AWS Elastic Beanstalk

Question: ***How does Elastic Beanstalk differ from ECS/EKS and Lambda in terms of abstraction and use cases?***

- **Elastic Beanstalk**
  - **Abstraction Level**: **High-level PaaS** (Platform as a Service).
  - **What it does**: You just provide your code (Java, .NET, Node.js, Python, PHP, Go, Docker, etc.), and Beanstalk handles provisioning of EC2, Load Balancers, Auto Scaling, Monitoring, and Deployment.
  - **Control**: Limited control over infrastructure
- **Amazon ECS / Amazon EKS**
  - **Abstraction Level**: **Container orchestration** (lower-level than Beanstalk).
  - **What it does**:
    - **ECS (Elastic Container Service)** → AWS's native container orchestrator.
    - **EKS (Elastic Kubernetes Service)** → Managed Kubernetes on AWS.
  - **Control**: You define how containers run, network configs, scaling policies, IAM roles, etc. You manage the application architecture more explicitly.
- **AWS Lambda**
  - **Abstraction Level**: **Highest-level FaaS** (Function as a Service).
  - **What it does**: Run code functions (Python, Node.js, Java, C#, Go, etc.) **without servers**. AWS manages everything — scaling, availability, fault tolerance. You pay only for execution time.
  - **Control**: No servers or containers to manage — but you are limited by runtime, execution duration (max ~15 mins), memory, and cold start issues.
- **EC2 (Elastic Compute Cloud) – IaaS (Infrastructure as a Service)**
  - **What it is**: Virtual machines you manage (OS, scaling, patching).

Question : ***What happens behind the scenes when you deploy an application to Elastic Beanstalk?***

when you deploy an application to **Elastic Beanstalk**, a lot happens behind the scenes. It feels simple to the developer ("upload code, get running app"), but internally, Beanstalk orchestrates multiple AWS services for you.

- **1. Environment Creation**

When you create a Beanstalk environment (e.g., Web Server environment):

- **Elastic Beanstalk Service** provisions AWS resources such as:
    - **EC2 instances** (compute)
    - **Auto Scaling Group** (for elasticity)
    - **Elastic Load Balancer (ALB/CLB/NLB)** (for traffic distribution)
    - **Security Groups** (network security)
    - **CloudWatch Alarms & Logs** (monitoring)
    - **S3 bucket** (to store your app versions/bundles)
    - **IAM roles** (permissions for EC2 and Beanstalk itself)

🔹 **2. Application Version Upload**

- Your application bundle (ZIP or WAR file, or Docker image reference) is uploaded to an **S3 bucket** that Beanstalk manages.
- Beanstalk creates a new **Application Version** entry that points to this artifact in S3.

🔹 **3. Environment Configuration**

- Beanstalk applies any settings you defined:
    - Environment variables
    - Scaling configuration (min/max instances, desired count)
    - Load balancer rules
    - Health check URLs

🔹 **4. Health Monitoring**

- Beanstalk continuously monitors environment health:
    - Checks load balancer health checks.
    - Uses CloudWatch metrics (CPU, memory, latency, 5xx errors).
    - Reports health as **Green, Yellow, Red, or Grey** in the console.

🔹 **5. Scaling & Updates**

- If traffic increases, the **Auto Scaling Group** launches new EC2 instances.
- During updates, Beanstalk can use deployment policies:
    1. **All at once**
    2. **Rolling**
    3. **Rolling with additional batch**
    4. **Immutable** (creates new instances before replacing old ones)

Feature	.ebextensions	.platform
Format	YAML/JSON config	Shell scripts
Scope	Environment (infra & instance setup)	Platform & runtime (OS, server, hooks)
When executed	During provisioning (CloudFormation phase)	During deployment lifecycle

Question : ***What's the difference between a single-instance environment and a load-balanced environment?***

- **Single-Instance Environment**
  - **Infrastructure**:
    - 1 EC2 instance only.
    - No load balancer, no auto scaling group.
  - **Use Case**:
    - Development, testing, proof-of-concept.
    - Low-traffic apps or internal tools.
    - Cost-sensitive environments (cheaper).
- **Load-Balanced Environment**
  - **Infrastructure**:
    - Elastic Load Balancer (ALB/NLB/CLB).
    - Auto Scaling Group managing multiple EC2 instances.
    - Scaling policies (min/max/desired capacity).
  - **Use Case**:
    - Production workloads.
    - Applications needing **high availability** and **scalability**.
    - Apps expecting variable or high traffic.

Question : **Can you configure target tracking scaling in Elastic Beanstalk?**

How Scaling Works in Elastic Beanstalk

- Every **Load-Balanced Beanstalk environment** has an **Auto Scaling Group (ASG)** created by Beanstalk.
- By default, Beanstalk uses **simple scaling policies** (like scaling based on average CPU utilization thresholds you set in environment configuration).
- However, Beanstalk also supports **target tracking scaling** (the newer, more intelligent scaling mode in ASG).

