1. What is live VM migration? Describe the process through "iterative copy" with an illustration? Why shared storage between source and destination VM is required for live migration?

- Live VM migration moves a running virtual machine between physical hosts with near-zero downtime by copying its in-memory state while the VM keeps serving traffic; the hypervisor coordinates memory transfer, device state synchronization, and a brief switchover.
- Iterative copy (pre-copy) starts by copying all guest memory pages to the destination while the VM still runs; dirty pages (written again during copy) are tracked via page tables/dirty-bit logging.
- Subsequent iterations copy only pages dirtied since the last round, shrinking the dirty set each pass; convergence criteria are based on a threshold of remaining dirty pages or a maximum number of rounds.
- When the dirty set becomes small enough, the hypervisor pauses the VM for a stop-and-copy phase, transfers CPU/device state and the remaining pages, updates ARP/ND and routing, and resumes the VM on the destination.
- Shared storage (e.g., NFS, iSCSI, SAN, or a distributed filesystem) ensures the VM’s virtual disks are consistently available to both source and destination, avoiding large disk copy during migration and guaranteeing block-level consistency.
- Without shared storage, you need post-copy of disks or storage vMotion-like replication, increasing migration time and risk of write ordering issues; shared storage decouples memory movement from persistent data.
- Network continuity is preserved by maintaining the VM’s IP/MAC via L2 adjacency (VXLAN/GRE) or updating network virtualization overlays; health checks and live migration hooks bound the total downtime to milliseconds.

2. What is the benefit of a private cloud? Provide three reasons why Enterprises are adopting this even though they need to invest in the infrastructure. How does Hybrid cloud help?

- Private cloud delivers control over data residency, compliance (HIPAA, PCI-DSS), and security configurations (custom IAM, network segmentation, HSM-backed keys) that are hard to guarantee in shared multi-tenant environments.
- Predictable performance: dedicated hardware eliminates noisy neighbors; enterprises can right-size clusters for latency-sensitive workloads, enforce QoS, and reserve capacity for peak periods.

- Cost governance: for steady, predictable workloads, amortized capex + reserved capacity can be cheaper than perpetual public-cloud opex; chargeback/showback models improve accountability.
- Customization and integration: direct access to on-prem networks, legacy systems, and specialized hardware (mainframes, GPUs, FPGAs) allows bespoke stacks and tighter SLAs.
- Hybrid cloud complements private cloud by bursting elastic workloads to public cloud during spikes, offloading non-sensitive components, and leveraging managed services (e.g., CDN, AI/ML) without moving crown-jewel data.
- Data gravity and low-latency locality: keep datasets on-prem while processing edges or ephemeral compute in cloud; use direct connectivity (Direct Connect/ExpressRoute) and consistent IAM to unify control.
- Unified operations: use consistent IaC (Terraform/CloudFormation), container platforms (Kubernetes), and observability to deploy portable artifacts across private and public environments.

3. What is the key difference between Full and Para virtualization?

- Full virtualization emulates the entire hardware ISA so unmodified guest OSes run as if on bare metal; para-virtualization exposes hypercalls and enlightened drivers, requiring guest awareness to avoid expensive traps/emulation.
- In full virtualization, privileged instructions are trapped by the hypervisor (or assisted via VT-x/AMD-V) and emulated; in para-virtualization, guests call the hypervisor directly for privileged operations (MMU, timers, I/O).
- Performance: para-virtualization reduces VM-exits and context switch overhead, yielding lower latency and higher I/O throughput; full virtualization offers broader compatibility at some overhead.
- Driver model: PV drivers (virtio, Xen PV) optimize network and disk paths; full virtualization can also use PV drivers (PV-on-HVM) to mitigate emulation costs.
- Security/isolation: both rely on hardware ring compression and EPT/NPT; full virtualization's stronger abstraction reduces guest impact on the hypervisor API surface, while PV's smaller surface reduces emulation complexity.
- Operational trade-off: full virtualization eases lift-and-shift of legacy OS images; PV requires guest kernel changes or drivers but rewards with better determinism and density.
- Modern hypervisors often blend both (HVM with PV drivers) to balance compatibility and performance.

