be violations without enforcing.

📖 **Start here (official docs)**

- [Google Cloud documentation](#)

Q80 **A deployment fails with 'Permission denied' even though the engineer has Owner. Which other layers could be blocking it?**

IAM & identity

 Spoken answer · ~61 s

Having a powerful role is only one input. The error text tells me which layer refused. A message naming a constraint, such as restricting external IPs or allowed regions, comes from an organization policy, which applies regardless of IAM role. A deny policy at a higher level blocks specific permissions even for owners. VPC Service Controls can reject calls from outside the perimeter. An API that is not enabled in the project returns a different error, as do billing being disabled or quota exhaustion. Policy Troubleshooter is the tool to confirm the IAM side, as it shows allow, deny and condition evaluation. If it is an org policy, I either change the design, for example use a private IP and NAT, or request an exception with the right approvers, applying it at the lowest sensible scope. I avoid granting more roles to work around a guardrail.

Likely follow-ups

- What is the difference between org policy and IAM?
- How do you request an org policy exception?

Owner can still be blocked by organisation policies, IAM deny policies, VPC Service Controls, missing API enablement, billing issues, or quotas. Use Policy Troubleshooter and the error's details to find which layer denied the call.

```
gcloud policy-troubleshoot iam //cloudresourcemanager.googleapis.com/projects/my-proj --  
principal-email=a@corp.com --permission=compute.instances.create
```

```
gcloud resource-manager org-policies list --project=my-proj
```

```
gcloud services list --enabled --project=my-proj | grep compute
```

Interview tip: Read the full error. Organisation policy errors name the constraint, like constraints/compute.vmExternalIpAccess.

 Start here (official docs)

- [Google Cloud documentation](#)

 Spoken answer · ~60 s

The first error means the API is disabled in the project, or was just enabled and has not propagated. I manage API enablement in Terraform with `google_project_service` resources, and make dependent resources depend on them, sometimes with a short delay or retry. For permission errors, I check which identity Terraform is actually using: a user's application default credentials, a service account key, or impersonation of a deployer service account. That identity needs the right roles in the target project, plus Service Usage Consumer, and the quota project setting must point to a project where the API is enabled. For reliability, I use a dedicated deployer service account through impersonation or workload identity federation in CI, split a bootstrap stack that creates projects, APIs and service accounts from the application stacks, store state in a versioned bucket with locking, and run plan checks before apply.

Likely follow-ups

- Why impersonate a service account in Terraform?
- What is the quota project?

Enable required APIs explicitly with `google_project_service`, wait for propagation using dependencies, grant the Terraform service account the needed roles, and check the right quota project and impersonation settings. Split bootstrap from application stacks.

```
gcloud services enable compute.googleapis.com container.googleapis.com --project my-proj
```

```
terraform apply -var=impersonate_sa=tf-deployer@admin-proj.iam.gserviceaccount.com
```

```
gcloud auth application-default set-quota-project my-proj
```

Interview tip: Use `depends_on` or the service resource output as a dependency so resources do not race the API enablement.

 Official docs

- [Kubernetes: RBAC](#)
- [Kubernetes: Service accounts](#)

Q82 **GKE cannot pull images from Artifact Registry and shows 403 or ImagePullBackOff. What do you check?**

CI/CD & IaC

 Spoken answer · ~65 s

The pod events show whether it is an authorization error, a not found error, or a network timeout. For a 403, the identity pulling images is the node's service account, so it needs the Artifact Registry Reader role on the repository or project, which is easy to forget when using a custom node service account instead of the default one. For cross-project pulls, the grant has to be on the project that owns the repository. For not found, I verify the full path, which includes location, project, repository, image name and tag, and that the tag exists. For timeouts in a private cluster, nodes without external IPs need Private Google Access or Cloud NAT to reach the registry endpoint. VPC Service Controls may also block the pull if the registry is in a different perimeter. I fix the specific cause and test with a manual pull from a debug pod or with `crictl` on a node.

Likely follow-ups

- Which identity pulls images on GKE?
- How do you pull across projects?

The node service account needs Artifact Registry Reader on the repository, the repository must be in a reachable location, private clusters need Private Google Access, and image paths must be exact. Check cross-project permissions and VPC Service Controls.

```
kubectl describe pod <p> | grep -A5 Events
```

```
gcloud artifacts repositories get-iam-policy repo --location=europe-west1
```

```
gcloud artifacts docker images list europe-west1-docker.pkg.dev/proj/repo
```

Interview tip: With custom node service accounts, remember to grant them the Reader role. Only the default account may have had it implicitly.