Question : ***How would you customize the load balancer in an EB environment***

- Use the **EB Console settings** for standard options (SSL, listeners, stickiness, draining).
- Use **.ebextensions configs** to extend the CloudFormation stack and add custom LB rules, listeners, or security settings.
- Use **.platform configs** for reverse proxy customization if it's app-specific.

Question : ***What strategies can you use to optimize cold start times when deploying EB applications?***

To optimize **Elastic Beanstalk cold start times**:

- Use **pre-baked AMIs** with dependencies pre-installed.
- Keep **startup scripts light** (.ebextensions / .platform).
- Optimize **app startup** (lazy-load, cache, precompile assets).
- Use **rolling/immutable deployments** instead of all-at-once.
- Tune **health checks & connection draining**.
- Keep **at least 1 instance running** to avoid scaling-from-zero delays

Pre-baked AMIs = custom **Amazon Machine Images** you create with your application's **dependencies, packages, and configurations already installed**.

✅ **Summary (Interview-Ready)**

- **Rolling** = update in **batches** → cost-efficient, but may serve mixed versions.
- **Immutable** = spin up a **new fleet**, then cut over → safer, zero-downtime, but costlier.

Question : **How can you use worker environments + SQS for asynchronous processing?**

- Elastic Beanstalk has **two environment tiers**:
    1. **Web server environment** (front-end, with ALB/ASG/EC2).
    2. **Worker environment** (background processing, with SQS integration).
- In a **Worker Environment**:
    - AWS automatically provisions an **SQS queue**.
    - EB also provisions EC2 instances that **poll the SQS queue**.
    - Messages from SQS are delivered as **HTTP POST requests** to your worker app running on EC2.

Question : ***Explain the different deployment policies in Elastic Beanstalk (All-at-once, Rolling, Rolling with Additional Batch, Immutable, Blue/Green)***

- **1. All-at-Once**
  - **How it works**:
    - Deploys the new version to **all instances simultaneously**.
    - Old app is stopped, new one started.
- **2. Rolling**
  - **How it works**:
    - Updates a **batch of instances at a time** (e.g., 25%).
    - Waits for them to pass health checks before moving to next batch.
- **3. Rolling with Additional Batch**
  - **How it works**:
    - Beanstalk launches a **new temporary batch** of instances.
    - Shifts traffic to them → updates old instances batch by batch.
    - After rollout, temporary batch is terminated.
- **4. Immutable**
  - **How it works**:
    - Launches a **whole new set of instances** with the new version.
    - Once healthy, reroutes traffic via Load Balancer.
    - Old instances are terminated afterwards.

Question **: *Blue/Green Deployment (Best Practice for Mission-Critical Apps)***

- Create a **separate "green" environment** with the new version.
- Test it thoroughly while your "blue" environment is still serving traffic.
- When ready, perform a **CNAME swap** (or Route 53 DNS switch) → instantly routes traffic to green.
- Keep blue as backup for quick rollback.

Question : ***To set up a multi-container Docker environment in Elastic Beanstalk:***

- Use **Dockerrun.aws.json v2** → defines container images, ports, volumes (EB converts it to an ECS task definition).
- Use **.ebextensions/** → environment settings (instance type, env vars, scaling).
- Use **.platform/** → deeper instance/OS customization.
- EB runs all containers on EC2 via ECS, behind an ALB.
- Considerations: **load balancing, scaling, logging (CloudWatch), networking, persistence (S3/EFS/RDS)**

EC2

Question: ***What is Amazon EC2, and how does it work?***

🔹 **Amazon EC2 (Elastic Compute Cloud)**

Amazon EC2 is a **core AWS service** that provides **resizable virtual servers in the cloud** (called **instances**).
It lets you run applications without having to buy and maintain physical servers.

