

## 1. Explain how the Kubernetes control plane handles cluster convergence.

### Answer:

Cluster convergence means the control plane continuously drives the cluster toward the desired state expressed in manifests.

- **kube-apiserver** authenticates, authorizes, validates and persists desired state in etcd. Further it can be extended to implement features like admission controller.
  - **Controllers** (ReplicaSet, Deployment, Node, Endpoint, etc.) compare desired vs. actual.
  - **kubelet** ensures PodSpec on each node is running as instructed.
  - **Scheduler** converges by assigning unbound Pods to nodes.  
All components run reconciliation loops (eventually consistent).
- 

## 2. What is the role of the *kube-controller-manager* and how does it scale controllers?

### Answer:

*kube-controller-manager* runs multiple controllers (Deployment, Node, Job, ServiceAccount, TTL controller, etc.) each acting independently.

- All controllers use **shared informers** to watch for relevant object changes.
  - They scale using **rate-limiting work queues** with exponential backoff and parallel workers.
  - Controllers are horizontally unscalable unless you use **ComponentConfig + leader election** to run multiple replicas.
- 

## 3. How does the Kubernetes scheduler score nodes for pod placement?

### Answer:

Scheduler logic:

### 1. Filtering (Predicates):

- Node resources

- Taints/Tolerations
- NodeSelector/NodeAffinity
- VolumeBinding

## 2. Scoring (Priorities):

- LeastRequestedPriority
  - NodeAffinityPriority
  - TopologySpreadPriority
  - ImageLocalityPriority
  - Custom scheduler plugins via **Scheduling Framework**  
Scores are normalized and highest-scoring node gets the pod.
- 

## 4. Explain how CRDs work internally and how controllers extend Kubernetes.

### Answer:

- CRDs define new resource types stored in **etcd** like native resources.
  - The API server dynamically creates REST endpoints.
  - Custom controllers watch CRDs, use reconciliation loops, and enforce desired state.
  - Operators = CRD + controller, encoding domain logic (DB provisioning, cluster mgmt).
- 

## 5. Describe the etcd architecture and how Kubernetes ensures consistency.

### Answer:

- etcd is a **distributed key-value store** using **Raft consensus**.
  - Only the leader handles writes. Followers replicate logs.
  - Guarantees **linearizable reads** when --consistent-read=true.
  - Control-plane components rely on watches for efficient state change detection.
  - Snapshots + WAL replay ensure durability.
-

## 6. How does Kubernetes handle multi-AZ or multi-region failures?

**Answer:**

- For multi-AZ: Deploy multiple control plane nodes in different AZs.
  - For multi-region: Kubernetes does **not** support multi-region clusters natively. Use:
    - **Cluster federation v2**
    - External traffic mgmt (GSLB / Cloud LB)
    - Global service mesh (Istio multi-primary)
  - etcd cannot span high-latency regions.
- 

## 7. What are ephemeral containers and how are they different from sidecars?

**Answer:**

- Ephemeral containers are injected for **debugging only**, not restarted, not part of PodSpec, and no guarantees.
  - Sidecars are part of PodSpec, lifecycle-managed, restarted, and participate in networking and volumes.
- 

## 8. How does Kubernetes implement zero downtime rolling updates?

**Answer:**

Deployment controller:

- Creates new ReplicaSet
  - Gradually increases new RS replicas
  - Decreases old RS replicas
  - Ensures `maxUnavailable` and `maxSurge` are respected  
Readiness probes ensure new pods are not added to Service endpoints until healthy.
- 

## 9. Explain Pod topology spread constraints.

**Answer:**

Ensures Pods are distributed across fault domains (zones, nodes).

- Constraints on labels: `topologyKey`, `whenUnsatisfiable`, `labelSelector`.
  - Scheduler tries to minimize skew across domains.
  - Useful for HA of replicated workloads.
- 

## 10. Describe how Kubernetes handles secret encryption at rest.

**Answer:**

- API server encrypts secrets before storing in etcd using **EncryptionConfiguration**.
  - Providers: AES-CBC, kms-provider, identity, aescbc.
  - Key rotation: update config → restart API server → re-encrypt with `encrypt secrets` command.
- 

## 11. What happens internally when you run `kubectl apply`?

**Answer:**

- Converts manifest into an object.
  - Computes a 3-way diff using `last-applied-configuration` annotation.
  - Sends patch request to API server.
  - API server validates schema via OpenAPI.
  - Object stored in etcd; controllers reconcile.
- 

## 12. Explain the pod lifecycle phases in deep detail.

**Answer:**

- **Pending:** Awaiting scheduling / image pull.
- **Running:** At least one container running.
- **Succeeded/Failed:** All containers completed.
- **Unknown:** Node unreachable.