4. What is the benefit of serverless architecture? Why is it scalable?

- Serverless abstracts server provisioning/patching—developers ship functions and event handlers, while the platform handles capacity, OS/runtime management, and fault domains.
- It scales automatically via fine-grained concurrency: each request/event can trigger a new function instance up to account/service concurrency limits; no warm pool management is required by the team.
- Cost efficiency: pay-per-invocation and per-GB-second billing align cost with usage; idle capacity is not billed, ideal for bursty/erratic workloads.
- Built-in integration with event sources (API Gateway, SQS/SNS, streams, object storage events, schedulers) simplifies event-driven pipelines and microservices choreography.
- Operational resilience: isolation per invocation, retries with exponential backoff, DLQs, and regional multi-AZ execution improve availability without bespoke autoscaling logic.
- Developer velocity: small deployable units, CI/CD-friendly packaging, and environment variables/secrets managers accelerate iteration and safe rollouts.
- Scalability stems from elastic control planes that place micro-VMs/containers rapidly, horizontal fan-out per event, and managed quotas that protect the platform while allowing burst capacity.

5. What is the role of a message queue system like Kafka/Kinesis/SQS in the overall design of a system?

- Decoupling: producers and consumers operate independently in time and scale; queues absorb burst traffic and smooth workload to protect downstream services.
- Durability and ordering: replicated logs/partitions store events durably with at-least-once semantics; FIFO queues provide ordering and exactly-once processing constraints where needed.
- Backpressure and retry: visibility timeouts and retry policies prevent message loss while avoiding overload; DLQs isolate poison messages for inspection.
- Scalability: partitioning/sharding allows horizontal throughput growth; consumers can parallelize by partition key while preserving per-key order.
- Reliability and resilience: queues act as circuit breakers—when dependencies are slow/down, messages accumulate rather than dropping user requests.
- Integration: streams/queues feed analytics, ETL, search indexing, and ML asynchronously without coupling transaction paths to heavy processing.
- Observability and control: metrics (lag, age, throughput) and idempotency keys enable controlled rollouts, capacity tuning, and safe reprocessing.

6. What is a hybrid cloud? Explain through examples/use cases when it is beneficial over a solely public cloud and/or a private cloud. In other words, when and why one would benefit from hybrid cloud.

- Hybrid cloud combines on-prem/private cloud with public cloud under a single operating model to place workloads where they fit best based on compliance, performance, and cost.
- Data sovereignty: keep regulated PII or PHI on-prem while running stateless frontends and analytics jobs in public cloud with secure connectivity (VPN/Direct Connect) and unified IAM.
- Cloud bursting: baseline capacity runs on private cloud; peak demand bursts into public cloud using autoscaling groups or serverless to avoid overprovisioning hardware.
- Edge/latency-sensitive processing: run real-time systems (factory floor, trading) on-prem for microsecond latency; use cloud for batch analytics and archival storage.
- Modernization: gradually refactor monoliths—wrap on-prem systems with APIs, move new services to Kubernetes in cloud, and interconnect via service mesh.
- Cost optimization: steady, predictable workloads stay on reserved/on-prem capacity; spiky or experimental workloads leverage pay-as-you-go elasticity.
- Resilience: disaster recovery with warm/cold standby in cloud for on-prem systems, tested via periodic failover drills and RPO/RTO objectives.

7. Explain the difference between full and para virtualization. What are the pros and cons in each of these Why are all modern hypervisors are para virtualization?

- Full virtualization runs unmodified guests by trapping and emulating privileged instructions (assisted by VT-x/AMD-V); para-virtualization exposes hypercalls/enlightened drivers so guests cooperate with the hypervisor.
- Pros full: maximal compatibility, easy lift-and-shift of legacy OS images; cons: higher VM-exit/emulation overhead and potentially lower I/O throughput/latency.
- Pros para: fewer traps, better I/O via PV drivers (virtio), improved determinism and density; cons: requires guest kernel support/drivers and tighter coupling to hypervisor interfaces.
- Security: both rely on hardware isolation (EPT/NPT); PV reduces emulation surface, while full virtualization keeps stronger abstraction boundaries.
- Operations: PV can achieve better performance under contention; full virt simplifies heterogeneous guest management.