 Official docs

- [Kubernetes: Debug Pods](#)
- [Kubernetes: Images \(pull policy, registries\)](#)
- [Kubernetes: Pull an image from a private registry](#)

Q83 **A large GCP network egress charge appeared. How do you find the traffic and reduce it?**

Cost

 Spoken answer · ~63 s

I begin by identifying which kind of egress it is, using the billing export grouped by SKU. Internet egress, inter-region transfer and inter-zone traffic have different prices and different fixes. Then I enable VPC Flow Logs, with sampling, on the relevant subnets and query them in BigQuery to find top talkers by source, destination and bytes, and use Network Intelligence Center to view the topology. Typical causes are large downloads served straight from instances or buckets without caching, chatty services spread across regions or zones, backups or replication to another region, and analytics exports. Remedies include putting Cloud CDN or an edge cache in front of static content, serving from the same region as the consumers, compressing payloads, using zonal affinity so traffic stays within a zone, moving heavy data jobs next to the data, and choosing the correct network service tier. I add a budget alert for network SKUs.

Likely follow-ups

- Premium versus Standard network tier?
- How do you keep traffic within a zone?

Use billing export by SKU to see whether it is internet egress, inter-region, inter-zone or CDN. Then VPC Flow Logs and Network Intelligence to find sources and destinations. Reduce with Cloud CDN, regional co-location, compression, caching, and private paths.

```
bq query --use_legacy_sql=false 'SELECT sku.description, SUM(cost) c FROM
`billing.gcp_billing_export_v1_X` WHERE service.description="Compute Engine" AND
sku.description LIKE "%Egress%" GROUP BY 1 ORDER BY c DESC'
```

```
gcloud compute networks subnets update app --region=europe-west1 --enable-flow-logs --
logging-aggregation-interval=interval-5-min
```

```
gcloud compute backend-services update web --enable-cdn --global
```

Interview tip: Cross-zone and cross-region traffic inside the same app is a quiet cost, so check topology before blaming the internet.

 Start here (official docs)

- [Google Cloud documentation](#)

Q84 **Cloud Logging costs have exploded. How do you identify the source and reduce them without losing important logs?**

Cost

 Spoken answer · ~63 s

I look at the ingestion volume metrics, which break bytes down by resource type and log name, to find the main producers. Common ones are a service accidentally left at debug level, GKE container stdout from a crash-looping pod, Data Access audit logs on busy services, VPC flow logs at a high sampling rate, and load balancer logs for every request. For each source, I decide the right action: reduce the log level in the application, fix the loop, lower the flow log sampling, or add exclusion filters on the Default sink to drop low-value entries such as health check requests, with sampling when I still want a fraction. Some logs are still required for audit or security, so I route those through a sink to a cheaper destination like Cloud Storage for retention, set shorter retention on the Default bucket, and create budget alerts on Logging spend. I review volume monthly.

Likely follow-ups

- Exclusion filter versus sampling?
- What must never be excluded?

Use Logs Explorer and the log volume metrics to see bytes ingested by resource and log name. Typical culprits are debug logging, load balancer or VPC flow logs, and chatty GKE containers. Add exclusion filters, sampling, routing to cheaper storage, and retention changes.

```
gcloud logging metrics list
```

```
gcloud logging sinks update _Default --add-exclusion=name=drop-health,filter='httpRequest.requestUrl=~"/healthz"'
```

```
gcloud logging buckets update _Default --location=global --retention-days=14
```

Interview tip: The logging/billing/bytes_ingested metric by resource_type quickly shows who generates the volume.

 Official docs

- [Cloud Logging](#)

 Spoken answer · ~65 s

First I describe the sink to confirm it is enabled and see its filter and writer identity. Then I test the filter in Logs Explorer to be sure matching entries exist for the time period. The most common problem is permissions: the writer identity, a service account unique to the sink, must have the BigQuery Data Editor role on the destination dataset, and if the sink is at the organization level with include children, I need to check that the identity is unique per sink. The destination dataset must exist, and be in a location that the sink supports, and for partitioned tables the settings must match. Exclusion filters on the sink or on the log bucket can drop entries, and organization policies such as restricting resource locations can block creation of tables. Sink errors are written to the logs of the project, so I search there. After fixing, new entries should appear within minutes.

Likely follow-ups

- Why is the writer identity unique?
- Can a sink export old logs?