Container states include Waiting, Running, Terminated.  
Kubelet reports state transitions using CRI.

---

## 13. How does kube-proxy work in iptables vs IPVS mode?

**Answer:**

**iptables:**

- Uses DNAT rules; performance degrades with large Services.
- No session persistence beyond simple round-robin.

**IPVS:**

- Uses kernel-level LVS.
  - Faster, supports LRU scheduling, persistence, and health checks.
  - Handles large-scale services better.
- 

## 14. Describe how service discovery works at the DNS and cluster networking level.

**Answer:**

- CoreDNS watches kube-apiserver.
  - Generates DNS A records like: `myservice.myns.svc.cluster.local`.
  - kube-proxy programs iptables/IPVS rules mapping ClusterIP → Endpoints.
  - Container queries DNS → resolves ClusterIP → NAT to backend pod.
- 

## 15. Explain how CNI plugins implement networking.

**Answer:**

CNI plugin responsibilities:

- Create veth pair for pod
- Assign IP to pod
- Add routes
- Apply network policies

Examples:

- Calico (BGP routing + policies)

- Cilium (eBPF dataplane)
  - Weave (mesh overlay)
- 

## 16. How does Kubernetes handle logs at scale?

**Answer:**

Options:

- Node-level: container runtime stores in `/var/log/containers`.
  - Sidecar logging pattern.
  - Cluster logging: Fluentd/Fluentbit → Elasticsearch/OpenSearch/Loki.
  - Structured logging integrated with API server audit logs.
- 

## 17. What is the priority & preemption mechanism?

**Answer:**

- Pods have priority classes.
  - Scheduler may evict lower-priority pods to accommodate higher-priority pods.
  - Preemption respects PodDisruptionBudgets and PDBs can prevent eviction.
- 

## 18. How do PodDisruptionBudget and eviction API interact?

**Answer:**

- PDB defines minAvailable or maxUnavailable.
  - Eviction API is triggered by kubelet or admin.
  - API server denies eviction if it violates PDB.
  - Node drains respect PDBs.
- 

## 19. How does the Kubelet manage container garbage collection?

**Answer:**

- Image GC and container GC triggered based on thresholds:
    - DiskPressure
    - ImageRemovalPolicy
  - Removes unused images + dead containers.
  - Uses container runtime (containerd) for cleanup.
- 

## 20. Describe the sidecar container lifecycle and startup order issues.

**Answer:**

- All containers start independently; no guaranteed start order.  
Patterns to handle dependency:
    - Init containers
    - Probes to block readiness
    - Sidecar startup scripts  
Improper order causes service-mesh or logging issues.
- 

## 21. Explain the difference between NodePort, LoadBalancer, and Ingress.

**Answer:**

- **NodePort:** exposes service on <nodeIP>:<port>.
  - **LoadBalancer:** creates cloud LB → forwards to NodePort.
  - **Ingress:** Layer-7 routing, TLS termination, host/path rules.
- 

## 22. How does Kubernetes handle image pull backoff and retries?

**Answer:**

- Exponential backoff starting at 10s → 20s → 40s.
  - Controlled by Kubelet.
  - `imagePullPolicy` drives behaviour (Always / IfNotPresent / Never).
-

## 23. Explain the container runtime interface (CRI) architecture.

Answer:

- Kubelet communicates with container runtime (containerd, CRI-O) using gRPC.
  - CRI components: ImageService, RuntimeService.
  - shim process isolates container lifecycle from kubelet restarts.
- 

## 24. How does Kubernetes avoid split-brain scenarios?

Answer:

- etcd quorum (majority) ensures only one leader.
  - API server is stateless → no leader issues.
  - Controllers use leader election to avoid multi-writer conflicts.
- 

## 25. Describe a complex real-world debugging scenario only advanced engineers face.

Answer:

Example:

Pods getting restarted randomly across nodes.

Diagnosis path:

1. Check events → OOMKilled?
2. Analyze host-level dmesg → kernel memory pressure.
3. Check node allocatable vs kube-reserved/system-reserved.
4. Discover CNI plugin leak causing memory exhaustion.
5. Fix by upgrading CNI version and adjusting node allocatable.

This tests cluster-wide troubleshooting skills.

## 26. Explain the exact sequence of events when the API server goes down while a controller is in the middle of reconciling an object. What consistency guarantees

## remain?

Answer:

- Controller's reconciliation is *stateless*; it requeues work if API server is unavailable.
  - Shared informers stop receiving events → controller falls back to resync loop.
  - No partial writes occur because the API server either commits or rejects with HTTP error; etcd ensures atomicity.
  - Consistency remains **eventual**, not strong; desired state may lag actual.
  - Controllers use optimistic concurrency via **resourceVersion**, preventing stale writes.
- 

## 27. How does Kubernetes handle a situation where thousands of Pods become unschedulable due to a constraint like NodeAffinity?

