AWS INTERVIEW QUESTIONS

1. **EC2, Auto Scaling and Load Balancing**
   1. Auto Scaling Policy in AWS?

An **Auto Scaling policy** defines **when and how** your Auto Scaling Group (ASG) should launch or terminate EC2 instances.

* They ensure the right number of instances run **based on demand**.
* Policies are **triggered by metrics** (like CPU, memory, custom CloudWatch metrics).

🔹 **Types of Auto Scaling Policies**

**1. Target Tracking Scaling Policy (Most Common ✅)**

* Works like a **thermostat**.
* You set a target value for a metric, and ASG automatically scales instances to maintain it.
* Example:
  + Target metric: Average CPU Utilization = **50%**
  + If CPU goes above 50% → Scale out (add instances)
  + If CPU goes below 50% → Scale in (remove instances)

👉 Best for **stable workloads** where you want to maintain a performance baseline.

**2. Step Scaling Policy**

* Scaling is based on **thresholds (steps)** in CloudWatch alarms.
* Example:
  + CPU > 70% → Add 2 instances
  + CPU > 90% → Add 4 instances
  + CPU < 40% → Remove 1 instance

👉 Useful for **spiky traffic** or when you want **granular control**.

**3. Simple Scaling Policy (Legacy, replaced by Step Scaling)**

* Triggered by a single CloudWatch alarm.
* Example: If CPU > 70%, add 1 instance.
* **Downside:** Waits for cooldown before acting again (less flexible).

👉 Not recommended anymore, Step Scaling is preferred.

**5. Predictive Scaling Policy (Advanced 🚀)**

* Uses **Machine Learning** to forecast demand based on historical data.
* Automatically adjusts capacity **before** demand spikes.
* Example: If traffic increases every Monday morning, predictive scaling adds instances in advance.

👉 Best for **repeatable traffic patterns** (e.g., e-commerce sales, financial market opens).

* 1. What is Instance Warm Up time?
* **Definition**: The time it takes for a newly launched EC2 instance to become fully **ready** (booted, app started, registered in the load balancer, healthy).
* **Why needed**: Without warm-up time, Auto Scaling might think the new instance is already contributing to load, and unnecessarily launch even more instances.

👉 Example:

* Your ASG scaling policy says:  
  *“Add 1 instance if CPU > 70%”*
* CPU spikes → ASG launches a new EC2 instance.
* The new instance takes **5 minutes** to install software, start app, and pass ELB health check.
* If no warm-up time is set, CloudWatch still sees **high CPU** on the old instances and might keep adding more instances (over-scaling).
* If you set **300 seconds warm-up**, ASG ignores that instance’s metrics until it’s ready.
  1. What is Scaling Cooldown Period ?
* **Definition**: A **pause period** after a scaling activity (scale-out or scale-in) before another scaling event can occur.
* **Why needed**: Prevents “flapping” (rapid scaling in and out).

👉 Example:

* Your ASG scaling policy:  
  *“Add 1 instance if CPU > 70%, remove 1 instance if CPU < 30%.”*
* Traffic spikes at 1 PM → ASG adds 2 instances.
* Immediately after traffic drops slightly, CPU goes below 30% → ASG might try to remove instances too soon.
* With a **cooldown period of 300 seconds**, ASG waits 5 minutes before checking again, giving the system time to stabilize.
  1. Real World Example – Putting WarmUp and cooldown time ?

Imagine an e-commerce website during a **flash sale**:

1. Traffic spikes → CPU reaches 80%.
2. ASG policy triggers → Adds 2 new instances.
3. Each instance takes **4 minutes warm-up** (boot + app start).
4. During warm-up, their metrics are ignored so ASG doesn’t keep adding more.
5. After scale-out completes, ASG enters **cooldown for 5 minutes** → no new scale-in/out happens, avoiding rapid oscillations.
6. Once cooldown ends, ASG evaluates again.

✅ **Summary**:

* **Warm-up time** → for *new* instances to get ready before being counted in metrics.
* **Cooldown period** → after *any* scaling action, ASG waits before scaling again.
  1. What is the difference between EC2 and ELB health checks?

**1. EC2 Health Checks (Instance-Level Health Check)**

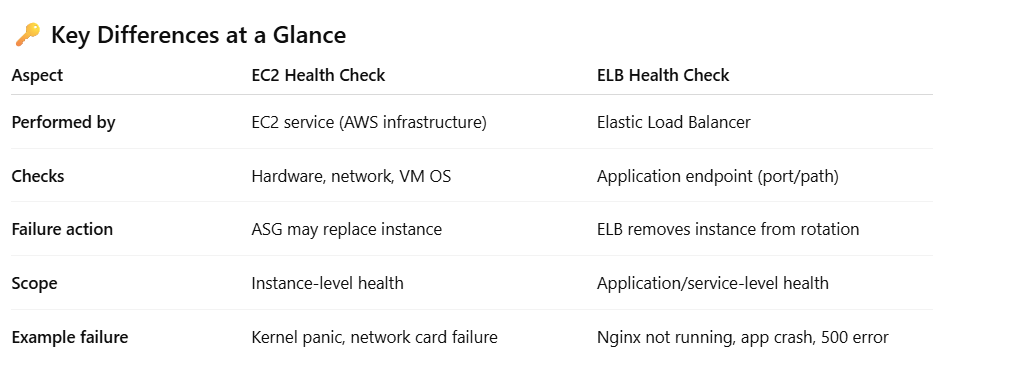
* **Who performs it?** → **Amazon EC2 service** itself.
* **What is checked?**
  + **System Status Check** → verifies AWS infrastructure (e.g., host hardware, networking, hypervisor).
  + **Instance Status Check** → verifies the VM itself (e.g., OS boot errors, kernel panics, unreachable due to misconfigurations).
* **Action on failure:**
  + If an instance fails its **EC2 health checks**, **Auto Scaling Group (ASG)** can terminate and replace it.
  + Example: Instance OS crashed, network card issue, or host hardware failure → EC2 marks it unhealthy.

📌 **Scope**: Purely about the instance itself being up and running.  
📌 **Frequency**: Every 1 minute (by default).

**2. ELB Health Checks (Application-Level Health Check)**

* **Who performs it?** → **Elastic Load Balancer (ELB)**.
* **What is checked?**
  + ELB sends a request (HTTP/HTTPS/TCP) to a **configured endpoint** (e.g., /health on port 80).
  + It expects a **200 OK** (or TCP success) as a healthy response.
* **Action on failure:**
  + If an instance fails its **ELB health check**, the ELB **stops sending traffic** to that instance, but EC2 still considers it "running."
  + ASG can also use ELB health checks to decide when to replace an instance.

📌 **Scope**: Checks whether the application/service **inside the instance** is responding.  
📌 **Frequency**: Configurable (e.g., check every 30 seconds, require 2 consecutive successes to mark healthy).



* 1. What are Lifecycle Hooks?
* Lifecycle hooks in an **Auto Scaling Group (ASG)** let you **pause** an instance when it is being launched or terminated, so you can perform **custom actions** before the instance is put into service or removed.
* Think of them as **"interception points"** during the scaling workflow.

**When do they apply?**

There are **two main hook points**:

1. **Instance Launching** (autoscaling:EC2\_INSTANCE\_LAUNCHING)
   * Instance comes up, but before it’s added to the ELB/target group and starts serving traffic.
   * Example use: Run configuration scripts, install agents, fetch secrets, patch the OS.
2. **Instance Terminating** (autoscaling:EC2\_INSTANCE\_TERMINATING)
   * Instance is selected for termination, but before it’s actually shut down.
   * Example use: Drain connections, push logs to S3, de-register from monitoring, clean up state.

**How do they work?**

* When the lifecycle hook is triggered, the instance enters a **wait state** (default up to 60 mins, configurable).
* You can extend the wait state using the **CompleteLifecycleAction** API.
* While in this state, you can use **SNS or SQS** (or EventBridge) to notify a worker (like a Lambda or custom script) that an action is required.
* Once the custom action is done, you signal ASG:
  + CONTINUE → ASG proceeds with launch/terminate.
  + ABANDON → ASG skips the normal flow (e.g., don’t add the instance to the group).
  1. Scaling Not Happening Despite High CPU. Your ASG has a target tracking policy set to keep CPU at 50%. CloudWatch shows CPU at 90%, but no new instances are being launched. What could be wrong
* **Max capacity** reached (check ASG min/max/desired values).
* **Cooldown period** or **warm-up time** blocking further scaling.
* **CloudWatch alarm misconfigured** (wrong metric/namespace).
* **EC2 credit limits** (e.g., T2/T3 burstable instances running out of credits).
* **IAM role** doesn’t allow scaling actions.
  1. Thrashing / Ping-Pong Scaling. Your ASG is constantly adding and removing instances, causing instability. How do you fix it?
* Increase **cooldown period** or **warm-up time**.
* Use **Target Tracking** instead of Step Scaling (smoother).
* Add **buffer capacity** (min capacity > 1).
* Use **smoothing metrics** (e.g., average CPU over 5 mins instead of 1 min).
  1. Predictive Scaling vs Scheduled Scaling . When would you use **Predictive Scaling** instead of **Scheduled Scaling**?
* **Scheduled Scaling** → predictable load (e.g., office hours, batch jobs).
* **Predictive Scaling** → recurring but not exact load patterns (e.g., holiday shopping peaks, weekly sales).
* Predictive scaling uses **machine learning** to anticipate demand and scale ahead of time.
  1. How would you design an Auto Scaling Group to **reduce costs** while keeping high availability?
* Use **Mixed Instance Policies**:
  + 70% Spot Instances
  + 30% On-Demand Instances
* Spread across **multiple AZs**.
* Enable **Capacity Rebalancing** so Spot interruptions trigger replacement.

**What is Capacity Rebalancing?**

Capacity Rebalancing is an **ASG feature that proactively replaces at-risk Spot Instances** before AWS actually interrupts them.

* Normally, Spot Instances can be **terminated with only a 2-minute warning** when AWS needs capacity back.
* With **Capacity Rebalancing**, the ASG tries to **launch a replacement Spot Instance early** (as soon as AWS sends an “interruption notice” signal), instead of waiting until the instance is killed.
* This helps maintain **application availability** and reduces the risk of sudden capacity drops.
  1. **What are sticky sessions and how it works?**

**What are Sticky Sessions?**

Sticky sessions (also called **session affinity**) mean that **a client is always routed to the same backend server** (EC2 instance, container, etc.) during the lifetime of their session.

👉 Without sticky sessions: requests from the same client could go to different servers.  
👉 With sticky sessions: requests from a client are “stuck” to one specific server.

**Why are they needed?**

Some applications store session data **locally on the instance** (not in Redis/DynamoDB/Memcached).

* Example: User logs in → session info is stored in /tmp or instance memory.
* If the next request goes to a different instance, the session is **lost**, and the user may need to re-login.

Sticky sessions solve this by ensuring the client always hits the **same server** that holds their session data.

How Sticky Sessions Work in AWS ELB

**1. Application Load Balancer (ALB)**

* Supports **stickiness via target group-level cookies**.
* ALB generates a cookie (AWSALB or AWSALBTG) when stickiness is enabled.
* Duration can be configured (1 sec – 7 days).

**2. Network Load Balancer (NLB)**

* Doesn’t support cookies (Layer 4).
* But you can achieve “stickiness” by enabling **source IP affinity** (the same client IP always goes to the same target).
  1. **EC2 Cost Optimization 🡪 Your company runs 200 EC2 instances for a batch processing workload that only runs 8 hours a day. Currently, all are On-Demand. How would you optimize cost?**

Expected Answer:

* Use Spot Instances for non-critical, fault-tolerant jobs.
* Use Scheduled Reserved Instances or Savings Plans for predictable workloads.
* Implement Auto Scaling to shut down EC2s when not in use.
* Optionally use AWS Batch for managed scheduling and scaling.
  1. **EC2 Storage Decisions 🡪 Your EC2 is running a database. You need high IOPS with persistence across reboots. Which storage type would you choose and why?**

We are dealing with three main requirements:

1. **High IOPS** (Input/Output Operations per Second) → fast read/write performance.
2. **Persistence across reboots** → data must survive instance stop/start or reboot.
3. **Database use case** → reliability and durability are critical.

**1. Amazon EBS (Elastic Block Store)**

Amazon EBS is a **block-level storage** that behaves like a virtual hard drive attached to EC2.

**Recommended Types for High IOPS**

* **io2 / io2 Block Express volumes**
  + **Designed for mission-critical workloads** like databases.
  + **High IOPS and throughput:** Up to 256,000 IOPS per volume with Block Express.
  + **Durable:** 99.999% durability, persistent across instance stops/reboots.
  + **Low latency:** Optimized for transactional workloads like MySQL, PostgreSQL, Oracle, or SQL Server.
* **Why not gp3?**
  + gp3 is cheaper and general-purpose SSD. Can do moderate IOPS (~16,000), but for high-performance databases, io2 is more suitable.

**Extra Tip:** For very high I/O workloads, you can use **RAID 0 (striping)** across multiple io2 volumes to increase throughput.

**2. Instance Store (Ephemeral Storage)**

* **Characteristics:** Directly attached SSD/NVMe storage.
* **Pros:** Very high IOPS, ultra-low latency.
* **Cons:** **Non-persistent** → data is lost if the instance stops, terminates, or fails.

**Use Case:** Only suitable for **temporary caches** or scratch space (e.g., Redis cache, buffer files).

* **Not suitable for databases** unless data is replicated elsewhere.

**3. Amazon EFS / FSx (For Distributed or Shared Databases)**

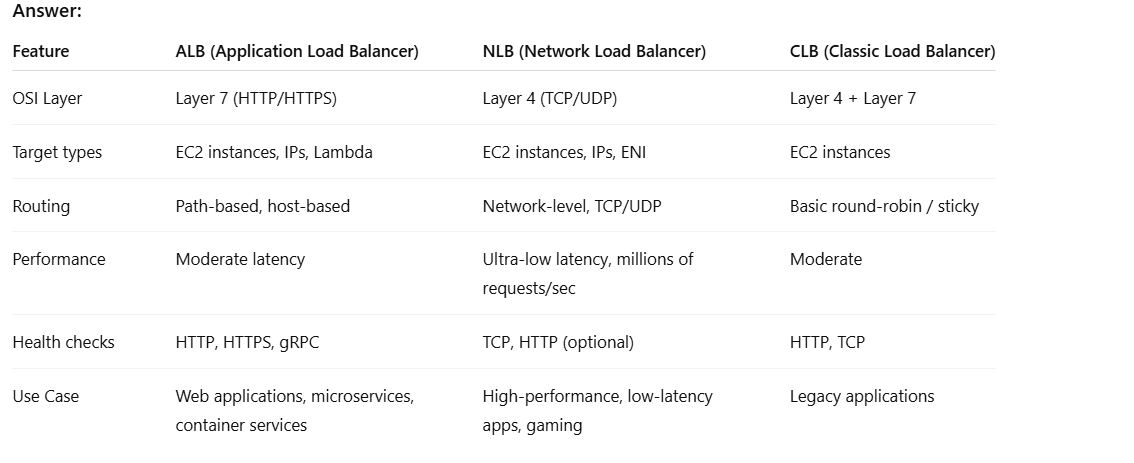
* **Amazon EFS (Elastic File System)**
  + Managed NFS file system, **shared across multiple instances**.
  + Can scale **automatically**.
  + Throughput scales with storage size.
* **Amazon FSx**
  + Managed Windows or Lustre file systems.
  + Can provide high throughput for **parallel/distributed workloads**.