- Modern hypervisors are “PV-on-HVM”: hardware virtualization plus PV drivers—this hybrid yields compatibility of full virt with the performance of PV.
- Conclusion: not all are pure PV; rather, the industry standard is HVM with para-virtualized I/O stacks for optimal performance.

8. What is the role of a message queue like SQS in a cloud based system? Give an example to show with and without SQS what would be the comparative benefit with SQS.

- Role: decouple producers from consumers, buffer bursts, and provide durable, at-least-once delivery with backpressure via visibility timeouts and DLQs.
- Without SQS: a web API making synchronous calls to an email/SMS provider fails under spikes—timeouts propagate to users and requests get dropped.
- With SQS: the API enqueues jobs instantly and returns 202; workers drain the queue at a safe rate with retries/backoff; users receive notifications asynchronously.
- Reliability: DLQ captures poison messages; idempotency keys prevent duplicate side effects during retries; monitoring queue age/length enables autoscaling.
- Scalability: horizontal fan-out by increasing worker count; FIFO queues maintain per-message-group ordering for workflows requiring sequence.
- Cost and resilience: transient upstream/provider outages no longer take down the API; backlog is drained when providers recover.
- Example metrics: track ApproximateNumberOfMessages, AgeOfOldestMessage, and worker concurrency to tune throughput and SLOs.

9. We have discussed about Virtualization, Container and Serverless (Lambda in AWS) compute model. Illustrate through an example when you would use one over the other and their relative benefits and drawbacks.

- Virtual machines: choose when you need OS-level control, custom kernels/drivers, or long-running stateful services (e.g., databases needing tuned filesystems); pros: isolation, mature tooling; cons: heavier, slower provisioning.
- Containers: pick for microservices with fast startup, predictable dependencies, and high density (e.g., REST APIs, workers on EKS/ECS); pros: portability, quick scaling; cons: weaker isolation than VMs, host kernel coupling.
- Serverless (Lambda): ideal for event-driven, bursty workloads (e.g., image processing triggered by S3, API endpoints via API Gateway); pros: zero ops, automatic scaling, pay-per-use; cons: cold starts, execution time/memory limits, ephemeral disk.

- Example composite system: front-end calls API Gateway → Lambda for request validation and enqueue to SQS; containers on ECS consume and process; a VM-hosted database provides durable storage.
- Operational trade-offs: VMs for stateful data, containers for steady microservices, functions for spiky glue/edge logic; mix to optimize cost, latency, and control.
- Security: VMs offer strongest isolation; containers rely on namespaces/cgroups/AppArmor/SELinux; serverless isolates per-invocation with provider-managed sandboxes.
- Observability and CI/CD: containers integrate well with service meshes and blue/green; serverless emphasizes event traces; VMs require host-level agents and config management.

10. Why is iterative copy needed to facilitate Live migration with virtualization?

- Iterative (pre-copy) reduces service downtime by sending most memory while the VM continues running, leaving only a small remainder for the final stop-and-copy.
- It progressively shrinks the dirty set using dirty-bit tracking; each round copies only pages modified since the prior pass, improving convergence.
- Without iterations, pausing the VM to copy all memory would cause seconds of outage proportional to RAM size and bandwidth.
- Iterative copy balances network utilization and pause time; thresholds stop the loop when remaining dirty pages are below a target.
- Combined with CPU/device state transfer and ARP/ND updates, it enables near-seamless handoff on the destination host.
- If the write rate exceeds bandwidth (non-convergence), policies force a cutover, throttle the guest, or fall back to post-copy techniques.
- The approach integrates with shared storage so disk state isn't part of the critical path of memory migration.

11. Describe how you may implement an AWS Lambda like serverless capability using container.

- Use a control plane to watch for events (HTTP via API Gateway/Nginx, queues, schedulers) and dispatch to short-lived containers per invocation.
- Build a template container image with a minimal runtime (e.g., distroless + language runtime) and mount user code via layers or OCI artifacts for rapid cold starts.
- Implement an autoscaler that creates containers on demand with concurrency limits; reuse warm containers within a TTL to reduce cold starts.