🔹 **How it Works**

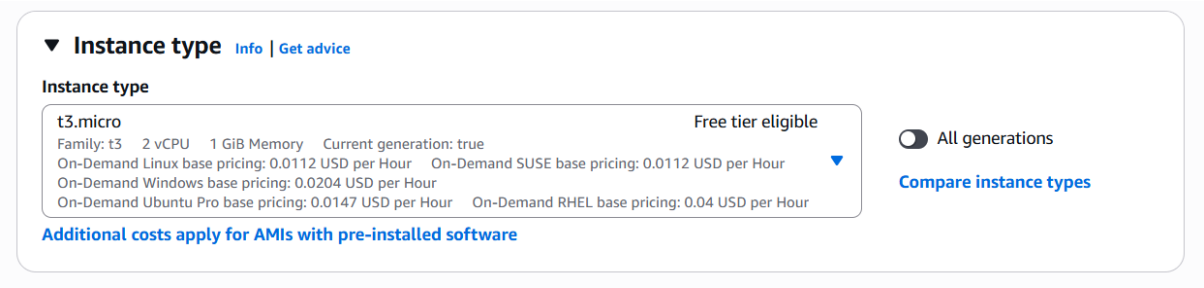
1. **Choose an AMI (Amazon Machine Image)**
    - An AMI is a preconfigured template (OS + software). Example: Ubuntu, Windows Server, Amazon Linux.
2. **Choose Instance Type**
    - Defines the hardware specs (CPU, RAM, networking, storage). Example: t3.micro, m5.large.
3. **Networking Setup**
    - Place the instance inside a **VPC subnet**, assign security groups (firewall rules), and optionally Elastic IPs.
4. **Storage**
    - Attach **EBS volumes** (block storage) or use **instance store**.
5. **Launch & Access**
    - When launched, EC2 runs on AWS's physical hardware as a **virtual machine**.
    - You connect via **SSH (Linux)** or **RDP (Windows)**.
6. **Scaling**
    - You can launch one or thousands of instances.
    - Integrated with **Auto Scaling** and **Load Balancing** for elasticity.

Question: ***Difference between On-Demand, Reserved, Spot, and Dedicated Hosts?***

- **On-Demand** → pay-as-you-go, flexible but costly.
- **Reserved** → commit (1–3 yrs), cheaper for steady workloads.
- **Spot** → unused capacity, very cheap but can be terminated anytime.
- **Dedicated Hosts** → physical server for compliance/licensing.

Question: ***What are EC2 instance types (General Purpose, Compute Optimized, Memory Optimized, Storage Optimized, Accelerated Computing)?***

- **General Purpose** → balanced (web apps, dev/test).
- **Compute Optimized** → CPU-heavy (batch processing, gaming).
- **Memory Optimized** → RAM-heavy (databases, analytics).
- **Storage Optimized** → high IOPS/throughput (data warehousing, NoSQL).
- **Accelerated Computing** → GPUs/FPGA for ML, AI, HPC.



Question: ***What's the difference between EBS volumes and Instance Store volumes?***

- **Amazon EBS (Elastic Block Store)**
  - **Network-attached** block storage.
  - **Persistent**: data survives instance stop/start/terminate (if volume not deleted).
  - Can be **detached and reattached** to different EC2 instances.
  - Supports **snapshots** (backups to S3).
- **Instance Store (Ephemeral Storage)**
  - **Physical disks** attached directly to the host machine.
  - **Non-persistent**: data is lost if the instance stops, terminates, or fails.
  - **High IOPS** and low latency (faster than EBS in some cases).
  - Cannot be detached or backed up like EBS.

Question: ***How do you stop, start, reboot, and terminate EC2 instances?***

- **Stop** → shuts down, can restart later, EBS persists.
- **Start** → boots a stopped instance.
- **Reboot** → restart, instance keeps running state, IP unchanged.
- **Terminate** → permanent deletion, volumes/data lost unless kept.

Question: ***What is an Elastic IP and how is it different from a public IP?***

- **Public IP**
  - **Auto-assigned** by AWS when you launch an EC2 in a public subnet.
  - **Dynamic** → changes every time you stop/start the instance.
- **Elastic IP (EIP)**
  - A **static public IPv4 address** allocated to your AWS account.
  - You can **attach/detach/reassign** it to any EC2 instance in your VPC.

Question: ***What's the difference between Security Groups and Network ACLs for EC2?***

- **Security Groups (SGs)**
  - **Instance-level firewall** (attached to ENI of an EC2).
  - **Stateful** → if inbound is allowed, outbound response is automatically allowed.
  - Rules are based on **allow only** (no explicit deny).
- **Network ACLs (NACLs)**
  - **Subnet-level firewall** (applies to all resources in subnet).
  - **Stateless** → inbound and outbound rules are evaluated separately.
  - Supports both **allow and deny** rules.

Question: ***How do you connect to an EC2 instance in a private subnet?***

You **cannot connect directly** to an EC2 in a private subnet. Instead, use a **bastion host in a public subnet**, **AWS Systems Manager Session Manager**, or a **VPN/Direct Connect** to securely access it.