**Use Case:**

* When your database is **clustered or distributed** (e.g., MySQL Galera cluster, Cassandra, or PostgreSQL with multiple read replicas).
* These file systems provide **shared storage** for multiple EC2 instances.
  1. EC2 Disaster Recovery 🡪 Your EC2 in us-east-1 is running a business-critical workload. How do you design for Disaster Recovery (DR)?

**Expected Answer:**

* **Cross-region AMI copies** for backups.
* Use **EBS snapshots replicated across regions**.
* Implement **pilot-light or warm standby DR strategy** in another region.
* Use **Route 53 failover routing** for DNS-based region failover.
  1. What are the differences between ALB, NLB, and CLB? When would you choose each?



**Scenario Example:**

* Microservices with dynamic routing → **ALB**
* High-throughput financial applications → **NLB**
  1. What is Cross-Zone Load balancing and how it works?

Cross-Zone Load Balancing (CZLB) in AWS is a feature of Elastic Load Balancers (ELB), particularly **Application Load Balancers (ALB)** and **Classic Load Balancers (CLB)**, that helps distribute traffic evenly across instances in multiple Availability Zones (AZs). Let me break it down step by step.

**How it Works?**

**Without Cross-Zone Load Balancing**

* Assume you have 2 AZs: AZ-A (2 instances), AZ-B (4 instances).
* A request is routed 50/50 across AZs:
  + AZ-A: 50% traffic → 2 instances → each gets 25%
  + AZ-B: 50% traffic → 4 instances → each gets 12.5%
* Result: Instances in AZ-A are overloaded, instances in AZ-B are underutilized.

**With Cross-Zone Load Balancing**

* Traffic is distributed evenly across **all instances**:
  + 6 instances in total → each instance gets ~16.67% of traffic
* No AZ is favored; every instance shares the load equally
* Impact: Without cross-zone, uneven load may cause hot spots. With cross-zone enabled, traffic might cross AZ boundaries → extra inter-AZ data transfer costs (~$0.01/GB).
  1. What is Weighted Routing in ALB?

Weighted routing lets you **distribute incoming traffic across multiple target groups** with a specified ratio (weights).  
It’s often used for **blue/green deployments**, **canary testing**, or **gradual traffic shifting**.

Example:

* Target group A (Blue) → 70% traffic
* Target group B (Green) → 30% traffic

**🔹 How ALB Implements Weighted Routing**

ALB uses **listener rules** and **forward actions** to achieve weighted routing:

1. **Listener Rule**:
   * Defines how ALB should process requests (based on conditions like path, host, headers, etc.).
   * For weighted routing, you use the **default action** or a rule with a **forward action**.
2. **Forward Action with Multiple Target Groups**:
   * You specify **two or more target groups**.
   * Assign a **weight (1–999)** to each target group.
   * ALB distributes requests **proportionally** to the weights.

**🔹 Example Scenario**

Suppose you’re migrating from v1 of your service (blue) to v2 (green).

* Listener rule:
  + If request → /app, then **forward** to:
    - **Target Group A (blue)**, weight = 80
    - **Target Group B (green)**, weight = 20

Result:

* 80% of traffic goes to Blue
* 20% goes to Green
* All targets still benefit from **cross-zone load balancing** inside their groups.

**🔹 Real Use Cases**

1. **Canary Deployment** → Send 5% traffic to new version, 95% to stable version.
2. **Blue/Green Deployment** → Gradually shift 100% of traffic from Blue to Green.
3. **Cost Optimization / Migration** → Shift traffic between different instance types or container versions.

✅ **Key Point:**  
ALB **does not support DNS-level weighted routing** (like Route 53). Instead, it does **application-level weighted routing** between target groups. Route 53 weighted routing works at the DNS request level, whereas ALB’s weighted routing works at the **HTTP request level**.

* 1. **How would you design Highly available, scalable and cost optimized web application in AWS ?**

**✨ High-level goals**

* **High availability: survive AZ failures and most single-component failures**
* **Scalable: handle traffic spikes automatically and gracefully**
* **Cost-optimized: pay for what you need, use cheaper capacity where safe**

**🔧 Core architecture (multi-AZ, single region)**

1. **Edge / CDN**
   * **Amazon CloudFront in front of everything for low latency, caching, WAF integration, and DDoS protection.**
   * **Serve static assets (JS/CSS/images) from S3 (website hosting or origin) with proper cache headers.**
2. **Ingress / Load Balancing**
   * **Application Load Balancer (ALB) in multiple AZs for HTTP/HTTPS traffic.**
   * **Use ALB target groups for microservices or autoscaled instances.**
   * **Offload TLS at ALB (use ACM certificates).**
3. **Compute layer (two main patterns — choose one or mix)**
   * **Serverless-first: Use AWS Lambda + API Gateway / ALB (Lambda targets) for maximal cost-efficiency at low/variable load.**
   * **Container-first: ECS (Fargate) for minimal ops or EKS if Kubernetes is required.**
   * **VM-first: EC2 Auto Scaling Groups (ASG) with multiple instance types (mixed policy) and Spot + On-Demand mix for cost savings.**

**Optionally combine: front-end serverless + backend containers/VMs for heavy workloads.**

1. **Session / State**
   * **Stateless app servers (best practice).**
   * **Persist sessions in ElastiCache (Redis) or DynamoDB (DAX optional) for fast shared session/state.**
2. **Database(s)**
   * **Relational: Amazon Aurora (MySQL/Postgres) with Multi-AZ and Aurora Replicas. For global apps use Aurora Global DB.**
   * **NoSQL: DynamoDB for highly scalable key-value access, on-demand or provisioned with autoscaling.**
   * **Backups: automated snapshots, point-in-time recovery (PITR), cross-region snapshots for DR.**
3. **Object & file storage**
   * **S3 for uploads, backups, static sites. Use lifecycle rules to move to S3 Standard-IA / Glacier for older data.**
4. **Cache & Search**
   * **ElastiCache (Redis or Memcached) for DB caching and sessions.**
   * **Amazon OpenSearch Service for search and analytics (or managed Elastic).**
5. **Messaging & async**
   * **SQS for decoupling and resilient queues.**
   * **SNS for pub/sub notifications and triggering Lambda.**
   * **EventBridge for event routing and integrations.**
6. **Observability & Ops**

* **CloudWatch for metrics, alarms, logs. Use CloudWatch Logs Insights for analysis.**
* **X-Ray for distributed tracing.**
* **Centralized logging to S3 or a log analytics platform.**
* **Use health checks (ALB / EC2) and ASG lifecycle hooks for graceful startup/termination.**

1. **Security**

* **VPC with public/private subnets (public for ALB/NAT in each AZ; private for app & DB).**
* **Security Groups + NACLs for network zoning.**
* **IAM least privilege, roles for services.**
* **AWS WAF on CloudFront/ALB; AWS Shield advanced if needed.**
* **KMS for encryption at rest; enforce TLS for in-transit.**
* **Secrets in AWS Secrets Manager or SSM Parameter Store (secure).**

**💡 Cost-optimization strategies (practical)**

* **Right-size** resources\*\*:\*\* monitor CPU/memory and tune instance/container sizes.
* **Autoscaling**: scale-in aggressively for base workloads and scale-out smoothly for spikes.
* **Spot Instances / Spot Fargate**: use for stateless, fault-tolerant workers; combine with On-Demand for critical capacity.
* **Savings Plans / Reserved Instances** for predictable baseline usage (compute and DB).
* **Aurora Serverless v2** (or RDS Serverless where applicable) for variable DB workloads.
* **S3 lifecycle policies** to move cold data to cheaper tiers automatically.
* **Use CloudFront caching** to reduce origin load and egress costs.
* **Turn off nonprod** resources outside working hours or run in cheaper instance families.
* **Use cost allocation tags** and budgets/alerts.

**📈 High availability & resilience tactics**

* **Multi-AZ deployments** for ALB, app, DB replicas — survive AZ failure.
* **Multi-Region** for critical apps: active-active (Aurora Global / DynamoDB global tables + Route53 latency/geo routing) or active-passive with failover.
* **Health checks + Auto Recovery** for EC2.
* **Backups and cross-region snapshots**; test restores regularly.
* **Chaos engineering** (e.g., occasional simulated AZ failure) to validate resilience.

**✍️ Example service mapping (one-liner)**

* CloudFront → ALB → (ECS Fargate services / EC2 ASG / Lambda) → ElastiCache + Aurora (Multi-AZ) → S3 for static + backups; SQS for async; CloudWatch/X-Ray for monitoring; IAM/KMS/Secrets Manager for security.
  1. **How does LB handle sudden traffic spikes?**

**1. Horizontal Load Distribution**

* **Elastic Load Balancer (ELB/ALB/NLB)** sits in front of your compute (EC2, ECS, Lambda, etc.).
* When a spike occurs, the LB **spreads incoming connections across multiple healthy targets** (instances/containers).
* This prevents a single server from being overwhelmed.

**2. Pre-Warming & Elastic Scaling of LB Itself**

* Modern AWS Load Balancers (ALB, NLB) are **fully elastic** — they automatically scale their capacity to handle millions of requests per second.
* AWS internally “pre-warms” the load balancer if it detects a sharp increase in traffic.
* If you expect **extremely large predictable spikes** (e.g., Super Bowl ads, Black Friday launches), you can **request AWS to pre-warm** the LB to handle the surge immediately.

**3. Health Checks & Target Deregistration**

* During a spike, if some targets become unhealthy, the LB automatically **stops routing traffic** to them and redistributes load to healthy ones.
* This helps avoid cascading failures.

**4. Integration with Auto Scaling Groups**

* The LB works hand-in-hand with **ASG (Auto Scaling Groups)**:
  + ASG scales out (adds new EC2 instances/containers) when CPU/requests cross thresholds.
  + LB automatically **registers new instances** into its target group.
  + This ensures backend capacity grows in response to spikes.

**5. Connection Management & Offloading**

* LB handles **TCP/SSL termination** and keeps long-lived connections to the backend.
* During spikes, this prevents servers from wasting resources on connection handling.
* Example: ALB terminates HTTPS and forwards HTTP to the backend → saves CPU on instances.

1. **Route 53**
   1. **What is hosted zone in route 53?**

A hosted zone is a container that holds information ( as records) about how you want to route traffic for a domain and its subdomains.

🔹 Key Concepts

* Domain Name: The hosted zone is always tied to a domain name (like example.com).
* Records: Within a hosted zone, you create DNS records (A, AAAA, CNAME, MX, TXT, SRV, NS, etc.).
* Delegation: When you create a hosted zone, Route 53 assigns four authoritative name servers. To make the hosted zone active, you must update your domain registrar to point to these NS records.
* **Public Hosted Zone → For domains accessible over the internet (e.g., myapp.com).**
* **Private Hosted Zone → For internal domains accessible only inside a VPC (e.g., myapp.local).**
  1. **Different Route 53 routing policies**

**🔹 1. Simple Routing Policy**

* **Default type: Routes all traffic to a single resource.**
* **Example:**
  + **example.com → single EC2 instance or ALB.**
* **Use case: Small websites or apps with one server.**

**🔹 2. Weighted Routing Policy**

* **Splits traffic across multiple resources by assigning weights (percentages).**
* **Example:**
  + **example.com → EC2-1 (70%), EC2-2 (30%).**
* **Use case:**
  + **A/B testing, gradual rollouts (e.g., shifting traffic slowly to a new version).**

**🔹 3. Latency-Based Routing (LBR)**

* **Sends traffic to the region with the lowest network latency for the user.**
* **Example:**
  + **US users → US-East-1,**
  + **India users → AP-South-1.**
* **Use case: Improve performance by directing users to the nearest AWS region.**

**🔹 4. Failover Routing Policy**

* **Defines primary and secondary (standby) endpoints.**
* **Route 53 checks health → if primary fails, it sends traffic to secondary.**
* **Example:**
  + **Primary ALB in us-east-1,**
  + **Secondary ALB in us-west-2.**
* **Use case: High availability, disaster recovery.**

**🔹 5. Geolocation Routing Policy**

* **Routes traffic based on user’s geographic location (continent, country, or state).**
* **Example:**
  + **US traffic → US servers,**
  + **EU traffic → EU servers.**
* **Use case:**
  + **Localized content (language-specific sites),**
  + **Compliance with regional laws (like GDPR).**

**🔹 6. Geoproximity Routing Policy *(with Traffic Flow only)***

* **Routes traffic based on geographic location of users AND resources, and lets you bias traffic to one resource over another.**
* **Example:**
  + **Push more traffic to us-east-1 even if latency is slightly higher, by applying a bias factor.**
* **Use case:**
  + **Controlling regional traffic distribution beyond just latency.**
  1. **What is the difference between Alias and CNAME records?**

🔹 CNAME Record (Canonical Name Record)

* Standard DNS record defined by DNS specs.
* Maps one domain name to another domain name.
* Example:
  + www.example.com → example.com.
  + app.example.com → myapp.s3-website-us-east-1.amazonaws.com.

⚠️ Limitations of CNAME:

* You cannot use CNAME at the root domain (also called apex domain, e.g., example.com).
  + Because DNS specs don’t allow a CNAME at the zone apex — only NS and SOA are permitted.
* Always incurs an extra DNS lookup → slightly more latency.

🔹 Alias Record (AWS-specific)

* A special Route 53 extension to DNS.
* Similar to CNAME, but can point directly to AWS resources (ELB, CloudFront, S3 static website, API Gateway, Global Accelerator, etc.).
* Can be used at the root domain (apex) → example.com → ALB or CloudFront.
* Alias lookups are handled by Route 53 without adding extra DNS queries → no extra cost, no added latency.