- Enforce per-invocation isolation via one-container-per-request or micro-VMs (e.g., Firecracker) fronting containers for stronger security.
- Provide a function API (build/deploy) and logs/metrics/traces via sidecars/collectors; support env vars, secrets, and IAM roles for tasks.
- Add retries, exponential backoff, DLQs, and idempotency support; integrate with object storage and queues as event sources.
- Package everything with Kubernetes (Knative/OpenFaaS) or ECS/Fargate for managed scheduling, scale-to-zero, and routing.

12. Construct a cloud devops pipeline leveraging appropriate AWS services. Your design should assume that your code repo is in GitHub and target environment is deployment of set of micro services using AWS container service. Illustrate through appropriate diagram and illustration.

- Source: GitHub repo → GitHub Webhook triggers AWS CodePipeline/CodeBuild (or GitHub Actions) upon push to main/feature branches.
- CI: CodeBuild executes unit/static tests, builds multi-arch Docker images, runs vulnerability scans, and pushes images to Amazon ECR with immutable tags (commit SHA, semver).
- CD: CodePipeline promotes artifacts to environments; AWS CDK/CloudFormation/Terraform applies infra changes; images are deployed to ECS (Fargate) or EKS with blue/green or canary via CodeDeploy/Argo Rollouts.
- Config/secrets: AWS Systems Manager Parameter Store/Secrets Manager inject runtime config and credentials; per-environment overrides managed via tags/namespaces.
- Observability: CloudWatch/X-Ray/CloudTrail collect logs, metrics, traces; alarms trigger SNS/PagerDuty; SLOs tracked via dashboards.
- Security: IAM least privilege for CI/CD roles, ECR scanning, OPA/Kyverno admission policies; image signing with Sigstore/Notary and policy enforcement.
- Promotion strategy: dev → staging → prod with automated tests and manual approvals; rollbacks use previous task definitions/Kubernetes ReplicaSets.

13. Why is hybrid cloud beneficial for an enterprise compared to either public or private cloud?

- Optimal workload placement: sensitive, latency-critical, or steady-state systems run on private/on-prem; bursty, experimental, or ML/analytics workloads leverage elastic public cloud.

- Risk and compliance management: keep regulated data on-prem under enterprise controls (DLP, HSM) while consuming managed cloud services through private links and unified IAM.
- Cost control: reserve capacity on-prem for predictable baseload; use pay-as-you-go for spikes, avoiding overprovisioning and stranded capex.
- Migration path: lift-and-shift to private cloud first, then refactor services incrementally to cloud-native components, reducing big-bang risk.
- Business continuity: cross-site DR with cloud as warm/cold standby; periodic failover validates RPO/RTO without building a second datacenter.
- Developer productivity: consistent toolchains (Kubernetes, IaC, CI/CD) across environments enable portability and faster delivery.
- Vendor flexibility: avoid lock-in by abstracting with containers/service mesh and adopting multi-cloud-ready patterns where appropriate.

14. What is paravirtualization and why is paravirtualization more efficient than full virtualization?

- Paravirtualization (PV) is a virtualization technique where the guest OS is aware of the hypervisor and uses hypercalls/enlightened drivers instead of executing privileged instructions that require trapping/emulation.
- Efficiency arises from fewer VM-exits and reduced instruction emulation; PV replaces expensive traps for MMU updates, timers, and I/O with direct calls.
- PV drivers (e.g., virtio-net, virtio-blk) provide fast paravirtualized device interfaces, avoiding full device emulation and delivering higher throughput/lower latency.
- CPU/memory: PV can batch TLB/MMU operations and leverage shared rings for I/O, decreasing context switches and cache pollution.
- Hybrid approach: modern HVM guests still use PV I/O (PV-on-HVM) to get near-PV performance while retaining compatibility.
- Caveat: PV requires guest support and can expose a broader hypercall surface; security hardening and stable ABIs mitigate risks.
- Net: PV improves performance and density, especially for I/O-intensive workloads, over pure full virtualization.