Verify the sink's filter matches entries, the sink's writer identity has BigQuery Data Editor on the dataset, the destination dataset exists in an allowed location, the sink is not disabled or excluded upstream, and check for sink errors in the logs.

```
gcloud logging sinks describe audit-to-bq
```

```
bq show --format=prettyjson proj:audit | grep -i location
```

```
gcloud logging read 'logName:"logging.googleapis.com%2Fsink_error"' --limit 5
```

Interview tip: The writer identity is created per sink. Copy it from the describe output and grant it on the destination, every time.

 Official docs

- [BigQuery](#)
- [Cloud Logging](#)

Q86 **A managed instance group keeps recreating instances in an autohealing loop. How do you stop it and find the cause?** Compute

 Spoken answer · ~63 s

A loop means each new VM fails its health check and is replaced before it can stabilise. I list the group's errors and recent operations to confirm autohealing is the actor. To stop the churn while I investigate, I temporarily remove autohealing or lengthen the initial delay. Then I take a failing instance, check the serial console output and startup script logs, and curl the health check path from the instance. Typical causes: the initial delay is shorter than boot plus warm-up time, the health check uses the wrong port or path or expects a status the app does not return, the firewall does not allow the Google health check ranges, a startup script fails because of a missing secret or unreachable dependency, or the app is overloaded and fails checks under load. After fixing it, I re-enable autohealing with a realistic delay and thresholds, and bake images to shorten boot time.

Likely follow-ups

- Which ranges do health checks come from?
- Why bake images instead of using startup scripts?

Instances are failing the autohealing health check, often because the initial delay is too short, the check path or port is wrong, a firewall blocks health check ranges, or the app crashes under load. Disable autohealing temporarily, inspect an instance, then fix.

```
gcloud compute instance-groups managed list-errors web-mig --region=europe-west1
```

```
gcloud compute instance-groups managed update web-mig --clear-autohealing --region=europe-west1
```

```
gcloud compute instances get-serial-port-output <vm> --zone=europe-west1-b | tail -50
```

Interview tip: The serial port output shows startup script failures, which are a frequent root cause for new instances never becoming healthy.

 Official docs

- [Compute Engine: Managed instance groups](#)

Q87 **A Cloud VPN tunnel is established but BGP is down or routes are missing on-premises. How do you troubleshoot?**

Networking

 Spoken answer · ~59 s

I separate the layers. First, tunnel status: if it is not established, the IKE settings, shared secret, peer IP, or on-premises firewall rules for UDP 500 and 4500 are wrong, and the VPN logs show negotiation failures. If the tunnel is established but BGP is down, I compare both sides on the ASNs, the link-local tunnel inside addresses, any authentication, and verify that TCP port 179 is allowed through the tunnel. Once BGP is up but routes are missing, I look at what Cloud Router advertises: by default it advertises subnets according to the dynamic routing mode, so I may need custom advertised ranges, and at the learned routes from the peer. Overlapping address ranges, route priority conflicts, and regional versus global routing mode can cause surprises. For production I use HA VPN with two tunnels and BGP and test failover.

Likely follow-ups

- HA VPN versus classic VPN?
- What does global dynamic routing change?

Check tunnel status and IKE logs, then BGP peer settings: ASNs, link-local IPs, MD5 if used, and Cloud Router advertised routes. Verify firewalls on both sides and that overlapping ranges are not present. Use route and learned-route views.

```
gcloud compute vpn-tunnels describe tun1 --region=europe-west1 --  
format='yaml(status,detailedStatus)'
```

```
gcloud compute routers get-status rtr1 --region=europe-west1
```

```
gcloud compute routers update-bgp-peer rtr1 --peer-name=peer1 --region=europe-west1 --  
advertisement-mode=CUSTOM --set-advertisement-ranges=10.10.0.0/16
```

Interview tip: Tunnel up and BGP down usually points to an ASN or link-local address mismatch, or a firewall dropping TCP 179.

 Official docs

- [RFC 4271: BGP-4](#)
- [Cloudflare Learning: What is BGP?](#)
- [Cloudflare Learning: What is a VPN?](#)

Q88 **A Compute Engine persistent disk is almost full. How do you grow it online and avoid a repeat?**

Storage & data

 Spoken answer · ~61 s

I start by snapshotting the disk, since resizing is generally safe but a restore point costs little. Then I increase the size with the `resize` command, which can be done while the disk is attached and the VM is running. Inside the guest, I extend the partition with `growpart` if the filesystem sits on a partition, and then grow the filesystem with `resize2fs` for ext4 or `xfs_growfs` for XFS. The size cannot be reduced later, so I increase by a sensible amount. For the root cause, I check what consumed the space, using `du`, looking at logs, temp files, database data or deleted-but-open files, and fix log rotation or add retention. Disk utilisation is not visible without an agent, so I install the Ops Agent, create an alert at eighty percent, and consider moving growing data to a separate data disk with automation to expand it.