Example:

* example.com → Alias to CloudFront distribution.
* example.com → Alias to ALB DNS name.
  1. What is Resolver in Route 53 and how does it help in hybrid cloud?
* **Route 53 Resolver** provides **DNS resolution between AWS and on-premises** networks.
* **Inbound endpoints:** Allow on-prem servers to resolve AWS private zone records.
* **Outbound endpoints:** Allow AWS workloads to resolve on-prem DNS names.

**Scenario:**

* Hybrid environment where AWS apps need to resolve corp.local (on-prem AD DNS).
  1. How does Route 53 handle DNS Failover for an S3 static website?
* Create an **S3 bucket website** in 2 regions (Primary & Secondary).
* Configure **Route 53 health checks** → if primary bucket unavailable, failover to secondary.
* Use **Alias record** pointing to S3 website endpoint.
  1. How does Route 53 integrate with Multi-region Disaster Recovery?
* Use **Failover Routing** policy:
* Primary → us-east-1 ALB
* Secondary → eu-west-1 ALB
* Combine with health checks so if primary fails, DNS automatically shifts traffic to secondary region.
  1. How do Route 53 health checks work with ELB and CloudFront?

**🔹 1. Route 53 Health Checks – Basics**

* A **Route 53 health check** is an independent monitoring system.
* It sends **HTTP/HTTPS/TCP requests** to an endpoint (IP, domain, or AWS resource).
* If the health check fails for a threshold period, Route 53 marks the endpoint as **unhealthy**.
* Health checks can be attached to:
  + DNS records (e.g., weighted or failover routing).
  + **CloudFront or ELB aliases** (special handling here).

**🔹 2. With Elastic Load Balancers (ELB)**

* **Native integration:** If you create an **Alias record** in Route 53 pointing to an ALB/NLB/CLB, **you don’t need to create a separate Route 53 health check**.
* Instead, Route 53 queries the ELB’s own health status. ELB continuously checks registered EC2 targets using its own health checks.
* Route 53 considers the ELB **healthy if it has at least one healthy target** in the load balancer’s target group.
* If all targets behind the ELB are unhealthy, Route 53 marks the Alias record as **unhealthy** and removes it from DNS responses (depending on your routing policy).

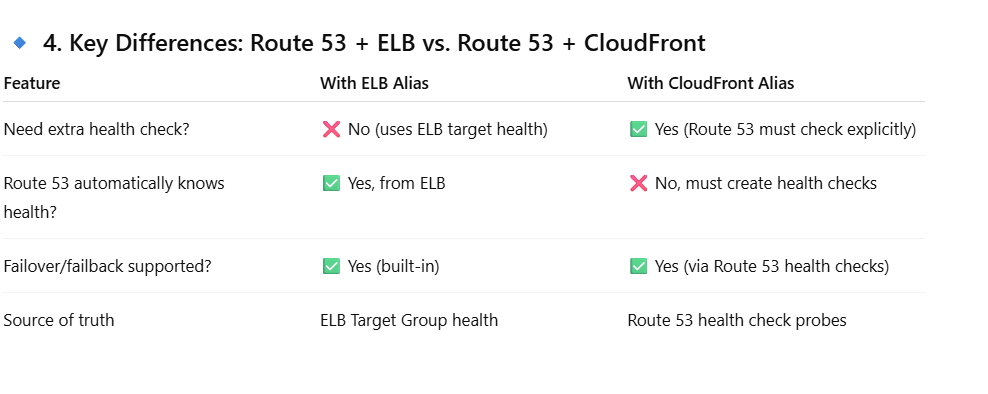
👉 **Example scenario**:  
You have a failover setup with two ALBs in different regions. Route 53 will automatically use the ALB’s internal health checks to decide which region to route traffic to.

**🔹 3. With CloudFront**

* **Health check behavior is different**:
  + You cannot attach a Route 53 health check directly to a CloudFront distribution Alias.
  + CloudFront itself doesn’t report target health to Route 53 like ELB does.
* Instead, if you need health-based routing with CloudFront, you must:
  + Create **Route 53 health checks** that probe the CloudFront distribution (or its origin).
  + Attach those health checks to DNS records in a **routing policy** (failover, weighted, etc.).

👉 **Example scenario**:  
You have two CloudFront distributions, one in **US** and another in **EU**, each pointing to regional origins.

* You set up Route 53 health checks against each distribution’s public DNS.
* In case the US distribution fails, Route 53 directs traffic to the EU distribution.



* 1. Explain TTL (Time-to-Live) tuning in Route 53 and its impact.
* **Low TTL (30s–60s):** Faster failover, more DNS queries → higher cost.
* **High TTL (300s–3600s):** Reduces cost, but slower failover.

**Scenario:**

* For Blue/Green deployment → use **low TTL** for quick cutover.
* For static resources → use **high TTL** to reduce cost.

1. **Cloudfront and API Gateway**
   1. **What is the usage of API gateway?**

Amazon API Gateway is a fully managed service that allows you to create, publish, secure, monitor, and manage APIs at any scale.  
It acts as a “front door” for applications to access backend services like Lambda functions, EC2, ECS, DynamoDB, or other HTTP endpoints.

**🔹 Usage of API Gateway in AWS Cloud**

**1. Expose Backend Services as APIs**

* Wrap microservices, Lambda functions, or databases behind a REST or HTTP API.
* Converts internal logic into standard endpoints (GET, POST, PUT, etc.).  
  ✅ Example: Create a /orders API that triggers a Lambda to store order details in DynamoDB.

**2. Serverless Applications**

* API Gateway is often paired with **AWS Lambda** to build **serverless apps**.
* No need to manage servers — just write the function logic, expose it via API Gateway, and scale automatically.  
  ✅ Example: A serverless CRUD application using API Gateway + Lambda + DynamoDB.

**3. Security & Access Control**

* Provides **authentication & authorization** using:
  + **IAM roles & policies**
  + **Amazon Cognito (user pools & identity pools)**
  + **Custom authorizers (JWT, OAuth, API keys, Lambda authorizers)**
* Supports **throttling, quotas, and WAF (Web Application Firewall)** to prevent abuse.

✅ Example: Only logged-in users from Cognito can call /checkout.

**4. Traffic Management & Scaling**

* API Gateway handles **traffic spikes**, caching, and rate limiting.
* Provides **auto-scaling** without you managing infrastructure.  
  ✅ Example: A flash sale app handling millions of API calls per second.

5. Protocol Translation

Supports REST APIs, WebSocket APIs, and HTTP APIs.

Converts different communication protocols for clients & backends.

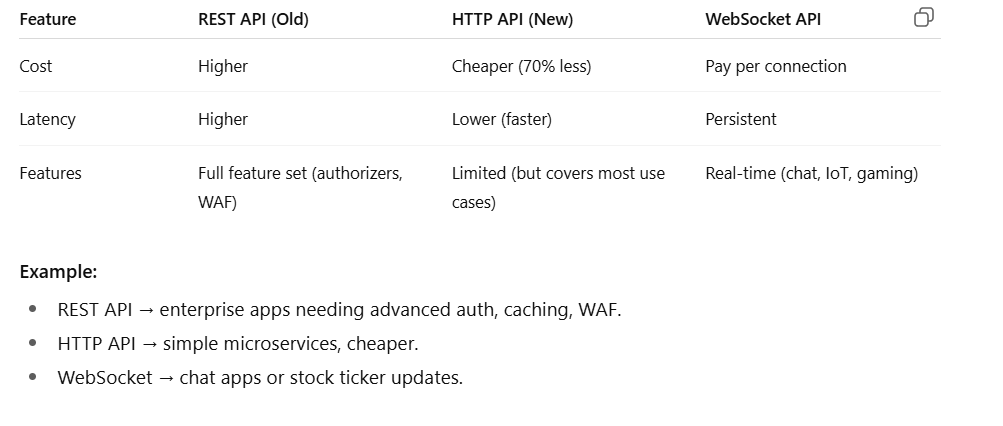
✅ Example: Mobile app uses WebSocket API for live chat, backend uses Lambda.

**7. Monitoring & Analytics**

* Built-in integration with **CloudWatch** for metrics, logging, and alarms.
* Helps track **latency, error rates, and API usage**.

**8. Cost Optimization**

* Pay only for **API calls + data transfer**.
* Cheaper than running your own reverse proxy / API management service on EC2.
  1. Explain the difference between REST API, HTTP API, and WebSocket API in API Gateway.



* 1. How do you secure an API Gateway endpoint?
* **Authentication/Authorization:**
  + IAM roles & policies (for internal AWS apps)
  + Cognito User Pools (OAuth/JWT tokens)
  + Lambda Authorizers (custom logic, e.g., RBAC)
* **Throttling & Rate Limits:** Prevent abuse (e.g., 1000 req/s).
* **WAF Integration:** Protect against SQLi/XSS.
* **Private Endpoints:** Restrict access within VPC.
  1. How does API Gateway caching work?
* **Enable cache** per stage → reduces Lambda/DynamoDB calls.
* Configurable **TTL (default 300s)**.
* Cache can be invalidated programmatically.

**Scenario:**  
If /products API is queried frequently → enable cache → fewer DynamoDB reads → lower cost and latency.

* 1. How do you handle versioning in API Gateway?

**🔹 Why Versioning is Needed**

* APIs evolve over time (new features, bug fixes, changes in response format).
* Breaking changes can disrupt existing applications.
* Versioning provides a way to support **multiple API versions simultaneously**.

**🔹 Methods of Handling Versioning in API Gateway**

**1. URI Path Versioning (Most Common)**

* Embed the version number in the API path.
* Example:
* https://api.example.com/v1/orders
* https://api.example.com/v2/orders
* Each version can map to different **API Gateway resources and Lambda functions**.

✅ **Pros**: Clear, simple, easy for clients.  
❌ **Cons**: Can create duplication in routes if many versions exist.

**2. Stage-Based Versioning**

* API Gateway supports **stages** (e.g., dev, test, prod).
* You can also treat a stage as a version:
* https://api.example.com/v1/orders → Stage v1
* https://api.example.com/v2/orders → Stage v2
* Stages can point to different **deployments** of your API.

✅ **Pros**: Easy to manage via deployments.  
❌ **Cons**: Versioning tied to environment concept, may get confusing if mixing dev/prod stages.

**3. Custom Domain + Base Path Mapping**

* Use a **custom domain** and **base path mapping** to map versions.
* Example:
  + Custom domain: api.example.com
  + Base path mappings:
    - /v1 → API Gateway deployment 1
    - /v2 → API Gateway deployment 2

✅ **Pros**: Clean domain setup, allows smooth transition from one version to another.  
❌ **Cons**: Extra setup with custom domains.

**4. Header-Based Versioning**

* Clients send API version in headers (e.g., Accept or custom header).
* Example:
* GET /orders
* Accept: application/vnd.example.v2+json
* API Gateway routes requests to appropriate backend integration (via **Lambda or Mapping Templates**).

✅ **Pros**: Keeps URLs clean, flexible.  
❌ **Cons**: Harder for caching/CDNs, clients must manage headers correctly.

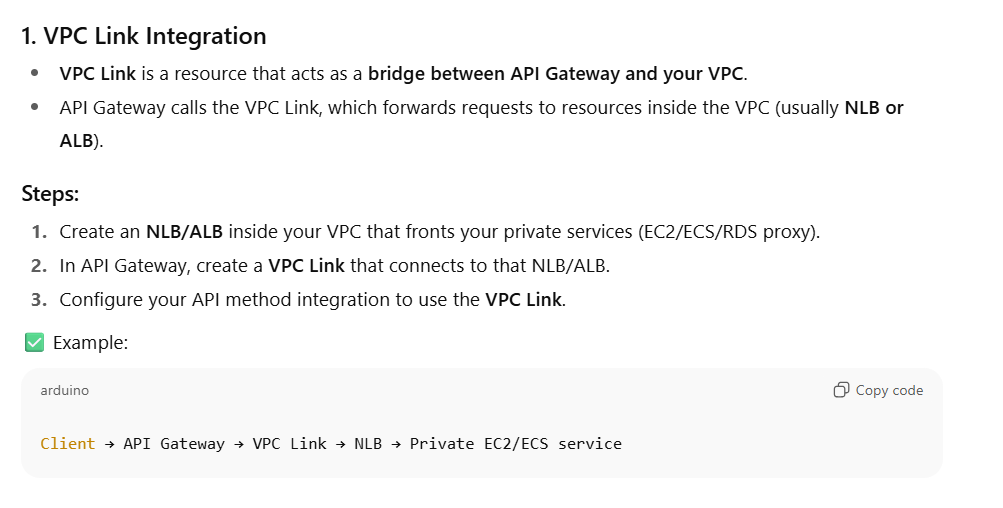
**5. Query String Versioning (Less Common)**

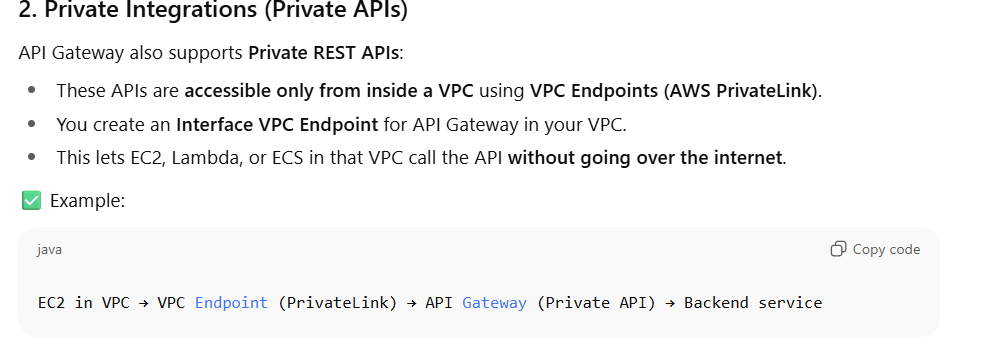
* Version passed as query parameter.
* Example:
* https://api.example.com/orders?version=2

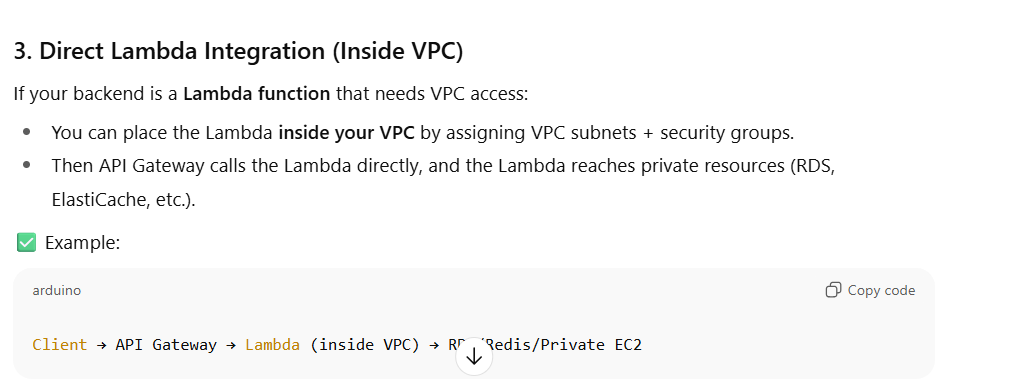
✅ **Pros**: Easy to implement.  
❌ **Cons**: Not REST-ideal, caching complications, less explicit.