Answer:

- Scheduler marks pods as `Unschedulable` and adds them to the *unschedulable queue*.
  - Pod is moved back to active queue only when cluster state changes (node events).
  - Use of **pod preemption**, **taints/tolerations**, or **over-constrained affinities** identified via scheduler debugging APIs.
  - In large-scale clusters, scheduler backpressure prevents starvation.
- 

## 28. Describe how the EndpointSlice controller maintains consistency when large numbers of Pod endpoints churn rapidly.

Answer:

- EndpointSlice controller batches updates using **delayed work queues**.
- Uses **max endpoints per slice (100)** to split large services.
- Slices updated via strategic merge patches to prevent replacing entire objects.
- Controller ensures *idempotency*; duplicate loads do not cause drift.
- Slices use revision numbers for ordering.

---

## 29. What is the impact of extremely large ConfigMaps (hundreds of MBs) on kubelet, API server, and container startup?

**Answer:**

- API server performance degrades due to serialization overhead.
  - etcd disk I/O and memory usage spike → potential heartbeat delays → leader election timeouts.
  - kubelet pulls config via PodSpec → increases CPU usage and startup latency.
  - Processes reading mounted ConfigMap may block container readiness.
  - Recommended max: < 1 MB per ConfigMap.
- 

## 30. Explain the deep mechanics of Kubernetes Service VIP implementation at packet level in IPVS mode.

**Answer:**

- kube-proxy programs IPVS virtual servers with ClusterIP + port.
  - LVS uses netfilter hooks before iptables.
  - Packet flow:
    1. DNAT to backend Pod IP.
    2. Connection is tracked via conntrack.
    3. Subsequent packets bypass scheduling via session affinity.
  - IPVS supports TCP/UDP load balancing algorithms like rr, sh, dh, fq.
- 

## 31. How do API Priority and Fairness (APF) limit system overload during cluster degradation?

**Answer:**

- API server groups requests into "flows".
- Each flow bucket has:
  - Priority level

- Concurrency limit
  - Queuing policy (FIFO, shuffle sharded)
  - In overload, low-priority flows are throttled or rejected.
  - Prevents “API death spiral” where controllers DDOS the API server.
- 

## **32. Explain the exact mechanism used by SIGTERM → SIGKILL sequence in pod shutdown.**

**Answer:**

- Kubelet sends TERM to main container process PID 1.
  - Container runtime injects signal inside namespace.
  - Grace period countdown begins (default: 30s).
  - If container doesn’t exit → forceful SIGKILL.
  - PreStop hooks run **before** SIGTERM but counted towards grace period.
- 

## **33. What mechanisms exist to prevent “thundering herd” reloads during config changes in thousands of Pods?**

**Answer:**

- Staggered rollouts via:
    - maxUnavailable in Deployments
    - PodAntiAffinity
    - Rate-limited controllers
  - Application-level circuit breakers
  - Horizontal Pod Autoscaler cooldown windows
  - Node-level spread reduces simultaneous restarts.
- 

## **34. How does the watch API scale to tens of thousands of parallel clients without overloading the API server?**

**Answer:**

- Watches never return full objects repeatedly; they send deltas.
  - Shared informers inside controllers de-duplicate watchers.
  - API server employs etcd watch caching.
  - HTTP/2 multiplexing reduces connection overhead.
  - Built-in watch timeouts prevent resource leaks.
- 

### **35. Explain the implications of running a multi-tenant cluster where tenants share the same CRDs.**

**Answer:**

Key issues:

- CRDs do not have namespace-level schemas → conflict between tenants.
  - All CRDs share etcd storage; heavy usage from tenant A affects B.
  - No RBAC scoping for CRD schemas → accidental modifications break all tenants.
- Mitigation:
- Use aggregated APIs
  - Use virtual clusters (vcluster, Loft, Kamaji)
- 

### **36. What happens when a Pod is scheduled to a node but fails to start due to container runtime misconfiguration?**

**Answer:**

- kubelet marks pod as Failed with ContainerCannotRun.
  - Pod remains bound to the node → scheduler will NOT retry automatically.
  - Deployment/ReplicaSet trigger new pod only when failure is terminal.
  - Node events show CNI / CRI failures.
  - DaemonSet pods keep retrying indefinitely.
- 

### **37. How do Sidecar containers break when upgrading to sidecar-less proxies like Ambient Mesh (Istio ambient)?**

**Answer:**

- Existing sidecars intercept traffic; ambient mode uses L4/L7 mesh at node level.
  - Conflicts:
    - Double proxying
    - Pod-level policies may not translate
    - Broken mTLS identity propagation
  - Requires strict migration plan between mesh modes.
- 