Likely follow-ups

- Can you shrink a persistent disk?
- How does the Ops Agent expose disk usage?

Resize the disk with `gcloud`, which works while attached, then grow the partition and filesystem in the guest. Take a snapshot first. Find the cause of growth and add monitoring with the Ops Agent for disk usage alerts.

```
gcloud compute disks snapshot data1 --zone=europe-west1-b --snapshot-names=pre-resize
```

```
gcloud compute disks resize data1 --size=600GB --zone=europe-west1-b
```

```
sudo growpart /dev/sdb 1 && sudo resize2fs /dev/sdb1
```

Interview tip: Disks can only grow, never shrink, so go up in measured steps.

 Official docs

- [Compute Engine: Managed instance groups](#)

Q89 **A Security Command Center finding reports a publicly accessible Cloud Storage bucket. How do you respond?**

Security

 Spoken answer · ~62 s

First I validate the finding: look at the bucket policy for allUsers or allAuthenticatedUsers, and find out with the owner whether the exposure was intended, such as a static website, or an accident. If it is not intended, I immediately remove the public bindings and turn on public access prevention, plus uniform bucket-level access so ACLs on individual objects cannot re-expose data. Then I assess impact: review Data Access and request logs, if they were enabled, to see which objects were fetched and from where, classify what data was inside with Sensitive Data Protection, and involve incident response or privacy teams if needed. To prevent a recurrence, I enforce the organization policy for public access prevention, scan Terraform in CI for public bindings, set Security Command Center notifications to a chat or ticketing channel, and educate teams about safer sharing options such as signed URLs or a CDN.

Likely follow-ups

- What is Sensitive Data Protection?
- Why prefer uniform bucket-level access?

Confirm whether the exposure is intended, then remove public access, enable public access prevention and uniform bucket-level access, check access logs for what was read, and fix the pipeline or policy that allowed it. Use org policy to stop recurrence.

```
gcloud storage buckets get-iam-policy gs://my-bucket --format=json | grep -E  
'allUsers|allAuthenticatedUsers'
```

```
gcloud storage buckets update gs://my-bucket --public-access-prevention --uniform-bucket-  
level-access
```

```
gcloud resource-manager org-policies enable-enforce storage.publicAccessPrevention --  
organization=123
```

Interview tip: Check Data Access audit logs, if enabled, to see whether anyone actually read objects while it was public.

 Official docs

- [Cloud Storage](#)
- [Security Command Center](#)

Q90 **Spot VM or Spot GKE node preemptions are causing batch jobs to fail. How do you make them resilient?**

Compute

 Spoken answer · ~59 s

With Spot capacity, interruptions are normal, so the design must expect them. I split big jobs into smaller idempotent tasks put on a queue such as Pub/Sub or Cloud Tasks, so if a worker disappears, its unacknowledged task is delivered again. For long computations, I checkpoint intermediate state to Cloud Storage regularly and resume from the latest checkpoint. A shutdown script or signal handler uses the thirty-second warning to flush state and release locks. To reduce correlated loss, I spread workers across several zones and machine types in a regional managed instance group or multiple GKE node pools. I keep a small on-demand pool or fallback so deadlines are met even when Spot capacity is scarce, and set sensible retry limits and backoff. For Kubernetes, Job backoffLimit and Indexed Jobs help. I track preemption rates and the cost-per-completed-job to verify the savings.

Likely follow-ups

- How do Indexed Jobs help?
- What belongs in a checkpoint?

Make work restartable: checkpoint progress, make tasks idempotent, use a queue so unfinished tasks return, handle the shutdown signal to save state, diversify zones and machine types, and keep a small on-demand pool to guarantee completion.

```
gcloud compute instances describe worker1 --zone=europe-west1-b --format='value(scheduling.provisioningModel)'
```

```
kubectl get pods -o wide --field-selector status.phase=Failed
```

```
gcloud logging read 'protoPayload.methodName="compute.instances.preempted"' --limit 10
```

Interview tip: Measure preemption frequency. If a job is shorter than the typical time between preemptions, retries are cheap.

 Official docs

- [Kubernetes: Pod priority and preemption](#)
- [GKE overview](#)