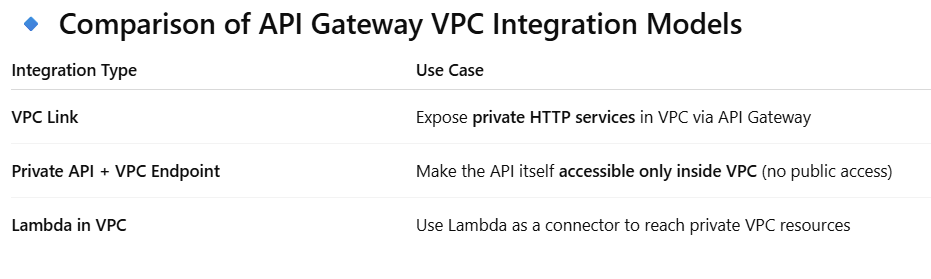
**🔹 Best Practices for Versioning in API Gateway**

1. **Use path versioning (/v1, /v2) for major versions** (breaking changes).
2. **Use stages** for environment separation (dev, qa, prod), not for versioning — unless it’s a controlled setup.
3. **Deprecate old versions gradually** with proper communication.
4. **Combine API Gateway with Lambda Aliases/Versions** to keep backend changes aligned with API versions.
5. **Monitor usage** with CloudWatch/Usage Plans to know when old versions can be retired.
   1. How does API Gateway integrate with VPC?









* 1. How do you handle throttling and request limits in API Gateway?

**🔹 1. API Gateway Throttling Basics**

* **Throttling** = limiting the rate of API requests.
* API Gateway enforces throttling at **two levels**:
  1. **Account-level throttling** (applies to all APIs in your account).
  2. **API-level & Method-level throttling** (applies to specific API stages or methods).

The limits are measured in:

* **Requests per second (RPS)**
* **Burst capacity** (short-term spike allowance before throttling kicks in).

**🔹 2. Default Throttling in API Gateway**

* By default, AWS sets a **soft limit** per account/region:
  + **10,000 RPS** (requests per second)
  + **5,000 concurrent requests**
* These can be **increased by AWS support** if needed.

**🔹 3. Stage & Method-Level Throttling**

* You can configure throttling per **stage** or per **method** in API Gateway.
* Example:
  + Stage: /prod → 500 RPS, burst 1000
  + Method: POST /orders → 50 RPS, burst 100

✅ This allows you to protect **sensitive endpoints** separately.

**🔹 4. Usage Plans & API Keys**

For **per-customer throttling**:

* Create a **Usage Plan** that defines:
  + **Rate limit** (RPS)
  + **Burst limit**
  + **Quota** (e.g., 1M requests per month)
* Attach the plan to **API Keys**.
* Clients authenticate with API Keys, and their usage is tracked/enforced.

✅ Example:

* Free tier plan: 10 RPS, 100k requests/month
* Premium plan: 100 RPS, 1M requests/month

**🔹 5. How Throttling Works (Flow)**

1. Client sends requests → API Gateway.
2. API Gateway checks **account-level, stage-level, and method-level limits**.
3. If API Key is used, applies **Usage Plan limits**.
4. If limits exceeded → returns **429 Too Many Requests**.
   1. How do you troubleshoot 429 Too Many Requests in API Gateway?

**🔹 Troubleshooting Steps**

**1. Check CloudWatch Metrics**

* Each API Gateway stage/method publishes metrics:
  + Count → total requests.
  + 4XXError → number of client errors.
  + ThrottledRequests → number of 429s.
* Identify whether throttling is happening at **account level, stage/method, or usage plan**.

👉 Look at:

* ThrottledRequests (per method)
* LimitExceeded (for account-level)

**🔹 Fixes / Mitigations**

1. **Increase Limits**
   * If hitting **account-level**: Request AWS service quota increase.
   * If hitting **stage/method**: Adjust throttling settings in API Gateway.
2. **Adjust Usage Plans**
   * Raise RPS/burst limits for premium customers.
   * Reset quota if mistakenly exhausted.
3. **Implement Client-Side Retry**
   * Use **exponential backoff with jitter** (best practice).
   * Avoid hammering API with retries after 429.
4. **Use Caching**
   * Enable **API Gateway caching** to reduce calls to backend.
   * Clients get faster responses, backend load decreases.
5. **Distribute Load**
   * If multiple clients → separate **usage plans** and API keys.
   * Prevents one client from starving others.
   1. **Why we use cloudfront?**

🔹 Usage of Amazon CloudFront

**1. Content Delivery (CDN)**

* Distributes **static content** (HTML, CSS, JS, images, videos, downloads) to edge locations worldwide.
* Users fetch content from the **nearest edge location**, reducing latency.  
  ✅ Example: Global website serving images faster to users in Europe, Asia, US.

**2. Dynamic Content Acceleration**

* Not just static files — CloudFront can also accelerate **dynamic APIs, live streaming, and personalized content**.
* Uses **TCP/UDP optimizations** to speed up communication between clients and origin.  
  ✅ Example: API responses from an ALB or API Gateway are served faster.

**3. Security**

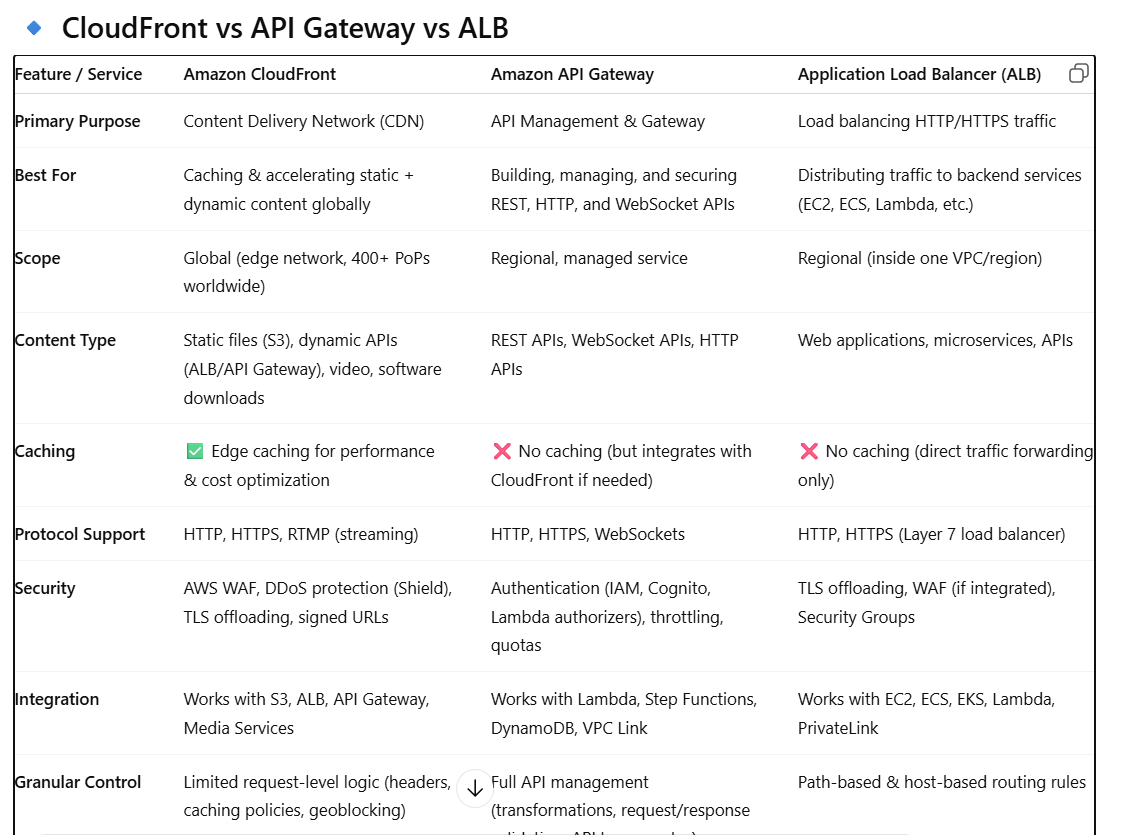
* **DDoS protection** (integrates with AWS Shield).
* **Web Application Firewall (AWS WAF)** for filtering malicious requests.
* **TLS/SSL termination** for HTTPS.
* **Geo-blocking** to restrict regions.  
  ✅ Example: Block traffic from unwanted geographies before it hits your servers.

**4. Cost Optimization**

* **Reduces load on your origin servers (S3, EC2, ALB).**
* **Cached content served from edge locations → fewer requests to origin → lower bandwidth + compute costs.  
  ✅ Example: Offload 80% of image traffic from EC2 to CloudFront edge caches.**

**5. Access Control**

* Restrict content with:
  + **Signed URLs & Cookies** (only authorized users can download).
  + **Origin Access Control (OAC)** → ensures S3 content is **not public**, only accessible via CloudFront.  
    ✅ Example: Video streaming site where only paid users can access movies.
  1. Difference between Cloudfront, API Gatway and Load balancer.



A screenshot of a phone

AI-generated content may be incorrect.

**🔹 How They Work Together**

* **CloudFront + API Gateway**:
  + CloudFront at the edge accelerates API responses.
  + API Gateway manages & secures the APIs.  
    👉 Example: Mobile app calling api.example.com → CloudFront → API Gateway → Lambda.
* **CloudFront + ALB**:
  + CloudFront caches static + dynamic content.
  + ALB load balances backend microservices.  
    👉 Example: Global e-commerce site → CloudFront → ALB → EC2/ECS.
* **API Gateway + ALB**:
  + API Gateway for public API management.
  + ALB inside VPC for microservice routing.  
    👉 Example: API Gateway → VPC Link → ALB → ECS microservices.
* **CloudFront + API Gateway + ALB (all three)**:
  + CloudFront = global CDN + DDoS shield.
  + API Gateway = API management + throttling.
  + ALB = routes traffic to backend services.  
    👉 Example: High-scale SaaS platform with APIs + web apps.
  1. **What are the various policies and functions used in cloudfront?**

🔹 1. Cache Policies

Purpose: Control what CloudFront caches, how long it caches, and how requests are considered unique for caching.

* Default Behavior: By default, CloudFront caches based on the full URL path and ignores query strings, cookies, or headers unless configured.

Key Features:

1. TTL Settings
   * Minimum TTL → minimum time an object stays in cache
   * Maximum TTL → max time before CloudFront revalidates with the origin
   * Default TTL → default cache duration
2. Include HTTP Request Elements
   * Headers → include specific headers in the cache key (e.g., Accept-Language)
   * Query Strings → include certain query strings for caching
   * Cookies → include selected cookies in cache key

🔹 2. Origin Request Policies

Purpose: Control what CloudFront forwards to the origin server.

* You can control which request components (headers, cookies, query strings) are sent from CloudFront to the origin.

Key Features:

* Forward selected headers (like Authorization)
* Forward cookies selectively
* Forward query strings

🔹 3. Response Headers Policies

Purpose: Add or modify HTTP response headers returned by CloudFront to the client.

Key Features:

* Security headers → Strict-Transport-Security, Content-Security-Policy, X-Frame-Options
* CORS headers → Access-Control-Allow-Origin, Access-Control-Allow-Methods
* Custom headers → any custom headers your application needs

🔹 4. Function Policies (CloudFront Functions & Lambda@Edge)

CloudFront allows code execution at the edge for request/response customization.

4.1 CloudFront Functions

* Lightweight JavaScript functions that execute on viewer request or response
* Low latency, ideal for simple manipulations like:
  + URL rewrites
  + Header manipulation
  + Access control

4.2 Lambda@Edge

* Full Lambda functions deployed to CloudFront edge locations
* Can execute on:
  + Viewer Request
  + Viewer Response
  + Origin Request
  + Origin Response

🔹 5. Origin Policies

* Control how CloudFront communicates with origin servers:
  + HTTP versions → HTTP/1.1, HTTP/2
  + Origin keep-alive settings
  + Origin SSL protocols
  + Connection timeout / read timeout

🔹 6. Security Policies

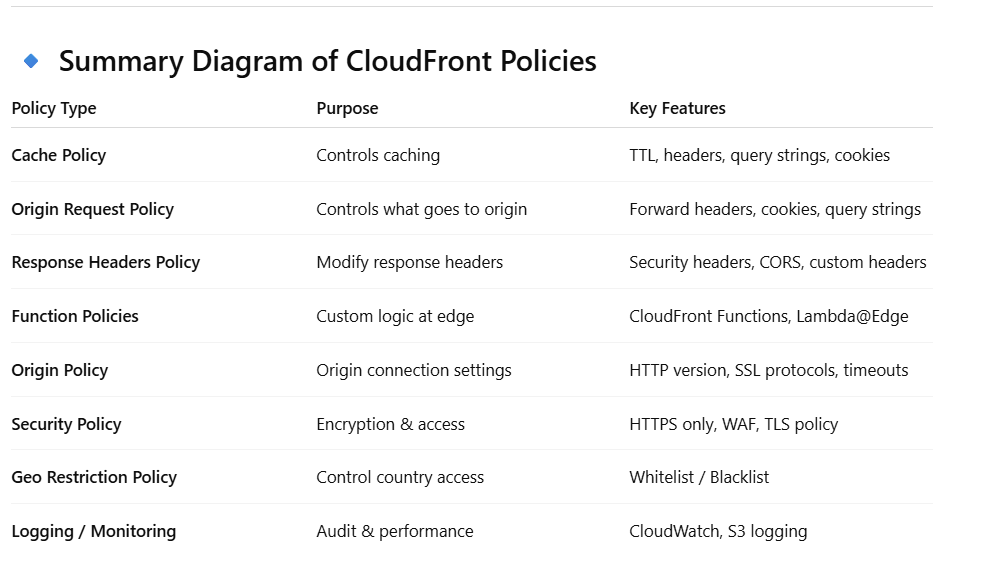
* Control encryption and security at the edge:
  + Viewer Protocol Policy → HTTP vs HTTPS
  + TLS Security Policy → which SSL/TLS versions CloudFront supports
  + WAF Policies → attach AWS WAF rules for firewall protection
  + Geo-restriction → allow/block traffic by country

🔹 7. Geo Restriction Policies

* Restrict content delivery to certain countries
* Modes:
  + Whitelist → only allow specific countries
  + Blacklist → block specific countries

🔹 8. Logging and Monitoring Policies

* Enable CloudFront access logs
* Integrate with CloudWatch metrics
* Optional: Send logs to S3 for analytics

****

* 1. **How do you secure CloudFront distributions?**
* HTTPS only (TLS 1.2/1.3).
* WAF integration for OWASP protections.
* Origin Access Control (OAC) → only CloudFront can access S3 origin.
* Signed URLs / Cookies → restrict content access.
* Geo-blocking → block unwanted regions.
  1. What’s the difference between putting CloudFront in front of API Gateway vs directly exposing API Gateway?
* **Direct API Gateway:** Good for regional APIs.
* **With CloudFront:**
  + Lower latency (global edge caching).
  + Shield + WAF protection.
  + Custom domain management.
  + Better DDoS resilience.