Question: ***How does Elastic Network Interface (ENI) work, and why would you use multiple ENIs?***

An **Elastic network interface(ENI)** is a virtual network interface that acts as a **point of interface** between VM and network by attaching a public IP , private IP , security groups and many more to you instance

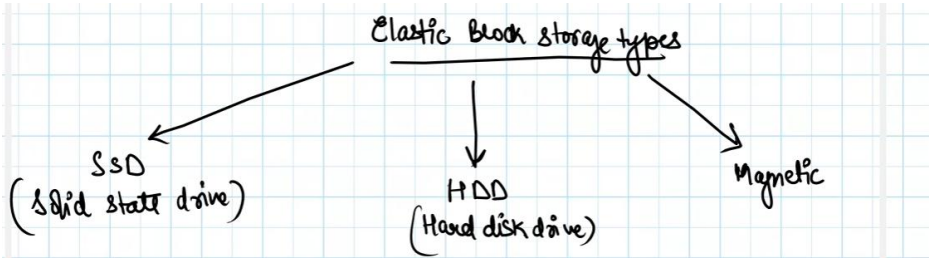
Attribute in Elastic network interface

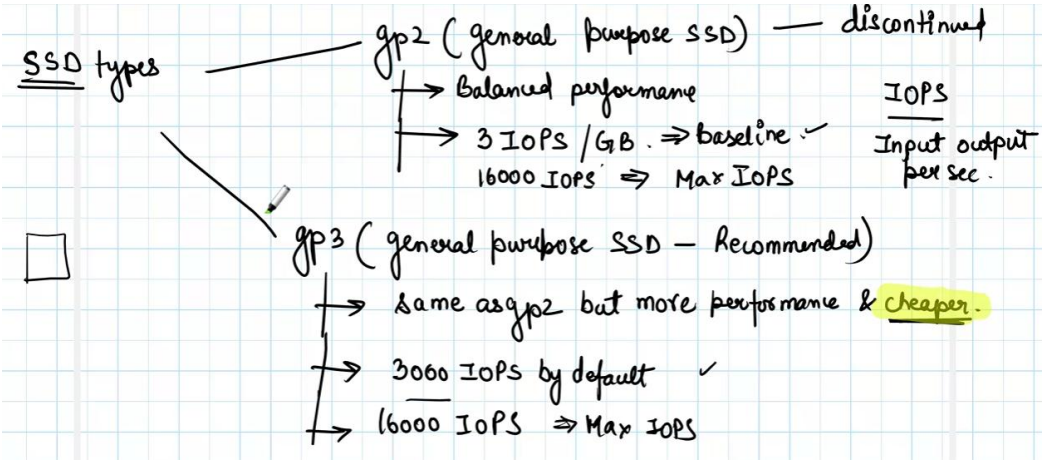
- Primary private IPv4 address
- Secondary private IPv4 address
- Elastic IP
- IPv6 Address
- Security Group
- Source/ Destination check flag

Question: ***Difference between EBS, EFS, and S3 with respect to EC2?***

Question: ***What is EBS snapshot, and how does it work?***

Question: **What's the difference between EBS volume types (gp3, io2, st1, sc1)?**





Question: ***What happens if the root EBS volume is deleted when terminating an instance?***

Question: ***How do you migrate data from one EC2 instance to another?***

- Direct File Transfer (for small/medium workloads)
- Create an AMI & Launch New Instance
- Detach & Reattach EBS Volumes
- EBS Snapshot → New Volume

Question: ***How do you migrate an on-prem VM to EC2 (lift-and-shift)?***

To migrate an on-prem VM to EC2, you typically use **AWS Application Migration Service (MGN)**, which replicates your VM to AWS and lets you launch it as an EC2 instance with minimal downtime. Alternatively, you can use **VM Import/Export** to create an AMI from the VM. After testing, cut over and switch DNS/traffic to the new EC2 instance.

Load Balancer

Question: ***How would you choose between ALB, NLB, and Gateway Load Balancer in a microservices setup?***

In a microservices setup, I'd typically use **ALB for routing HTTP/HTTPS traffic** to microservices based on paths or hostnames. If a microservice needs **low-latency TCP/UDP communication or must handle millions of requests per second, I'd choose NLB**. For cases where I need to **insert security appliances like firewalls or intrusion detection inline, I'd use Gateway Load Balancer**. Often, these are combined — for example, ALB for external traffic, NLB for internal service-to-service calls, and GWLB for centralized security.

Question: **Can you place a load balancer in a private subnet? What are the use cases?**

Yes, you can place a load balancer in a private subnet, but it will be an **internal load balancer** accessible only inside the VPC or through VPN/Direct Connect. Common use cases are **internal microservice communication**, **hybrid connectivity with on-prem**, **internal APIs/databases**, and **multi-tier architectures** where only the frontend is exposed publicly.

Question: ***How would you implement multi-region active-active architecture using load balancers and Route 53?***

I'd deploy my application in at least two AWS regions, each with its own load balancer (ALB or NLB). Then I'd configure Route 53 with **Latency-Based Routing** or **Weighted Routing** to distribute traffic across regions. Each region would have health checks so that if one goes down, Route 53 automatically routes traffic to the healthy region. For the database, I'd use **Aurora Global Database** or **DynamoDB Global Tables** to ensure data replication. This gives me an **active-active architecture** with low latency for global users and built-in disaster recovery.