## 38. Why does a Pod with shared PID namespace expose security risks?

**Answer:**

- All containers share same process table.
  - Container can send signals (kill, ptrace) to others.
  - Escape vectors: reading `/proc/<pid>/cmdline`, debugging processes.
  - Violates strict isolation; used only for debugging.
- 

## 39. Describe an edge case where a Pod is “Running” but traffic to it fails.

Examples:

- Readiness probe fails → Pod excluded from EndpointsSlice.
  - misconfigured NetworkPolicy denies traffic.
  - CNI failure creates veth but no route.
  - IP conflict across nodes due to overlay network bug.
  - kube-proxy not updating rules → stale endpoints.
- 

## 40. What happens during an etcd compaction, and how does it impact watches and controllers?

**Answer:**

- Compaction removes old revisions.

- Watches using older resourceVersion get “**410 Gone**”.
  - Controllers detect error → re-establish watch from latest state.
  - API server experiences temporary increased I/O.
- 

## **41. Explain a scenario where Horizontal Pod Autoscaler causes instability instead of scaling.**

**Answer:**

- Oscillating metrics cause thrashing.
  - HPA + Cluster Autoscaler work against each other → latency increases.
  - Slow-start of application causes premature scaling.
  - Missing metric normalization.
- 

## **42. Why is running stateful workload rebalancing hard in Kubernetes?**

**Answer:**

- StatefulSet pods have sticky identity bound to ordinal + hostname.
  - Even if node fails temporarily, Kubernetes avoids moving them.
  - ReadWriteOnce volumes restrict failover.
  - Controllers prioritize consistency over liveness.
- 

## **43. Explain why a pod can remain in “Terminating” state permanently.**

**Answer:**

- Finalizers stuck
  - Mounted PersistentVolumes fail to unmount
  - Container runtime bug that doesn’t kill processes
  - Network disconnect between kubelet and API server
  - Node stuck in NotReady and forced eviction disabled.
-

## 44. Describe a scenario where memory limits cause worse performance than having no limits at all.

Answer:

- Memory limit triggers kernel OOM sooner than actual available memory would require.
  - Throttling via cgroups v2 affects garbage-collected languages (Go, Java).
  - Leads to micro-OOM cycles → app freeze → tail latency spikes.
- 

## 45. How does the Kubelet's pod admission logic enforce node allocatable constraints?

Answer:

- Computes:  
$$\text{NodeAllocatable} = \text{Capacity} - \text{kube-reserved} - \text{system-reserved} - \text{eviction-thresholds}$$
  - Pod admission checks requests, not limits.
  - Guaranteed pods admitted first during eviction.
  - Race conditions solved with atomic admission check locks.
- 

## 46. Why does PVC binding sometimes occur before scheduling and sometimes after?

Answer:

- VolumeBindingMode controls behavior:
    - Immediate: bind before scheduling – may cause node qualification failures.
    - WaitForFirstConsumer: scheduler picks node → PV binding considers topology → reduces cross-zone mismatch.
- 

## 47. What happens when two controllers manage overlapping fields of the same object?

Answer:

- ManagedFields keeps track of field ownership.

- `kubectl apply` may conflict with controller-managed fields.
  - Controller fights → infinite reconciliation → high API server load.
  - Fix: move to server-side apply with proper field ownership.
- 

## 48. Explain how eviction due to local ephemeral storage pressure differs from memory pressure.

**Answer:**

**Ephemeral storage eviction:**

- kubelet evicts based on `/var/lib/kubelet`, container logs, `emptyDir` usage.
- Reclaims storage by deleting pod's writable layers.

**Memory pressure eviction:**

- Kernel OOM kills process.
  - kubelet marks pod as `OOMKilled`.
  - No reclamation beyond termination.
- 

## 49. What causes persistent NetworkPolicy rule inconsistencies across nodes?

**Answer:**

- CNI plugin race conditions
  - Node reboot causing stale iptables rules
  - Multiple CNIs chained incorrectly
  - Version skew between CNI agents across nodes
  - Large NetworkPolicy sets → rule explosion.
- 

## 50. Deep explain: how does the cluster-autoscaler decide which node to remove without causing application downtime?

**Answer:**

- Picks underutilized nodes via heuristics.

- Simulates eviction of all pods from the node.
- Checks:
  - PodDisruptionBudgets
  - Pod affinity/anti-affinity
  - Local PV constraints
  - DaemonSet pods (non-evictable)
- Only drains node if **all** pods are safely re-schedulable.
- Uses exponential backoff to avoid thrashing.