**Best Practice:** Always use **CloudFront + API Gateway** for global, internet-facing APIs.

1. **ECS and FARGATE**
   1. What are the differences between **ECS EC2 launch type** and **Fargate launch type**? When would you use one over the other?

* ECS on EC2
  + You manage the underlying EC2 instances (cluster capacity, scaling, patching).
  + **Good for workloads needing custom AMIs, GPUs, or specific kernel tuning**.
  + Cost-effective if you already have EC2 Reserved Instances/Savings Plans.
* ECS on Fargate
  + Fully serverless — no EC2 management, AWS handles scaling and patching.
  + Pay per vCPU and memory allocated, not for idle EC2 capacity.
  + Best for bursty, unpredictable workloads or when you want to reduce ops overhead.

👉 Rule of thumb: Use Fargate for agility, EC2 for fine-grained control & cost optimization.

* 1. How does auto scaling work in ECS, and how is it different when using Fargate?

**🚀 ECS Auto Scaling Basics**

In ECS, you can scale **in two main ways**:

1. **Service Auto Scaling** – adjusts the number of **tasks** in a service.
2. **Cluster Auto Scaling (for EC2)** – adjusts the number of **container instances (EC2s)** in the cluster to fit tasks.

Scaling can be triggered by **CloudWatch alarms**, **Target Tracking**, or **Step Scaling** policies.

**⚙️ Auto Scaling with ECS on EC2 Launch Type**

When you use **ECS with EC2**, you manage the underlying EC2 cluster. Scaling happens at **two levels**:

**1. Service Auto Scaling**

* ECS can scale the **task count** (e.g., 2 tasks → 10 tasks).
* Scaling policies can be based on metrics like **CPUUtilization**, **MemoryUtilization**, or even **custom CloudWatch metrics** (like request count from ALB).

Example:  
If CPU > 70% for 2 minutes → Scale out by +2 tasks.  
If CPU < 30% for 5 minutes → Scale in by -1 task.

**2. Cluster Auto Scaling (EC2 capacity)**

* If there aren’t enough EC2 instances to place the new tasks, ECS triggers **ECS Cluster Auto Scaling** (CAS) to launch more EC2s via an **Auto Scaling Group**.
* Similarly, if EC2s become underutilized, CAS can terminate them.
* You need to size instances properly (CPU/Memory resources, AMI, instance families).

👉 So in EC2 mode, scaling is a **two-step dance**:  
**Scale tasks → Scale EC2 instances.**

**⚙️ Auto Scaling with ECS on Fargate Launch Type**

With **Fargate**, AWS manages the infrastructure, so you don’t deal with EC2s or ASGs at all.

**1. Service Auto Scaling**

* You still scale **tasks up/down** based on CloudWatch alarms or target tracking (same as EC2).
* Example: If API Gateway requests increase → ALB target group → ECS tasks scale automatically.

**2. No Cluster Auto Scaling**

* No EC2s to manage.
* Fargate directly provisions compute capacity per task, so scaling is **faster** and **simpler**.
* You only pay for CPU + memory requested per task (cost-optimized if tuned well).

👉 So in Fargate mode, scaling is a **single step**:  
**Scale tasks directly (AWS handles infra).**

* 1. What are the networking modes available in ECS, and which one is used by Fargate?
* ECS supports:

1. **Bridge Mode** → Classic Docker bridge network.
2. **Host Mode** → Task shares host’s network interface.
3. **Awsvpc Mode** → Each task gets its own ENI (Elastic Network Interface) with a private IP in your VPC.

* **Fargate ONLY supports awsvpc mode.**
* Each task behaves like an independent EC2 instance with its own IP.
* Easier for security (use SGs per task) and VPC-level networking.
  1. What happens if all ECS tasks behind an ALB fail health checks? How would you troubleshoot?
* **ALB health check config:** Ensure the path (e.g., /health) and port are correct.
* **Task definition networking:** If using awsvpc, verify that the task ENIs allow inbound traffic from the ALB security group.
* **Security groups:** ALB SG must allow outbound traffic to ECS tasks, and ECS task SG must allow inbound from ALB SG.
* **Application issue:** The container may not expose the correct port or may take too long to start (consider **container health checks with containerHealthCheckGracePeriod**).
* **Logs:** Check ECS task logs and ALB access logs for failures.
  1. Your ECS service is behind an ALB and experiencing intermittent 5xx errors under load. How do you debug whether the issue is ECS, ALB, or application-related?
* **Check ALB metrics (CloudWatch):**
  + HTTPCode\_ELB\_5XX (ALB issue)
  + HTTPCode\_Target\_5XX (task/application issue)
* **Check ECS service events:** Are tasks being killed/restarted due to health check failures?
* **Enable ALB access logs:** See whether requests are reaching tasks or failing at the ALB level.
* **Check ECS task logs:** Look for OOM errors, connection resets, or app crashes.
* **Scaling considerations:** If tasks are maxing out CPU/memory, configure scaling before they fail.
* **Network bottlenecks:** Misconfigured SGs/NACLs could cause dropped connections.
  1. Suppose you deployed a microservices application on ECS with ALB. One service suddenly receives 10x more traffic, while others remain stable. How would you handle scaling?
* Configure **service auto scaling** on that particular ECS service only (based on CloudWatch metrics like ALB RequestCountPerTarget or TargetResponseTime).
* Each ECS service has its **own ALB target group**. Only the overloaded service should scale independently, avoiding wasted compute on stable services.
* Use **target tracking scaling** (e.g., maintain 500 requests per target).
* For **Fargate**, scaling is straightforward since tasks spin up independently.
* For **EC2**, ensure **Cluster Auto Scaling** can launch more EC2 instances when needed.
  1. Task Placement Strategies for ECS EC2.

**⚙️ Task Placement in ECS (EC2 Launch Type)**

When you run ECS tasks on **EC2 instances**, ECS must decide:  
👉 *“Which container instance in the cluster should this task be placed on?”*

This is controlled by:

1. **Task Placement Strategies** – *the logic ECS uses to distribute tasks across EC2s*
2. **Task Placement Constraints** – *rules you define to limit which instances can run a task*

**🔹 Common Task Placement Strategies**

You can combine up to **five strategies** per service.

**1. binpack**

* Places tasks on the **least available resources** (CPU or memory).
* Goal: **pack tasks efficiently** on as few instances as possible.
* Reduces unused capacity, saves cost.

✅ Example:  
If you have 4 EC2 instances and want to minimize costs → ECS will keep filling one instance before moving to the next.

* Useful in **cost-sensitive environments**.
* Downside: less HA, because tasks are concentrated.

**2. spread**

* Distributes tasks **evenly across values of a specified attribute**.
* Common attributes:
  + attribute:ecs.availability-zone → Spread tasks across AZs for HA.
  + instanceId → Spread tasks evenly across instances.

✅ Example:  
If you spread across availabilityZone and have 2 AZs with 4 tasks → ECS places 2 tasks in each AZ.

* Ensures **high availability**.
* Good for production workloads requiring resilience.

**3. random**

* Places tasks randomly across instances.
* Least predictable, but may be useful for test environments or when you don’t care about balance.

**🔹 Task Placement Constraints**

Constraints let you filter *where tasks can be placed* before applying strategies.

1. **distinctInstance**
   * Ensures each task is placed on a **different EC2 instance**.
   * Great for HA (no single EC2 has multiple copies).
2. **memberOf**
   * Uses an **expression** to restrict placement (based on instance attributes).
   * Example: attribute:ecs.instance-type =~ t3.\* → Place tasks only on t3 family instances.
   1. Persistent Storage in ECS/Fargate

**⚙️ Persistent Storage in ECS (EC2 vs Fargate)**

By default, **ECS tasks are ephemeral**:

* Container filesystems are destroyed when the task stops.
* Any data written inside the container is lost.

To achieve **persistent storage**, you must use **external storage services**.

**🔹 Options for Persistent Storage in ECS (EC2 Launch Type)**

**1. Amazon EBS (Elastic Block Store)**

* You can attach an EBS volume to an EC2 instance.
* ECS task can mount it via the **EC2 host bind mount**.
* Data persists as long as the volume exists (even if instance stops, if not deleted).

✅ Use case: Databases like MySQL/Postgres running on ECS EC2.  
⚠️ Limitation: EBS volumes are AZ-scoped and tied to a single EC2 instance → not ideal for scaling across multiple instances.

**2. Amazon EFS (Elastic File System)**

* Fully managed, scalable, shared **NFS filesystem**.
* Tasks in ECS EC2 (with awsvpc or bridge networking) can mount an EFS volume.
* Multiple tasks (across AZs, instances) can read/write the same filesystem.

✅ Use case: Web applications needing shared uploads directory, ML jobs sharing datasets.  
⚠️ Slightly higher latency than local disks.

**3. Bind Mounts (Ephemeral)**

* Tasks can mount a host EC2 directory (/mnt/data).
* Data persists **only as long as the EC2 instance lives**.
* Not truly persistent across restarts.

✅ Use case: Temporary caching, logs.

**🔹 Options for Persistent Storage in ECS (Fargate Launch Type)**

Since **Fargate has no underlying EC2 host you can manage**, storage options are limited but simplified.

**1. Amazon EFS (Recommended)**

* Native integration with ECS Fargate tasks.
* Mount EFS volumes directly in task definitions.
* Data persists across task restarts and can be shared between tasks.

✅ Best option for persistence in Fargate.

**2. Ephemeral Storage (Temporary)**

* Each Fargate task gets **20 GiB ephemeral storage by default**.
* Can be increased up to **200 GiB per task** (configurable in task definition).
* Storage is lost when the task stops.

✅ Use case: Caching, temporary file processing, builds.

**3. S3 for Object Storage**

* Not a filesystem, but commonly used for persistence.
* Containers interact with **S3 buckets** via SDK/CLI.
* Durable, cross-region, highly available.

✅ Use case: Logs, media files, data lakes.

* 1. Security in ECS and Fargate

**1️⃣ Security in ECS with EC2 Launch Type**

When running ECS on **EC2 instances**, you are responsible for **securing both the containers and the underlying EC2 hosts**.

**Key Areas:**

* **Host Security**
  + Harden EC2 AMIs (use ECS-optimized AMIs or Bottlerocket).
  + Patch hosts regularly.
  + Restrict SSH access (use SSM instead of direct SSH).
* **IAM Roles**
  + **Instance IAM Role**: The EC2 instances in the ECS cluster need permissions (e.g., to pull images from ECR).
  + **Task IAM Role**: Each ECS task can assume its own role (fine-grained access control).
* **Networking**
  + Control traffic with **Security Groups** and **NACLs**.
  + With awsvpc networking, each task gets its own ENI and can have separate SGs.
* **Storage Security**
  + Encrypt EBS volumes attached to EC2.
  + Encrypt data at rest/in transit (TLS).
* **Monitoring**
  + Enable **CloudWatch Logs** and **Container Insights**.
  + Use AWS GuardDuty + Inspector for threat detection.

**2️⃣ Security in ECS with Fargate Launch Type**

With Fargate, AWS manages the **host infrastructure**, reducing your responsibility.  
You focus on **container + network + IAM**.

**Key Areas:**

* **No Host Management**
  + No SSH, no patching required.
  + Reduces attack surface (AWS handles infra).
* **Task IAM Roles**
  + Each task can assume its own IAM Role → follows **least privilege principle**.
* **Networking**
  + Tasks run in **awsvpc mode only**, each gets an ENI.
  + Can apply **per-task Security Groups**, isolating workloads at the task level.
  + Supports **private subnets + NAT Gateway** for outbound access.
* **Storage Security**
  + **EFS volumes** can be encrypted with KMS.
  + **Ephemeral storage** (20–200 GiB per task) is encrypted at rest by default.
* **Isolation**
  + Fargate tasks are isolated at the kernel level → stronger security compared to EC2 shared instances.

1. **Networking**
   1. Your company is deploying a 3-tier web app (web, app, DB) in AWS. Each tier must be isolated, but you want **high availability** and **scalability** while keeping costs optimized. How would you design subnets and control communication between them?

**1) High-level topology (one VPC, multi-AZ)**

* Create a single VPC (pick a CIDR large enough, e.g. /22 or /21 depending on scale). Spread resources across **≥2 Availability Zones** (3 AZs is ideal for production). AWS recommends creating subnets in multiple AZs for high availability. [AWS Documentation+1](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-best-practices.html?utm_source=chatgpt.com)
* For each AZ create **three subnet groups** (so you get the 3 tiers across AZs):
  + **Public subnet** (one per AZ) — hosts ALB / NAT Gateway (if you choose NAT gateway per-AZ) / bastion (if used).
  + **Private application subnet** (one per AZ) — hosts app servers (Autoscaling Group for EC2) or private Fargate tasks.
  + **Private DB subnet (isolated)** (one per AZ) — hosts RDS / Aurora (multi-AZ clustering or reader replicas).