15. Explain how iterative memory copy is leveraged for live migration?

- The hypervisor initiates a pre-copy phase that transfers all guest memory to the destination while tracking dirty pages using write-protection/dirty-bit logging.
- Subsequent iterations send only dirty pages, shrinking the remaining set; copy bandwidth and guest write rate determine convergence.

- When the dirty set falls below a threshold, the VM is briefly paused for stop-and-copy of CPU/device state and the last dirty pages, then resumed on the target.
- ARP/ND updates or network overlays ensure traffic flows to the new host; the VM's IP/MAC remain stable to minimize connection disruption.
- If write intensity prevents convergence, policies cap iterations and force cutover, throttle the VM, or switch to post-copy with demand paging and page fault forwarding.
- Shared storage or storage migration ensures disk consistency independent of RAM transfer, keeping downtime tied to memory, not disk size.
- Health checks and bounded pause budgets maintain SLAs, typically yielding millisecond-level downtime.

16. What are the key differences between VM and containers?

- Isolation layer: VMs virtualize hardware and run separate kernels; containers share the host kernel using namespaces/cgroups for isolation.
- Footprint and startup: VMs are heavier (GB images) and boot in seconds/minutes; containers are lightweight (MB images) and start in milliseconds.
- Portability and density: containers achieve higher density per host and are well-suited to microservices; VMs are better for heterogeneous OSes and strong isolation.
- Security: VMs offer stronger isolation boundaries; containers rely on kernel hardening (seccomp, AppArmor/SELinux) and require additional controls for multi-tenant security.
- Lifecycle and orchestration: containers integrate with Kubernetes for declarative scaling/rollouts; VMs use hypervisor tooling and autoscaling groups.
- State management: stateful apps often prefer VMs or specialized Kubernetes patterns (StatefulSets, PVCs) to handle storage and identity.
- Use cases: VMs for databases/legacy stacks/custom kernels; containers for stateless APIs, workers, and CI/CD pipelines.

17. How does the trade-off between memory space and false positives influence the design of Bloom filters in large databases, and how might this impact overall database performance?

- Bloom filters use bit arrays and multiple hash functions to test set membership with tunable false positive rate (FPR); no false negatives, possible false positives.
- For a given number of elements n, the bit-array size m and number of hashes k determine FPR; increasing m lowers FPR at the cost of memory.

- Databases place Bloom filters in memory (e.g., SSTable-level in LSM trees) to avoid unnecessary disk I/O; lower FPR reduces disk seeks/scans.
- Diminishing returns: beyond a certain m, further FPR reductions yield little latency gain versus extra RAM cost; designers pick m to balance cache hit rates and memory availability.
- Workload-aware tuning: higher FPR may be acceptable for hot data in cache but harmful for cold data requiring disk; tier-specific filters reduce tail latency.
- CPU overhead: more hashes (k) increase CPU cost; choose $k \approx (m/n) \ln 2$ for optimal FPR given m/n.
- Impact: properly sized Bloom filters cut read amplification, improving throughput and P99 latency; oversized filters waste memory that could serve as cache.

18. Explain the key difference between Full Virtualization, and Para Virtualization and state the pros-cons of each.

- Key difference: full virtualization runs unmodified guests via hardware trapping/emulation; para virtualization requires guest awareness and uses hypercalls for privileged operations.
- Full virtualization pros: broad OS compatibility, simple lift-and-shift; cons: higher overhead on privileged paths and emulated devices.
- Para virtualization pros: lower overhead, faster I/O with PV drivers, better performance density; cons: needs guest kernel support/drivers and increases coupling.
- Security: full virtualization offers stronger abstraction; PV reduces emulation surface but expands hypercall interface—both rely on hardware isolation.
- Operations: full virt simplifies heterogeneous estates; PV suits high-throughput, latency-sensitive workloads.
- Modern practice: combine HVM with PV I/O (virtio) to achieve a balanced trade-off.
- Selection: choose based on workload profile, required OS compatibility, and performance targets.

19. Imagine you are developing a containerized application where user-uploaded files need to be persistently stored across container restarts. How would you use Docker volumes to achieve this, and provide a small Docker Compose template illustrating this.