Question: ***Explain how you would secure an ALB that serves both internal microservices and public APIs.***

I'd separate traffic by using a **public ALB** in public subnets for external APIs and an **internal ALB** in private subnets for microservice communication. Both would enforce **HTTPS with ACM certificates**, but the public ALB would also be fronted by **AWS WAF** for application-layer protection. For internal services, I'd restrict access with **security groups**,

Question: ***How does an ALB handle sudden traffic spikes (e.g., flash sales)?***

Techniques to Handle Flash Sales

✅ **Pre-Warming ALB**

- If you know a big event (sale, product launch) is coming, you can **contact AWS Support** to **pre-warm the ALB**.
- This sets higher initial capacity, avoiding slow ramp-up.

✅ **Use Auto Scaling for Targets**

- Even if ALB scales, your **EC2/ECS targets must also scale fast enough**.
- Configure **Auto Scaling Groups with predictive or scheduled scaling** before the event.

✅ **Cross-AZ Load Distribution**

- ALB automatically balances across all **healthy targets in all AZs** → improves resiliency.

✅ **CloudFront + ALB**

- For global flash sales, put **CloudFront in front of ALB** to cache static/dynamic content and reduce load directly on ALB.

Question: ***How would you configure target tracking scaling policies with ALB request counts?***

I'd configure target tracking using the **RequestCountPerTarget metric** from the ALB. For example, I might set a target of 100 requests per instance. The Auto Scaling group then adds more EC2 instances whenever the average requests per target exceed 100 and scales in when load drops. I'd also configure **instance warmup** to avoid premature scaling decisions. This way, the application dynamically adjusts capacity based on real user demand.

Question: ***How do you tune idle connection timeouts for ALB vs. NLB?***

🔹 **Idle Connection Timeouts: ALB vs NLB**

1. Application Load Balancer (ALB)

- **Default:** 60 seconds idle timeout (HTTP/HTTPS, WebSockets, gRPC).
- **Configurable:** Can be tuned from **1 second → 4000 seconds (≈66 mins)**.

2. Network Load Balancer (NLB)

- **Default:** 350 seconds (≈5.8 minutes).
- **Configurable:**
  - Initially not configurable, but now supports tuning with **TCP idle timeout** (1–4000 seconds).
  - UDP traffic uses flow stickiness, where timeout is configurable (20–120 seconds default, up to 3600s).

Question: **What happens if all targets in a target group fail health checks?**

ALB (Application Load Balancer)

- ALB continuously runs **health checks** (default: every 30s, 2s timeout, configurable).
- If **all targets are marked unhealthy**:
    - **ALB returns 503 (Service Unavailable)** to clients.

NLB (Network Load Balancer)

- NLB also uses health checks (TCP/HTTP/HTTPS).
- If all targets are unhealthy:
    - By default, **NLB still forwards traffic to all targets**, even if marked unhealthy.
        - (This is different from ALB — NLB assumes the app might handle connections even if checks fail).

Gateway Load Balancer (GWLB)

- If all appliance targets (firewalls, IDS, etc.) fail health checks:
    - GWLB returns a **generic TCP reset (RST)** → traffic is dropped.

Question: ***How can you integrate WAF (Web Application Firewall) with ALB?***

To integrate WAF with ALB, I'd first create a **Web ACL** with **rules for SQL injection, XSS, IP blocking, or rate limits**. Then I'd associate the **Web ACL with the ALB** so that every request is inspected before reaching backend targets. This allows us to enforce security policies at Layer 7, protect APIs from common web exploits, and add rate limiting. I'd also monitor WAF logs in CloudWatch, starting with "count" mode before switching to "block" to avoid false positives.

S3 (Simple Storage Service)

Question: ***How does S3 integrate with IAM for fine-grained access control?***

1. IAM Policies
    - Attached to IAM users, groups, or roles

```
{
  "Effect": "Allow",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::my-bucket-name/*"
}

```

2. Bucket Policies

- Attached directly to **S3 buckets**

```
{
  "Effect": "Allow",
  "Principal": {"AWS": "arn:aws:iam::123456789012:user/JohnDoe"},
  "Action": "s3:PutObject",
  "Resource": "arn:aws:s3:::my-bucket-name/*"
}

```

3. Access Control Lists (ACLs)

- Legacy method for access control
- Used for **object-level permissions**

4. IAM Conditions

- Enable **fine-grained control** using conditions like:
    - IP address (aws:SourceIp)
    - Time of day (aws:CurrentTime)
    - Object tags (s3:ExistingObjectTag/<tag-key>)

```
"Condition": {
   "IpAddress": {
     "aws:SourceIp": "203.0.113.0/24"
   }
}

```

Question: ***difference between Policy vs Role***

🔐 **IAM Policy**

- A **policy** is a document that defines **permissions**.
- It specifies **what actions** are allowed or denied on **which resources**.
- Policies are written in **JSON** and can be attached to:
    - IAM users
    - IAM groups
    - IAM roles
    - AWS resources (like S3 buckets)

👤 **IAM Role**

- A **role** is an **identity** with a set of **permissions** defined by policies.
- Unlike users, roles **do not have long-term credentials**.
- Roles are **assumed** by trusted entities (users, services, or accounts).