This gives you AZ-level redundancy for each tier and lets ASGs scale within AZs

**2) Route tables & routing**

* **Public subnet route table**: 0.0.0.0/0 → Internet Gateway (IGW). Place ALB here (ALB nodes are put in public subnets). [AWS Documentation](https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html?utm_source=chatgpt.com)
* **Private app subnet route table**: route to Internet via NAT Gateway(s) in public subnet(s) for outbound access (software updates, pulling container images). Use **one NAT Gateway per AZ** for true AZ fault tolerance (more cost). If you need to optimize cost, you can place a single NAT Gateway in one AZ (cheaper, but single AZ NAT increases outage risk). Use VPC endpoints to reduce NAT egress cost. [AWS Documentation+1](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-example-private-subnets-nat.html?utm_source=chatgpt.com)
* **DB (isolated) subnet route table**: **no route to IGW** (no direct internet). If DB needs to reach S3/backups, prefer **VPC endpoints (gateway for S3, interface endpoints for other AWS APIs)** so DB traffic never needs to go out to the public internet or NAT. This keeps DB subnet truly isolated.

**3) Security controls — Security Groups (primary) and NACLs (secondary)**

Use **security groups (SGs)** as your primary, east-west + north-south access control because they are stateful and operate at instance/service level.

Suggested SGs (names and rules):

1. **ALB-SG (frontend)**
   * Inbound: HTTP(80)/HTTPS(443) from 0.0.0.0/0 (or from a restricted CIDR if you limit traffic).
   * Outbound: allow to App-SG on the app port (e.g., 80/443/whatever).
2. **App-SG (application tier)**
   * Inbound: allow only from **ALB-SG** on the app port. (Reference other SG by id — more secure than CIDR.)
   * Outbound: allow to DB-SG on DB port (e.g., 5432 or 3306) and to necessary external endpoints (S3, ECR) via endpoints/NAT.
3. **DB-SG (database tier)**
   * Inbound: allow only from **App-SG** on DB port(s). Optionally allow a management CIDR (your office / bastion SG) for DB admin.
   * Outbound: only what DB needs (usually minimal — perhaps to monitoring endpoints).

Key points:

* Use **SG-to-SG** rules rather than IP ranges for east-west traffic (this is more secure and simpler to manage). [AWS Documentation](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html?utm_source=chatgpt.com)
* Keep SGs narrow — least privilege.

**Network ACLs (NACLs)**:

* Use NACLs as a coarse subnet-level guardrail or for compliance where stateless filtering / explicit denies are required. For most apps, keep the default permissive NACL for simplicity and rely on SGs. If you add NACLs, remember they’re **stateless**, so you must add both inbound & outbound rules. Use NACLs to block broad malicious CIDR blocks or to enforce explicit deny rules at subnet boundary.

**4) Load balancing, autoscaling & HA**

* Place an **Application Load Balancer (ALB)** in the public subnets, with targets in the private app subnets (ASG). ALB distributes across AZs automatically. [AWS Documentation](https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html?utm_source=chatgpt.com)
* App tier: Use an **Auto Scaling Group** spanning the private app subnets in each AZ. Scale on CPU / request latency / custom metrics.
* DB tier: Use **RDS Multi-AZ** or **Aurora (writer + readers)** for high availability and read scale. Keep DB replicas in separate AZs/subnets.

**5) VPC endpoints, private connectivity, and management**

* **Gateway VPC Endpoint** for S3 (avoids NAT egress and saves cost + increases security). **Interface VPC Endpoints** (AWS PrivateLink) for services like ECR, SSM, Secrets Manager, KMS, etc., so your private instances can access AWS services without internet. This reduces NAT bandwidth and cost, and improves security. [Amazon Web Services, Inc.+1](https://aws.amazon.com/blogs/database/best-practices-for-creating-a-vpc-for-amazon-rds-for-db2/?utm_source=chatgpt.com)
* **Bastion / SSM Session Manager**: don’t open SSH to the world. Prefer SSM Session Manager (no inbound port open) or a hardened bastion in a restricted public subnet. Use IAM + session logging.
  1. **Your EC2 instance is in a private subnet. You configured the Security Group correctly, but traffic from a web server in another subnet is still failing.**

**1. Verify the subnets’ routing**

* If your **EC2 is in a private subnet**, check that:
  + The route table for the **web server’s subnet** can reach the target EC2’s subnet (both are inside the same VPC by default, so they should have a “local” route to each other).
  + If they are in **different VPCs**, you’d need **VPC peering / Transit Gateway** and proper routes on both sides.  
    👉 **Common pitfall**: People forget to update the route table on the *other side* when using peering.

**2. Check Network ACLs (NACLs)**

* Even if your **security groups are correct**, a restrictive NACL could block the traffic.
* NACLs are **stateless**, so you must allow both inbound and outbound rules explicitly. For example:
  + Inbound rule on target subnet NACL must allow the web server’s port (e.g. 80, 443, custom app port).
  + Outbound rule must allow the **ephemeral response ports (1024–65535)** back.  
    👉 **Very common interview trap**: Security Groups are fine, but NACLs silently drop packets.
  1. NAT Gateway Cost Optimization 🡪 You have 6 private subnets across 3 AZs. Each subnet currently routes to its own NAT Gateway. Your CFO complains about high costs. How would you redesign while maintaining availability?
* NAT Gateway charges **per AZ**. Having 6 NAT Gateways is costly.
* Optimized approach:
  + Create **1 NAT Gateway per AZ** (total 3).
  + Update **route tables** so all private subnets in an AZ route traffic to the NAT Gateway in the same AZ.
  + Avoid routing to a NAT Gateway in a different AZ → reduces cross-AZ data transfer charges.
* This keeps **HA** (each AZ independent), lowers NAT costs from 6 → 3, and avoids inter-AZ traffic costs.
  1. **VPC Peering vs Transit Gateway 🡪 Your company has 10 VPCs across multiple accounts that need full-mesh connectivity. Initially you used VPC Peering, but now network management is too complex. What’s the better solution and why?**
* **VPC Peering**: Good for small setups, but it doesn’t support **transitive routing** (A↔B and B↔C doesn’t mean A↔C). With 10 VPCs, the number of peering connections grows as n(n-1)/2 (45 connections). Very hard to manage.
* **Transit Gateway (TGW)**:
* Acts as a **hub** to connect multiple VPCs and on-premises networks.
* Provides **transitive routing**, so all attached VPCs can talk via TGW without peering mesh.
* Centralized control and scalable up to **5,000 attachments**.
* Cost trade-off: TGW has data processing charges, but drastically simplifies management vs peering sprawl.
  1. **How Transit gateway works?**

**🔹 How does it work (at a high level)?**

1. **You create a Transit Gateway (TGW)** in a region.
   * It provides a **hub** for traffic to flow between attached networks.
2. **You attach networks to TGW** using *attachments*:
   * **VPC Attachments** → connect subnets from your VPCs.
   * **VPN Attachments** → connect to on-prem via IPSec VPN.
   * **Direct Connect Gateway Attachments** → for high-speed private on-prem connectivity.
   * **Peering Attachments** → to connect TGWs across regions.
3. **Subnets in each VPC attachment**:
   * You select **one subnet per AZ** for TGW to use as an ENI.
   * Traffic from/to TGW flows through those subnets.
4. **Routing**:
   * TGW has its own **route tables** (independent of VPC route tables).
   * You associate attachments with TGW route tables and configure **propagation** (automatic route sharing).
   * Example: VPC-A can reach VPC-B only if TGW’s route table allows it, and each VPC’s route table points to the TGW ENI.
5. **Traffic flow** (example):
   * Web in VPC-A → App in VPC-B.
   * Route table in VPC-A points to TGW → TGW route table points to VPC-B → packet is routed accordingly.
   1. **What is PrivateLink and how it works?**

**🔹 What is AWS PrivateLink?**

AWS **PrivateLink** lets you access **services across VPCs, AWS accounts, or AWS regions** **privately** over the **AWS backbone** — **without using public IPs, NAT, Internet Gateways, or VPC Peering**.

It’s implemented through **VPC Endpoints** (specifically **Interface Endpoints**) that connect to an **endpoint service**.

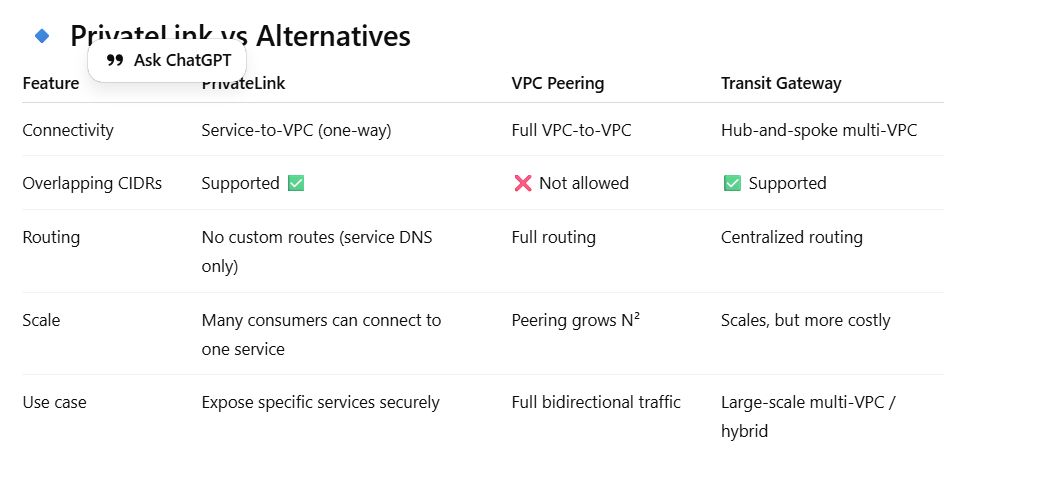
Think of it as **a private, one-way connection to a service, exposed via a Network Load Balancer (NLB)**.

**🔹 How does it work? (Step-by-step)**

1. **Service Owner side**:
   * Creates a service and exposes it using a **Network Load Balancer (NLB)** in their VPC.
   * Marks it as an **endpoint service** (can be shared with specific accounts or all AWS).
2. **Service Consumer side**:
   * Creates a **VPC Interface Endpoint** (ENIs in consumer subnets).
   * These ENIs map traffic to the service provider’s NLB — **via AWS’s private backbone**, not the internet.
3. **Routing**:
   * Consumers connect using a **DNS name** (AWS gives a private DNS name for the endpoint).
   * Their traffic goes through the **ENI → AWS backbone → Provider’s NLB → Service instances**.
4. **Security**:
   * Provider controls who can connect by approving endpoint connection requests.
   * No need to expose the service publicly.
   * Consumer and provider VPCs don’t need peering or overlapping CIDRs.

**🔹 Example Scenarios**

* ✅ **SaaS provider model**: A SaaS company exposes their API privately. Customers create a PrivateLink endpoint and consume the API **inside their own VPC** — without any internet exposure.
* ✅ **Shared services**: Central security VPC with a firewall or logging service — other VPCs use PrivateLink to connect.
* ✅ **RDS, S3, Secrets Manager, KMS** — AWS itself offers these services via PrivateLink endpoints.



**🔹 Step 1: Consumer creates an Interface VPC Endpoint**

* When you create a PrivateLink endpoint in your VPC, AWS automatically provisions **ENIs (Elastic Network Interfaces)** in your chosen subnets (one per AZ).
* Each ENI gets a **private IP** inside your subnet.
* These ENIs are managed by AWS, but they appear like “normal” NICs to your VPC.

**🔹 Step 2: DNS mapping to the endpoint ENIs**

* AWS gives you a **DNS name** for the service (e.g., vpce-1234567890abcdef.service.region.vpce.amazonaws.com).
* When your app resolves this DNS, it maps to the **private IP of the ENI** in your subnet.
* So from your EC2’s perspective, it’s just sending packets to another private IP in the same VPC.

**🔹 Step 3: Traffic enters the AWS backbone**

* Packets destined to that ENI are “caught” by AWS’s PrivateLink infrastructure.
* Instead of being delivered to an EC2 instance, those ENIs are **special AWS-managed endpoints** that tunnel traffic into the **AWS backbone**.
* No **Internet Gateway**, **NAT**, or **VPC peering** is involved.

**🔹 Step 4: Traffic delivered to the provider’s NLB**

* AWS’s internal PrivateLink fabric forwards the traffic to the **Network Load Balancer (NLB)** of the **provider’s VPC**.
* The NLB then distributes traffic to the registered service targets (EC2s, containers, etc.) in the provider’s subnets.

**🔹 Step 5: Response flows back**

* Since connections are **stateful**, the response traffic goes back from the provider → NLB → PrivateLink fabric → consumer’s ENI → consumer’s app.
* To the consumer, it looks like they’re just talking to a private IP inside their own subnet.
* To the provider, it looks like requests are coming from the **PrivateLink ENIs**, not directly from consumer EC2s (so logs show the endpoint connection IDs, not consumer’s original private IPs).

**🔹 Why it’s private (no Internet)**

* The ENIs are created in **consumer’s VPC** and have **only private IPs**.
* AWS routes packets directly from those ENIs to the provider’s NLB **inside AWS’s internal backbone network** — never touching the public internet.
* Unlike VPC peering or Transit Gateway, you don’t exchange CIDR routes. Instead, AWS internally “pins” the ENI to the service endpoint.

**🔹 Key differences vs Peering**

* With **peering**, you’re adding routes: 10.1.0.0/16 → peer. That exposes *whole subnets*.
* With **PrivateLink**, there are **no routes**. The ENI acts like a **private proxy** tied only to one service. You can’t send arbitrary traffic — only to that endpoint’s DNS.

✅ **In short:**  
When you hit the DNS of a PrivateLink service, AWS maps it to an **ENI in your subnet**. That ENI isn’t a “real NIC” but an AWS-managed proxy. It forwards traffic over AWS’s backbone to the provider’s NLB, then returns responses back to you. All stays private, no internet, no NAT, no peering.

* 1. Security Group Chaining 🡪 You have 50 microservices, each in its own SG. You don’t want to manually update every SG when services need to talk. How can you simplify?

**🔹 Problem Recap**

* You have **50 microservices**, each deployed in **its own Security Group (SG)**.
* These services often need to communicate (for example, microservice A → B → C → D).
* **Manually adding 49 inbound rules per SG** (to allow all-to-all or selective traffic) is unmanageable, error-prone, and doesn’t scale.

**🔹 Goal**

Simplify or automate how SGs reference each other while keeping least privilege and avoiding manual updates.

**✅ Solution 1: Use Security Group references dynamically**

AWS allows you to **reference another SG ID** inside a rule instead of IPs or CIDRs.

So, you can create:

* A **shared “microservices-common” SG**, and
* Add it as a **source SG** in inbound rules for all microservice SGs.

Then attach this “common” SG to each service **in addition** to its own SG.

**Example:**

* SG-microservices-common: inbound rule allows traffic from itself (source = SG-microservices-common).
* Each microservice instance has:
  + SG-serviceX (specific rules)
  + plus SG-microservices-common (shared communication rule).