- Persist data by mounting a Docker volume or bind mount to the container's data directory so files live outside the container's writable layer and survive

restarts/redeploys.
- Prefer named volumes for portability and easier lifecycle management; use bind mounts for development when you need to live-sync a host directory.
- In Compose, declare a `volumes:` section at both service and top-level; map the volume to the container path where your app writes uploads (e.g., `/app/uploads`).
- Ensure correct permissions/UIDs inside the container; use `chown` at build time or run the process as a user that owns the mount to avoid permission errors.
- For multi-container scenarios, attach the same named volume to all services that need read/write access; consider access modes (ro/rw) accordingly.
- Backups and migration: snapshot named volumes or sync the underlying volume path; in production on Kubernetes, use PVCs backed by durable storage.
- Minimal Compose example:

```
version: "3.9"
services:
  web:
    image: myapp:latest
    ports:
      - "8080:8080"
    volumes:
      - uploads:/app/uploads
volumes:
  uploads:
```

20. You built a docker image where the dockerfile looks like follows: Docker image 1 —--------------------------------- 1) COPY./Assignment ./src 2) RUNcd./src 3) RUN"some build operation" ——---> point 1 Docker image 2 —--------------------------------- 1) COPY./Assignment ./src 2) RUNcd./src 3) RUN"some build operation" 4) RUN"rm-rf ./src" 5) RUN"ls" —---------> point 2 What do you think about the image size of docker image 1 and docker image 2, which one will be greater ? Explain your answer

- Image 2 will be larger or equal despite removing `./src` in a later layer, because each `RUN/COPY` creates an immutable layer; deleting files in a new layer does not reclaim bytes from prior layers.
- Layering model: Docker images are stacked layers; the final filesystem is the union, but underlying layers remain, contributing to size and pull time.
- Best practice: minimize layers and avoid copying large build contexts; use `.dockerignore` to exclude artifacts.

- Use multi-stage builds: compile in a builder stage, then copy only runtime artifacts into a slim final stage.
- Combine related commands into fewer `RUN` steps and clean up in the same layer to actually reduce size.
- Choose minimal base images (alpine/distroless) where feasible; pin versions for reproducibility and caching.
- Verify size with `docker history` and `docker images`; observe unchanged large layers across builds.

21. Explain Master-Slave Architecture as discussed in Lecture 1 and its issues concerning database replication

- Master-slave (primary-replica) replication has one writable primary applying changes and streaming logs/checkpoints to read-only replicas.
- Replication modes: asynchronous (low write latency, risk of data loss on failover), synchronous (stronger consistency, higher write latency), semi-sync (compromise).
- Issues: replication lag leads to stale reads; write amplification and network partitions can cause divergent states and complex failover.
- Failover complexity: electing a new master requires consensus/quorum (e.g., Raft/Paxos) or external tooling; split-brain can occur without fencing.
- Consistency semantics: read-your-writes and monotonic reads are not guaranteed on async replicas; apps must route critical reads to primary or use session stickiness.
- Operational concerns: schema changes, large transactions, and hot partitions stress replication; backups and point-in-time recovery must align with replication logs.
- Modern alternatives: multi-leader or leaderless (CRDTs, Dynamo-style) trade stronger availability for consistency; or use managed services with automated failover.

22. Describe the live migration process and how it enables the seamless movement of virtual machines between physical hosts without service interruption and how it differs from traditional offline migration methods.

- Live migration copies VM memory iteratively while the VM runs, then briefly pauses to transfer CPU/device state and the final dirty pages before resuming on the destination, yielding near-zero downtime.

- It leverages dirty-page tracking, convergence thresholds, and L2/L3 network continuity (ARP/ND updates or overlays) to keep connections intact.
- Shared storage or synchronized storage migration ensures disk consistency, decoupling persistent data from the memory transfer path.
- Offline migration stops the VM first, then copies the entire memory and disk, causing downtime proportional to data size and bandwidth.
- Live migration includes health checks and pre/post hooks to bound pause time; offline migration offers simplicity but disrupts service.
- Policies handle non-convergence by throttling the guest, forcing a cutover, or using post-copy techniques with demand paging.
- Operationally, live migration supports maintenance, load balancing, and power optimization without impacting SLAs.