Question: ***How would you enforce encryption at rest and in transit for all S3 objects?***

a. Enable Default Encryption on the Bucket

- Go to the S3 console → Bucket → Properties → Default encryption.
- Choose:
  - **SSE-S3** (Amazon-managed keys)
  - **SSE-KMS** (AWS Key Management Service – customer-managed keys)

b. **Enforce Encryption Using a Bucket Policy**

Prevent uploads of unencrypted objects:

```
"Effect": "Deny",
"Principal": "*",
"Action": "s3:PutObject",
"Resource": "arn:aws:s3:::your-bucket-name/*",
"Condition": {
  "StringNotEquals": {
    "s3:x-amz-server-side-encryption": "aws:kms"
  }
}
```

Question: ***Explain how to use AWS KMS with S3***

Using **AWS Key Management Service (KMS)** with **Amazon S3** allows you to manage encryption keys for data stored in S3 with greater control and security. Here's how it works and how it compares to other encryption options:

Question: ***How does S3 handle high request rates? What are best practices for prefix optimization?***

Amazon S3 is designed to **scale automatically** to handle **high request rates**, and AWS has made significant improvements over the years to eliminate many of the scaling limitations that used to exist.

⚙️ **How S3 Handles High Request Rates**

- **Automatic Scaling**: S3 automatically scales to support **thousands of requests per second** per prefix.
- **No More Prefix Partitioning Limits**: Previously, AWS recommended distributing keys across multiple prefixes to avoid throttling. Now, **S3 supports high request rates per prefix**, so this is no longer a strict requirement.

✅ Best Practices for Prefix Optimization

1. Distribute Object Keys

- Avoid sequential naming like log-0001, log-0002, etc.
- Use **hashing**, **UUIDs**, or **timestamp-based sharding** to spread keys across prefixes.

Example: Instead of:

logs/2025/08/22/log-0001.txt

Use:

logs/2025/08/22/3f9a-log-0001.txt

2. Use Parallelization

- Use **multi-threaded uploads/downloads** for large files.
- Use **Transfer Acceleration** or **S3 Multipart Upload** for better throughput.

3. Monitor with CloudWatch

- Track metrics like 4xx, 5xx, FirstByteLatency, and TotalRequestLatency.
- Use **S3 Storage Lens** for usage and activity trends.

4. Use S3 Access Points or Multi-Region Access Points

- For large-scale applications with many consumers, access points can help isolate and optimize access patterns.

Question**: *What is S3 Transfer Acceleration?***

- It routes data through the **nearest AWS edge location** using optimized network paths.
- Then it transfers the data to your S3 bucket over AWS's **high-speed backbone network**.
- This reduces latency and improves throughput for long-distance transfers.

Question: 🗂️ ***What Are S3 Lifecycle Policies?***

Amazon S3 Lifecycle Policies allow you to **automate transitions and deletions** of objects based on their age. This helps optimize **storage costs** and **data management** by moving data between storage classes or deleting it when no longer needed.

Question : ***What is S3 Object Lock and how does it support compliance use cases like WORM (Write Once Read Many)?***

Amazon S3 Object Lock is a feature that allows you to **prevent objects from being deleted or overwritten** for a fixed amount of time or indefinitely. It supports **WORM (Write Once, Read Many)** compliance, which is essential for industries like finance, healthcare, and legal where data immutability is required.

Key Features of S3 Object Lock

Feature	Description
WORM Protection	Ensures data cannot be modified or deleted during the retention period

Feature	Description
Retention Modes	- **Governance**: Allows privileged users to override - **Compliance**: No overrides allowed
Retention Period	Can be set per object or bucket default

Question**: *Limitations of S3 Static Website Hosting***

Limitation	Description
No HTTPS on S3 endpoint	You need **CloudFront** to enable HTTPS for custom domains.
❌ **No server-side logic**	Only supports static files—no PHP, Python, or databases.

Question : **Describe a use case where S3 Event Notifications are used to trigger downstream processing.**

🔄 **Workflow Overview**

1. **User uploads an image** to s3://my-photo-bucket/uploads/.
2. **S3 Event Notification** is triggered on PUT events.
3. The event invokes an **AWS Lambda function**.
4. The Lambda function:
    - Downloads the image.
    - Resizes it to a thumbnail.
    - Uploads the thumbnail to s3://my-photo-bucket/thumbnails/.