**Result:**  
All instances sharing this SG can talk to each other on allowed ports automatically — you don’t need to touch 50 SGs every time a new one appears.

✅ **Benefits**

* Scales easily — just attach the shared SG to a new service.
* Traffic stays private and restricted to AWS resources using that SG.
* No need to manage individual CIDRs.

⚠️ **Caution**

* Avoid over-permissive “all ports” rules; limit to known app ports (e.g., 8080, 443).
* Don’t use a single SG for *everything* if you have distinct trust boundaries.

**✅ Solution 2: Use SG-to-SG referencing pattern (grouping by role)**

Instead of one SG per service, group by **service role or trust level**:

| **Role** | **Example SG** | **Allows from** |
| --- | --- | --- |
| Frontend | SG-frontend | From ALB |
| Backend APIs | SG-backend | From SG-frontend |
| Database | SG-database | From SG-backend |

This creates a **tiered model** instead of a fully meshed network.

✅ Easier to reason about.  
✅ Enforces principle of least privilege.  
✅ Works with auto scaling.

* 1. **What is AWS Direct Connect and why use it instead of a VPN?**

**🔹 What is AWS Direct Connect (DX)?**

**AWS Direct Connect** is a **dedicated, private network connection** between your **on-premises data center (or colocation facility)** and **AWS**.

Instead of sending your traffic over the public internet (like a VPN does), Direct Connect provides a **dedicated fiber link** into the AWS network backbone.

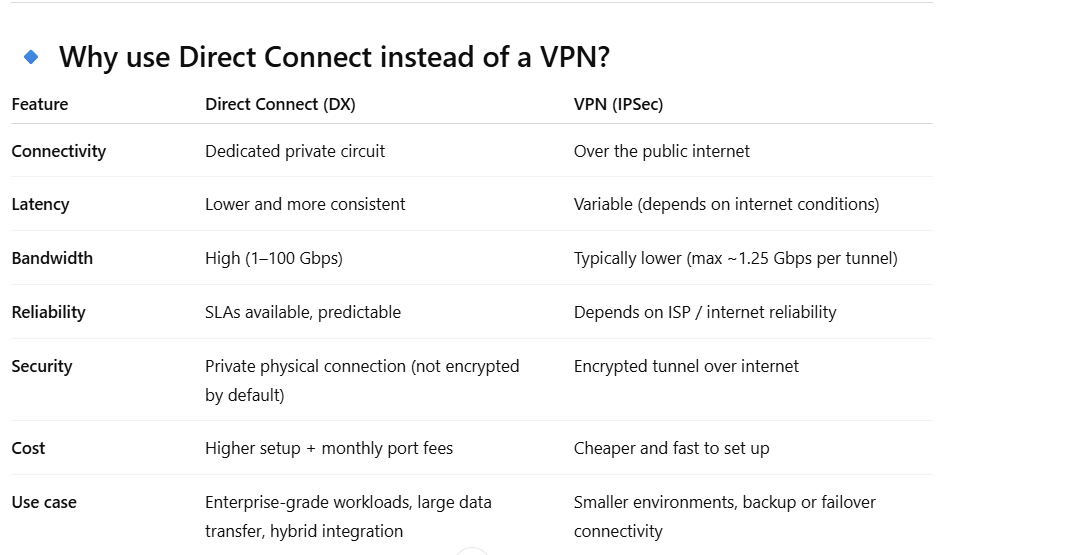
So — it’s essentially a **private WAN link** into your AWS VPCs or AWS services.

**🔹 How it Works (High-level)**

1. You order a **Direct Connect connection** from AWS (usually 1 Gbps, 10 Gbps, or 100 Gbps ports).
2. AWS provisions this at a **Direct Connect location** (colocation data center).
3. You either:
   * **Co-locate** your router at that location, or
   * **Use a partner carrier** (AWS Direct Connect Partner) to extend the connection from that DX location to your on-prem data center.
4. You create a **Virtual Interface (VIF)** on the DX connection:
   * **Private VIF** → connects to your VPC via a Virtual Private Gateway (VGW) or Transit Gateway.
   * **Public VIF** → connects to AWS public services (S3, DynamoDB, etc.) privately.
   * **Transit VIF** → connects multiple VPCs via a Transit Gateway.

Once configured, your on-prem router peers with AWS using **BGP** (Border Gateway Protocol) to exchange routes.

✅ **Architecture:**  
On-prem → DX Router → DX Location → DXGW → VGW/TGW → VPC



* 1. Scenario: You need to connect multiple AWS accounts and VPCs from your data center using Direct Connect. How do you design it?

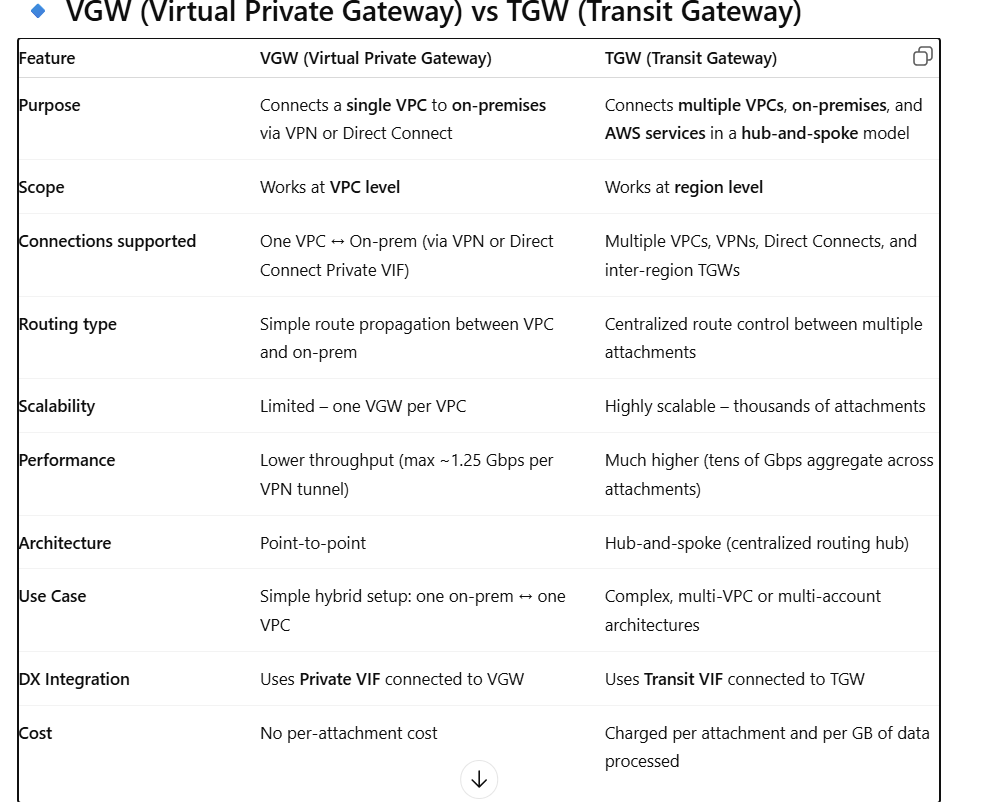
Use **Direct Connect Gateway (DXGW)** with **Transit Gateway (TGW)**:

* Create **one DX connection** to AWS.
* Attach that DX to a **DX Gateway**.
* Associate multiple **Transit Gateways** (across accounts) with the DXGW.
* Each TGW connects to multiple VPCs.

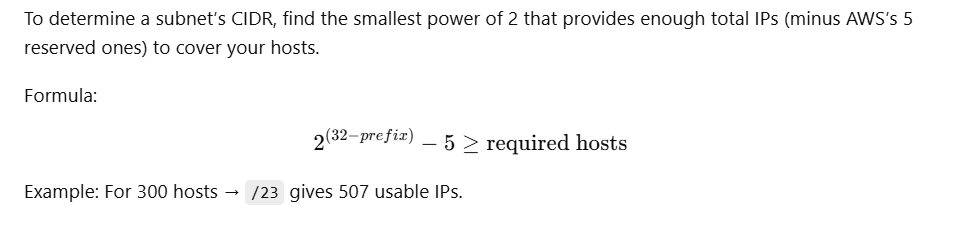
✅ **Result:**  
Single DX → multiple TGWs → multiple VPCs (multi-account, multi-region).

This avoids having to create separate DX connections per account or VPC.

* 1. Difference Between VGW and TGW?



* 1. Find Subnet CIDR that cover 300 hosts.

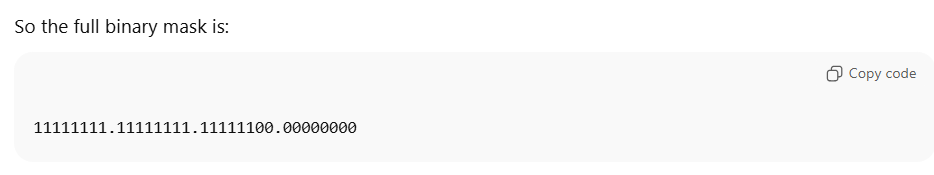


* 1. Find the CIDR notation for subner mask 255.255.252.0

**🔹 Step 1: Write the subnet mask in binary**

Each octet is 8 bits:

| **Decimal** | **Binary** |
| --- | --- |
| 255 | 11111111 |
| 255 | 11111111 |
| 252 | 11111100 |
| 0 | 00000000 |



**🔹 Step 2: Count the number of 1 bits**

Each **1** = a **network bit**.  
Each **0** = a **host bit**.

11111111 → 8 bits

11111111 → 8 bits

11111100 → 6 bits

00000000 → 0 bits

-----------------

Total = 8 + 8 + 6 + 0 = 22 bits

✅ Therefore:  
**CIDR = /22**

A white background with black text

AI-generated content may be incorrect.

**🔹 Step 4: Calculate usable hosts**

**In normal networking:**

Subtract 2 (network + broadcast):

1024−2=1022 usable hosts1024 - 2 = 1022 \text{ usable hosts}1024−2=1022 usable hosts

**In AWS:**

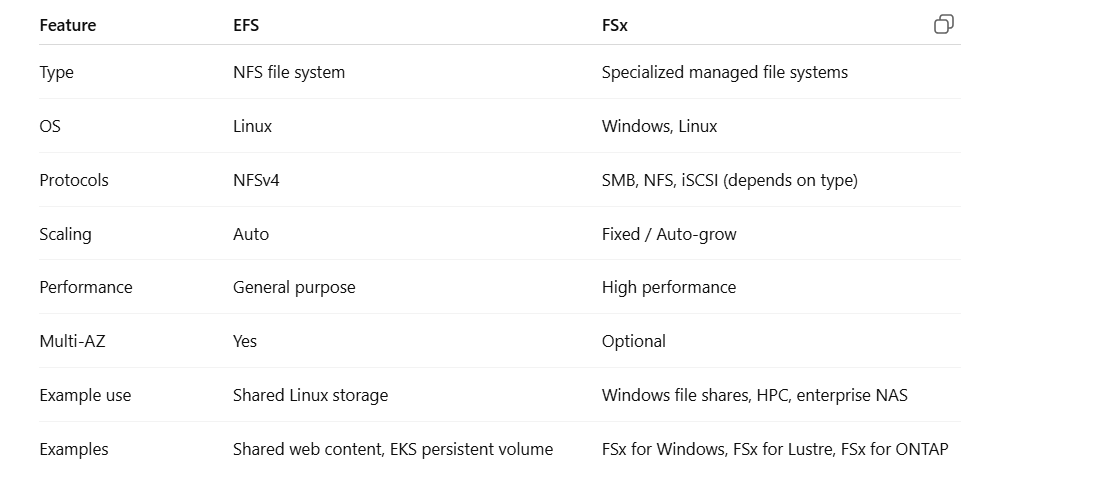
AWS reserves 5 addresses per subnet:

1024−5=1019 usable hosts1024 - 5 = 1019 \text{ usable hosts}1024−5=1019 usable hosts

1. **AWS storage ( RDS, DynamoDb, Elasticache, s3 ,etc)**
   1. What is the difference between EFS and FSx.?

**Amazon EFS** is a **fully managed NFS file system** for Linux workloads — simple, scalable, and regional.  
It’s ideal for **shared storage** across EC2, ECS, or EKS instances.

**Amazon FSx** is a **family of managed file systems** for specialized use cases — such as **Windows file shares (SMB)**, **HPC workloads (Lustre)**, or **enterprise storage (NetApp ONTAP)**.



* 1. **RDS Multi-AZ vs Read Replica — When and Why to Use Each?**

**You’re running an RDS MySQL database. You want high availability and read scalability. What’s the difference between using a Multi-AZ setup vs Read Replica, and can you use both together?**

* Multi-AZ Deployment:
* Provides high availability (HA) — AWS automatically replicates data synchronously to a standby in another AZ.
* Failover is automatic during maintenance or primary failure.
* Not used for read scaling (standby is passive).
* Read Replica:
* Used for read scalability and offloading reporting/analytics.
* Asynchronous replication (eventual consistency).
* No automatic failover (unless promoted).
  1. **RDS — Performance and Cost Optimization Scenario 🡪 Your production RDS PostgreSQL instance faces high IOPS usage and frequent performance issues. How would you troubleshoot and optimize it without increasing the instance class immediately?**
* **Step 1: Identify the bottleneck**
  + **Use Enhanced Monitoring, Performance Insights, and CloudWatch metrics (ReadIOPS, WriteIOPS, FreeableMemory, CPUUtilization).**
* **Step 2: Optimize before scaling**
  + **Enable Query Caching via ElastiCache.**
  + **Add Read Replicas for read-heavy workloads.**
  + **Use Provisioned IOPS (io2) if consistent performance is needed.**
  + **Optimize schema & indexes.**
  + **Enable RDS storage autoscaling (to avoid hitting IOPS limits).**
  + **Offload analytics/reporting to a replica or Athena (if S3 integrated).**
* **Step 3: Use Storage Type Change (GP3 vs IO2):**
  + **GP3 offers separate IOPS provisioning (cheaper than IO2 for same performance).**

**✅ Cost Optimization Tip: Move infrequently accessed data to S3 via RDS Snapshot + Export.**

* 1. **RDS Disaster Recovery Scenario 🡪 You have RDS in ap-south-1 (Mumbai). You need cross-region DR with minimal data loss. How do you design it?**
* Use Cross-Region Read Replica:
* Replicates data asynchronously from Mumbai to Singapore region.
* Promote the replica to standalone DB during disaster.
* Use Automated Backups + Point-in-Time Recovery (PITR) for RPO < 5 minutes.
* Optionally, use Aurora Global Database, which provides <1 second lag and sub-second failover across regions.
  1. **What is the usage of Elasticache Redis?**

**Amazon ElastiCache for Redis is a fully managed, in-memory data store and cache service.**

**It’s used to make your applications faster, more scalable, and cost-efficient by reducing database load and latency.**

**A screenshot of a computer

AI-generated content may be incorrect.**

****

* 1. **S3 Lifecycle and Cost Optimization 🡪 Your logs stored in S3 are 100 TB and growing daily. How would you optimize cost while maintaining access to 30-day-old data?**
* Use Lifecycle Policies:
  + 0–30 days: S3 Standard.
  + 30–90 days: Transition to S3 Standard-IA.
  + >90 days: Transition to S3 Glacier Instant Retrieval or S3 Glacier Deep Archive (lowest cost).
* Use Intelligent-Tiering if access patterns are unpredictable.
* Optionally enable S3 Object Lock for compliance.