23. Explain how Kubernetes achieves high availability and fault tolerance in a cluster. What components and mechanisms are used to ensure that applications running in Kubernetes remain available in the event of node failures?

- Control plane HA: multiple API servers behind a load balancer, redundant controller managers/schedulers, and etcd clusters (odd size, quorum) across AZs.
- Node health and reconciliation: kubelet reports status; controllers reschedule pods from failed nodes to healthy nodes automatically based on desired state.
- ReplicaSets/Deployments/StatefulSets maintain pod replicas; PodDisruptionBudgets and anti-affinity spread replicas across nodes/AZs.
- Probes and load balancing: readiness/liveness/startup probes gate traffic and restart unhealthy containers; Services + kube-proxy/IPVS/ingress route around failed pods.
- Persistent storage: CSI drivers with replicated backends and PVCs reattach volumes to new nodes; StatefulSets retain stable identities.
- Upgrades and rollouts: rolling updates/canary with maxSurge/maxUnavailable reduce downtime; workload autoscalers adjust capacity.
- Disaster scenarios: cluster autoscaler replaces failed nodes; backup/restore for etcd and app data plus multi-region strategies enhance resilience.

24. How does hybrid cloud work and what are its challenges?

- Operation: connect on-prem/private cloud and public cloud via VPN/Direct Connect, unify identity (SSO/IAM), and deploy portable workloads (containers/Kubernetes) with shared CI/CD and observability.

- Data placement: keep sensitive/large datasets on-prem for compliance/data gravity; run stateless and burst workloads in public cloud for elasticity.
- Networking: extend L3/L7 routing, service discovery, and zero-trust policies across environments; use private endpoints and routing domains to reduce exposure.
- Governance: consistent policies for secrets, audit, and cost controls across clouds; implement guardrails, budgets, and tagging.
- Challenges: network complexity/latency, inconsistent IAM/policies, data synchronization/replication, and different service semantics/quotas.
- Skills and tooling: teams must master diverse platforms; invest in platform engineering, IaC, and SRE practices.
- Vendor lock-in vs. portability: abstractions (service mesh, Kubernetes, open APIs) mitigate but not eliminate friction and feature gaps.

25. Explain etcd, Persistent Volume, Persistent Volume Claim, and statefulsets in Kubernetes.

- etcd: a distributed, strongly consistent key-value store that holds Kubernetes cluster state; API server persists objects in etcd and controllers reconcile from it.
- Persistent Volume (PV): a cluster resource representing provisioned storage (static or dynamic via CSI) decoupled from pod lifecycle; has capacity, access modes, and reclaim policy.
- Persistent Volume Claim (PVC): a namespaced request for storage by a pod; the control plane binds a PVC to a matching PV (or triggers dynamic provisioning) based on size/modes/storageClass.
- StorageClass: defines provisioner and parameters (IOPS, filesystem, zones) for dynamic PV creation; enables tiered storage and automation.
- StatefulSets: workload controller for stateful apps providing stable network identities, ordered deployment/termination, and volume claims per replica via `volumeClaimTemplates`.
- Data durability: PVs persist beyond pod restarts; StatefulSets reattach the same volume to a replacement pod, preserving data and identity.
- Operations: backups/snapshots via CSI, resizing PVCs, and managing reclaim policies (Retain/Delete/Recycle) are critical for safe state management.

26. Explain the differences between the three primary cloud service models—Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) and provide 3 points for each and two example products for each

- IaaS: provides virtualized compute, storage, and networking primitives; users manage OS/middleware. Pros: flexibility/control; Use cases: custom stacks, lift-and-shift. Examples: AWS EC2/EBS/VPC, Azure VMs/Disks/VNet.
- PaaS: provides managed runtimes/databases/build services; users deliver code without managing OS or many middleware components. Pros: higher velocity, autoscaling, built-in CI/CD. Examples: AWS Elastic Beanstalk/Heroku, Azure App Service.
- SaaS: delivers complete applications over the internet; provider manages everything. Pros: fastest time-to-value, no infra ops. Examples: Salesforce, Google Workspace.
- Responsibility model: IaaS—customer handles OS/patching; PaaS—provider handles runtime/patching; SaaS—provider handles application/data plane within SLAs.
- Cost/lock-in: IaaS widest portability, PaaS moderate lock-in via APIs, SaaS highest lock-in but lowest ops overhead.
- Scaling: IaaS via autoscaling groups; PaaS via platform scaling knobs; SaaS scales internally and is opaque to users.
- Security/compliance: shared responsibility shifts toward provider from IaaS→SaaS; governance must adapt accordingly.