⚙️ **How to Set It Up**

1. Enable Event Notification on the S3 Bucket

- Event type: s3:ObjectCreated:*
- Prefix filter: uploads/
- Destination: Lambda function ARN

2. Lambda Function Logic

- Use libraries like Pillow (Python) or Sharp (Node.js) to resize images.
- Upload the resized image back to S3.

3. IAM Permissions

- Grant S3 permission to invoke the Lambda function.
- Grant Lambda permission to read from and write to the S3 bucket.

RDS

Question: ***How does Amazon RDS Multi-AZ deployment work internally? What happens during a failover?***

1. **Primary and Standby Instances**:
    - When you enable Multi-AZ, Amazon RDS automatically provisions and maintains a **synchronous standby replica** in a different Availability Zone (AZ).
    - The standby is **not accessible** for read or write operations. It's purely for failover.
2. **Synchronous Replication**:
    - Data is synchronously replicated from the primary to the standby instance.
3. **Shared Endpoint**:
    - RDS provides a **single DNS endpoint** for the database.

Question: **RDS vs Aurora: Key Differences**

Feature	Amazon RDS	Amazon Aurora
Database Engines	Supports MySQL, PostgreSQL, MariaDB, Oracle, SQL Server	Supports MySQL and PostgreSQL-compatible engines only
Performance	Standard performance based on selected instance type	Up to 5x faster than MySQL and 3x faster than PostgreSQL on RDS
Storage Architecture	EBS-backed storage, tied to instance	Distributed, fault-tolerant, self-healing storage across 6 copies in 3 AZs
High Availability	Multi-AZ deployments with standby instance	Built-in HA with automatic failover, no need for standby instance
Read Replicas	Up to 5 for MySQL/PostgreSQL	Up to 15 low-latency replicas with automatic failover
Backups	Automated backups and snapshots	Continuous backup to S3 with point-in-time recovery
Failover Time	1–2 minutes typically	Less than 30 seconds
Cost	Lower cost, pay per instance and storage	Higher cost, but better performance and availability

🔁 Multi-AZ Deployment

Feature	Description
Purpose	High availability and automatic failover
Replication Type	Synchronous
Read Access	Not available (standby is not readable)
Failover	Automatic to standby in case of failure

Read Replica

Feature	Description
Purpose	Scalability and read performance
Replication Type	Asynchronous
Read Access	Available (can be used for queries)
Failover	Manual (promote replica to standalone)

Question: **Can you explain how RDS handles failover in a Multi-AZ deployment?**

In a **Multi-AZ RDS deployment**, Amazon RDS automatically creates a **synchronous standby replica** in another Availability Zone. If the **primary instance fails** (due to hardware, network, or AZ issues), RDS **automatically fails over** to the standby. The database endpoint remains the same, so applications reconnect without needing changes. This ensures **high availability** with minimal downtime.

Question: **How do you secure an RDS instance? What are the best practices for encryption at rest and in transit?**

To **secure an RDS instance**:

- **Network**: Use VPC, security groups, and subnet groups to restrict access.
- **Authentication**: Use IAM DB authentication and enforce least privilege.
- **Encryption at Rest**: Enable RDS encryption with AWS KMS during creation.
- **Encryption in Transit**: Use SSL/TLS to encrypt connections; enforce SSL via DB parameters.

Question: **How does IAM integrate with RDS, and what are the limitations?**

IAM integration with RDS allows:

- **Access control** to RDS resources via IAM policies.
- **IAM DB authentication** (for MySQL & PostgreSQL) using temporary tokens instead of passwords.

Limitations:

- Only works with **MySQL and PostgreSQL**.
- No control over **database-level permissions** or **data access**.
- Tokens expire every **15 minutes**, requiring frequent renewal.

Question: **What are the limitations of RDS backups in terms of retention and cross-region recovery?**

- **Retention**: Automated backups can be retained for up to **35 days**. Manual snapshots can be kept indefinitely.
- **Cross-Region Recovery**:
    - **Automated backups** are **not cross-region** by default.
    - You must **manually copy snapshots** to another region for disaster recovery.
    - Cross-region snapshot copying may incur **additional costs** and **latency**.

Question : **How would you migrate an on-premise database to RDS with minimal downtime?**

To migrate an on-premise database to RDS with minimal downtime:

- Use **AWS Database Migration Service (DMS)**.
- Perform a **full data load** followed by **continuous replication** using **Change Data Capture (CDC)**.
- Once replication lag is low, do a **cutover** by switching application traffic to RDS.

Lambda

Question : **What is AWS Lambda?**

AWS Lambda is a **serverless compute service** that lets you run code **without provisioning or managing servers**.