✅ This automates tiering and cuts storage costs up to 70–90%.

* 1. **S3 Security and Access Control 🡪 Your DevOps team accidentally made an S3 bucket public. How do you prevent this organization-wide?**

**🔹 Step-by-Step Solution**

**🛑 Use S3 Block Public Access (BPA) — Account & Org Level**

**🔸 What it does:**

**S3 Block Public Access** is the **strongest guardrail** to prevent buckets or objects from being made public — **even if someone tries to add a public ACL or policy**.

**🔸 Where to apply:**

* You can enable it at:
  + **Bucket level**
  + **Account level**
  + **Organization level (via AWS Organizations + SCPs or Service Control Policies)**

**🔸 Key Settings:**

| **Setting** | **Description** |
| --- | --- |
| BlockPublicAcls | Prevents new public ACLs |
| IgnorePublicAcls | Ignores existing public ACLs |
| BlockPublicPolicy | Prevents bucket policies granting public access |
| RestrictPublicBuckets | Restricts access to buckets with public policies to AWS accounts only |

**🏛️ 2️⃣ Use Service Control Policies (SCPs) at Org Level**

If you manage multiple AWS accounts under **AWS Organizations**, you can apply an **SCP** to prevent users from disabling S3 Block Public Access.



**🔒 3️⃣ Use AWS Config Rules to Detect Violations**

Use **AWS Config Managed Rules** like:

* s3-bucket-public-read-prohibited
* s3-bucket-public-write-prohibited

These continuously **monitor all buckets** and alert if any bucket becomes public.

You can integrate these with:

* **SNS** for alerts
* **Lambda** for auto-remediation (e.g., revert public access)
* **Security Hub** for centralized compliance tracking

**⚙️ 4️⃣ Optional — Use IAM Policies to Restrict S3 Changes**

Add IAM policies for developers preventing public access configuration:

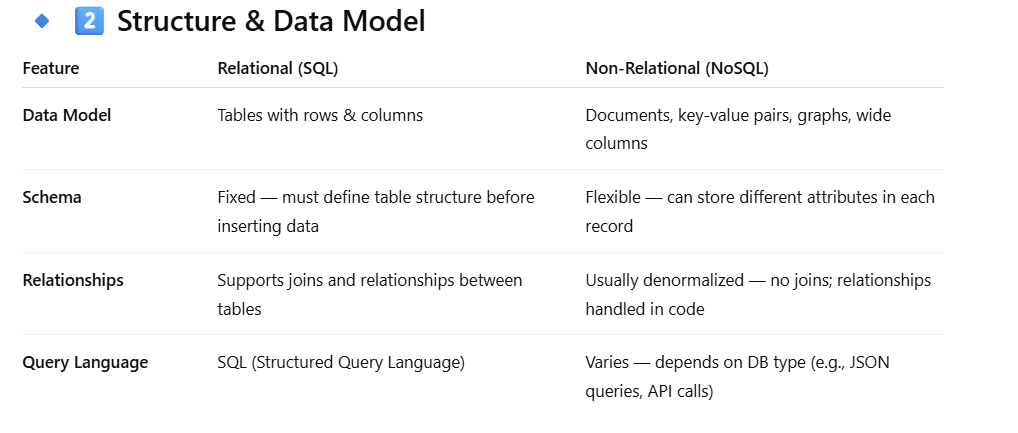
A computer code on a white background

AI-generated content may be incorrect.

* 1. **What is the difference between relational and non relational databases?**

**🔹 1️⃣ Overview**

| **Type** | **Definition** |
| --- | --- |
| **Relational Database (RDBMS)** | **Stores data in tables (rows and columns) with relationships between them, using structured schema (SQL-based).** |
| **Non-Relational Database (NoSQL)** | **Stores data in flexible, unstructured formats like key-value, document, graph, or columnar, without a fixed schema.** |

****

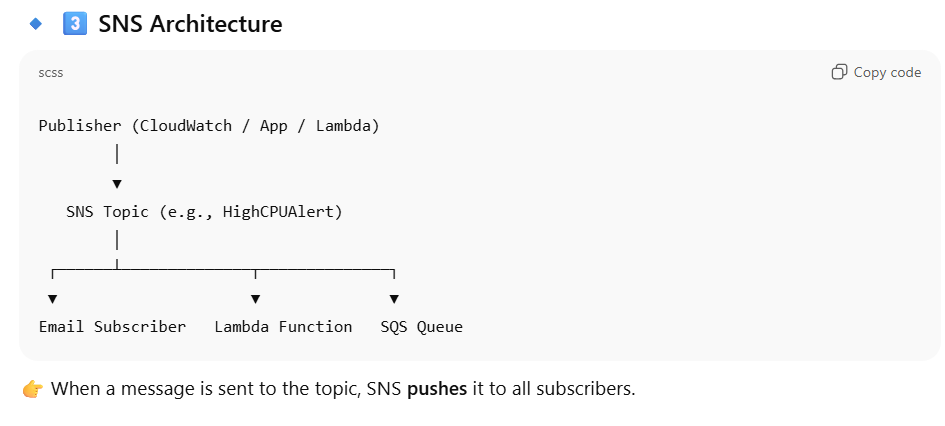
**🔹 5️⃣ Performance & Scaling**

| **Aspect** | **Relational (SQL)** | **Non-Relational (NoSQL)** |
| --- | --- | --- |
| **Scaling** | **Vertical (increase instance size: CPU, RAM)** | **Horizontal (add more nodes/shards)** |
| **Consistency** | **Strong (ACID)** | **Often eventual (BASE)** |
| **Transactions** | **Fully supported (ACID-compliant)** | **Limited or per-partition basis** |
| **Speed** | **Great for structured queries, but joins can be expensive** | **Fast for simple key-based lookups and high-volume workloads** |

1. **SNS and SQS**
   1. **What is SNS and how it works ?**

**Amazon SNS (Simple Notification Service) is a fully managed pub/sub (publish–subscribe) messaging service that allows you to send notifications or messages from one publisher to many subscribers — instantly and reliably.**

**It helps you decouple microservices, fan out messages, and send alerts or event notifications across systems.**

****

**🔹 5️⃣ How It Works (Step-by-Step)**

1. **Create a Topic**
   * **Example: arn:aws:sns:us-east-1:123456789012:HighCPUAlert**
2. **Add Subscribers**
   * **Add email, SQS queue, or Lambda function as subscribers.**
3. **Publish a Message**
   * **From CloudWatch, CLI, Lambda, or application.**
4. **SNS Delivers Message**
   * **SNS fans out the message to all subscribers immediately.**
5. **Subscribers Act**
   * **Emails get alerts.**
   * **SQS queues store messages for later processing.**
   * **Lambda runs code automatically.**
   1. **What is SQS and how it works?**

**Amazon SQS (Simple Queue Service) is a fully managed message queuing service that enables asynchronous communication between distributed systems, microservices, or serverless applications.**

**It allows one system (the producer) to send messages, and another (the consumer) to process them later, reliably, and at its own pace — decoupling the two systems.**

**🔹 2️⃣ SQS Basic Concept**

**Producer → Queue → Consumer**

* **Producer: Application or service that sends a message.**
* **Queue: A temporary storage for messages.**
* **Consumer: Application, Lambda, or service that retrieves and processes the message.**

**🔹 4️⃣ How SQS Works (Step-by-Step)**

**Producer → SQS Queue → Consumer**

**Step 1️⃣: Producer sends a message**

* **Your app, Lambda, or AWS service sends a message to the queue.**
* **The message is durably stored in multiple AZs.**

**Step 2️⃣: Message stays in the queue**

* **Messages remain until a consumer retrieves them (up to 14 days retention).**

**Step 3️⃣: Consumer polls the queue**

* **Consumers (EC2, ECS, Lambda, etc.) poll the queue to get messages.**
* **Once a message is received, it becomes invisible for a certain period (Visibility Timeout).**

**Step 4️⃣: Consumer processes and deletes message**

* **If successful → consumer deletes the message.**
* **If failure → message becomes visible again after timeout for reprocessing.**

**✅ This ensures no message loss, even if a consumer crashes mid-processing.**

* 1. **key terms.**
* **What is Long Polling in SQS?**

By default, when you poll (receive messages) from an SQS queue, the ReceiveMessage API immediately returns — even if there are no messages available.  
This is known as Short Polling.

Long Polling, on the other hand, allows the poll request to wait (up to 20 seconds) until a message arrives in the queue before returning an empty response.

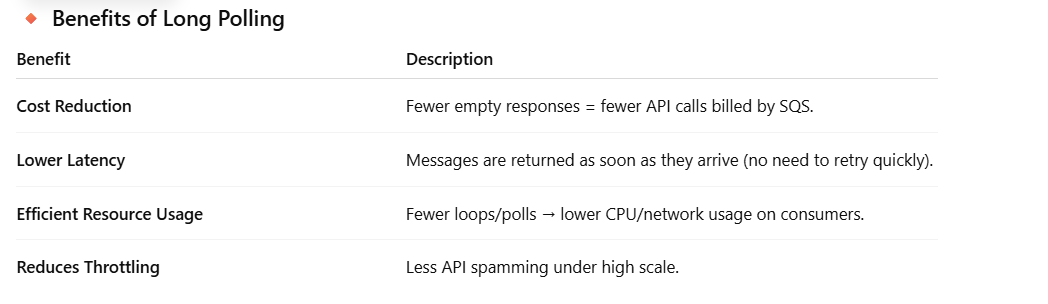
**🔸 How It Works**

**When you call:**

**ReceiveMessage --queue-url <url> --wait-time-seconds <value>**

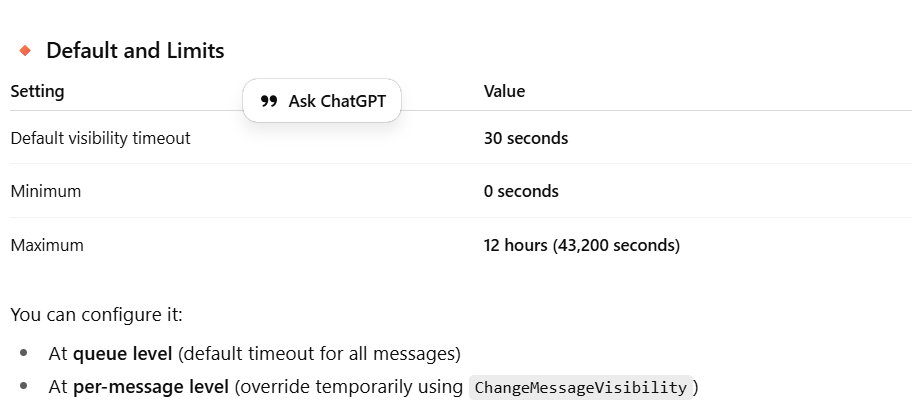
* **If the queue already has messages → response is immediate.**
* **If the queue is empty → SQS holds the connection open for up to the specified WaitTimeSeconds (max 20s).**
* **If a message arrives within that time → SQS immediately returns it.**
* **If no message arrives → SQS returns an empty response after waiting.**

**✅ This is Long Polling.**

****

* **What is Visibility Timeout in Amazon SQS?**

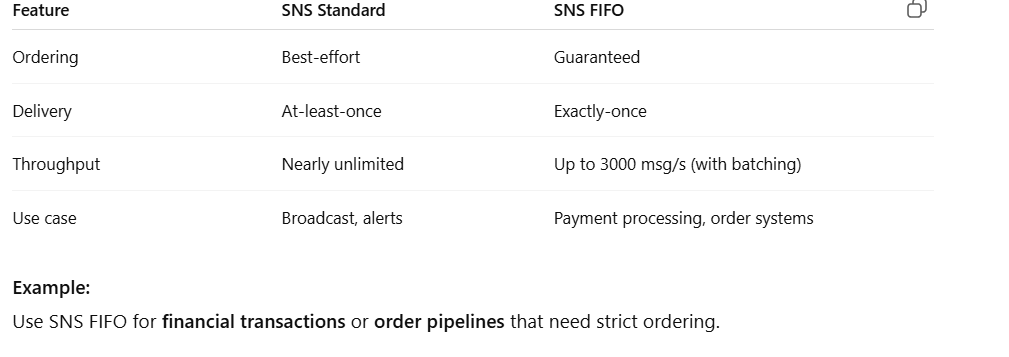
When a consumer retrieves a message from an SQS queue, that message **is not deleted immediately**.  
Instead, it becomes **invisible** to other consumers for a specific period — called the **Visibility Timeout**.



A screenshot of a white box

AI-generated content may be incorrect.

* 1. What’s the difference between SNS FIFO topics and Standard topics?



7.5 How can you scale consumers dynamically based on SQS queue depth?

 Use **CloudWatch metrics**:

* ApproximateNumberOfMessagesVisible (backlog)
* ApproximateAgeOfOldestMessage (latency)

 Integrate with:

* **Application Auto Scaling** (for EC2/containers)
* **Lambda concurrency scaling**

 **Example:**

* If messages > 1000 → scale out ECS tasks by 2x.
  1. **Combined SNS + SQS Scenario Question**

**Scenario:**

**You have a distributed system where:**

* **Microservices publish events to SNS.**
* **Multiple consumers subscribe via SQS queues.**
* **You notice consumers processing the same message twice.**

**How would you debug and fix this?**

** Check SNS delivery retries – duplicate deliveries possible on transient network errors.**

** Check SQS visibility timeout – too short → message reappears → double processing.**

** Enable FIFO topics and queues for exactly-once semantics.**

** Implement idempotent consumers – e.g., track processed message IDs in DynamoDB or Redi**