27. Your company is deciding between serverless (AWS Lambda) and containerized (ECS/EKS) architectures for a high-traffic web application.Explain: ? The key differences in scalability, cold start latency between serverless functions and containers. ? A scenario where serverless would be the better choice, and another where containers would be preferable.

- Scalability: Lambda scales per-invocation up to concurrency limits with event-driven fan-out; ECS/EKS scale by tasks/pods and nodes, giving sustained throughput and stronger control.
- Cold starts: Lambda has cold start latency (tens to hundreds of ms, higher with VPC/large packages); containers keep warm processes and typically exhibit lower tail latency under steady load.
- Serverless best-fit: bursty, unpredictable traffic with sporadic usage (e.g., processing uploads, webhook handlers), where pay-per-use and zero idle cost are valuable.
- Containers best-fit: consistent high-throughput APIs requiring low steady-state latency, long-lived connections (WebSockets/gRPC), or custom runtimes/system packages.

- Operational control: containers allow fine-grained resource limits, sidecars, service mesh, and advanced networking; serverless minimizes ops but with platform constraints.
- Cost: serverless economical at low average utilization; containers/EC2 more cost-efficient at high steady utilization.
- Hybrid pattern: API Gateway + Lambda for edge validation/enqueue, ECS/EKS for heavy processing and persistent services.

28. In a distributed system with unpredictable traffic, how does a message queue system like SQS improve resilience and scalability? Explain: ? How message queues prevent system overload and failures. ? How they enable scalability across microservices. ? A real-world use case demonstrating these benefits.

- Overload prevention: producers enqueue requests quickly and return; queues absorb bursts and decouple latency from downstream throughput, preventing cascading failures; visibility timeouts and DLQs manage retries safely.
- Scalability: consumers scale horizontally based on queue depth/age; partition keys (message groups) preserve ordering while enabling parallelism across groups.
- Flow control: rate limits and concurrency caps per consumer protect external dependencies; backoff and idempotency keys avoid duplicate side effects.
- Resilience: temporary outages of downstream services accumulate backlog rather than dropping requests; once healthy, workers drain the queue.
- Observability: CloudWatch metrics and tracing on enqueue/dequeue enable autoscaling policies and SLO enforcement; DLQ redrive supports remediation.
- Real-world: e-commerce checkout posts payment/fulfillment jobs to SQS; surges (flash sales) are buffered, workers process as capacity allows, customers receive confirmations asynchronously.
- Security and reliability: SSE encrypts messages at rest, IAM policies restrict access, and multi-AZ durability preserves data through infrastructure failures.

29. What is the use of Secondary Indexes in DynamoDB tables? List and explain the different types of secondary indexes available in AWS DynamoDB.

- Purpose: secondary indexes provide alternate query paths without scanning the base table, enabling efficient access patterns beyond the primary key.
- Types: Global Secondary Index (GSI) with its own partition/sort keys independent of the table PK/SK; Local Secondary Index (LSI) shares the same partition key as the table but defines an alternate sort key.

- GSIs: scale independently with separate RCUs/WCUs; good for many-to-one lookups and different partitioning strategies; eventual consistency by default.
- LSIs: enable multiple sort orders within the same partition (e.g., by timestamp and by status) and support strongly consistent reads; limit of 5 per table, must be defined at table creation.
- Projections: KEYS_ONLY, INCLUDE, or ALL_ATTRIBUTES control which attributes are copied; tailor to minimize storage and read cost.
- Maintenance: writes to the base table propagate to indexes; design for write amplification and hot partitions by choosing high-cardinality partition keys.
- Patterns: multi-tenant PK with composite SK, inverted GSIs for reverse lookups, and sparse indexes to include only items with specific attributes.