Key Features:

- **Event-driven**: Triggered by events from AWS services (e.g., S3, DynamoDB, API Gateway).
- **Automatic scaling**: Scales based on the number of incoming requests.
- **Pay-per-use**: You only pay for the compute time your code uses.
- **Supports multiple languages**: Python, Node.js, Java, Go, .NET, and more.

Question: **How does AWS Lambda pricing work?**

Number of Requests

- First **1 million requests/month** are **free**.
- After that, you pay **$0.20 per million requests**.

Question: **What languages are supported by AWS Lambda?**

AWS Lambda supports the following programming languages:

- **Python**
- **Node.js (JavaScript)**
- **Java**
- **Go**
- **C# (.NET Core)**
- **Ruby**
- **PowerShell**
- **Custom runtimes** (via AWS Lambda Runtime API)

Question: **What is the maximum execution time for a Lambda function?**

The **maximum execution time** for an AWS Lambda function is **15 minutes (900 seconds)** per invocation. If the function runs longer, it is automatically terminated.

Question: **How do you trigger a Lambda function?**

You can **trigger an AWS Lambda function** in several ways:

🔹 **Common Trigger Sources**

- **API Gateway** – for RESTful APIs.
- **S3** – on file uploads or deletions.
- **DynamoDB** – on table updates.
- **CloudWatch Events** – for scheduled tasks (cron jobs).
- **SNS/SQS** – for messaging-based triggers.
- **EventBridge** – for event-driven workflows.
- **RDS Proxy** – for database-related triggers.

Question : **What is the difference between synchronous and asynchronous invocation?**

🔄 **Synchronous Invocation**

- The caller **waits for the function to finish** and gets the result immediately.
- Used by services like **API Gateway** or direct SDK/CLI calls.

🔁 **Asynchronous Invocation**

- The caller **does not wait**; Lambda queues the event and processes it later.
- Used by services like **S3**, **SNS**, or **EventBridge**.

Question: **Can Lambda functions access environment variables?**

Yes, **AWS Lambda functions can access environment variables**.
You can define environment variables when creating or updating a Lambda function. These variables are accessible within your code using standard methods

Question: **What is the role of IAM in Lambda?**

IAM (Identity and Access Management) plays a key role in AWS Lambda by:

🔐 **Controlling Access**

- Defines **who can create, update, or invoke** Lambda functions using IAM policies.
- Manages permissions for Lambda to access other AWS services (e.g., S3, DynamoDB) via **execution roles**.

Question: **How do you monitor and debug Lambda functions?**

🔍 **Monitoring**

- **Amazon CloudWatch Logs**: Captures logs from your Lambda function (console.log, print, etc.).
- **CloudWatch Metrics**: Tracks invocation count, duration, errors, throttles, etc.
- **AWS X-Ray**: Provides detailed tracing and performance insights.

🐞 **Debugging**

- Use **log statements** in your code to trace execution.
- Review **CloudWatch Logs** for errors and stack traces.

Question: **What are cold starts in Lambda and how do you mitigate them?**
Cold starts in AWS Lambda occur when a function is invoked after being idle or when a new instance is created. This causes a slight delay as AWS initializes the runtime environment.

❄️ **What Causes Cold Starts?**

- First invocation after deployment or update.
- Scaling up to handle more requests.
- Idle functions being reactivated.

🛠️ **How to Mitigate Cold Starts**

- Use **Provisioned Concurrency** to keep instances warm.
- Minimize **package size** and dependencies.
- Choose **lighter runtimes** (e.g., Node.js, Python).
- Avoid heavy initialization in global scope.

Question: **How do you manage dependencies in a Lambda function?**
Packaging Dependencies

- For **Python**: Use pip install to install packages into a folder, then zip it with your code.
- For **Node.js**: Use npm install to include modules in your project directory.
- For **Java**: Package dependencies using tools like Maven or Gradle.
- For **Go**: Compile with dependencies into a single binary.

✅ **Best Practices**

- Keep packages **lightweight** to reduce cold start time.
- Use **layers** to share common dependencies across functions.

Question: **How is a lambda function different from a regular function?**
🔁 **Lambda Function (AWS)**

- Runs in a **serverless environment**.
- Triggered by **events** (e.g., S3 upload, API call).

🧱 **Regular Function**

- Runs within a **program or application**.
- Called directly by code, not by external events.

Question: **Can a lambda function have multiple expressions?**

Yes, a **Lambda function** (in AWS or in programming in general) can contain **multiple expressions**.
In AWS Lambda, your function code can include:

- Multiple **statements and expressions**.
- Complex **logic**, **loops**, and **conditionals**.
- Calls to other functions or services.

Question: **How can you secure an AWS Lambda function?**

Security Measures

- **IAM Roles & Policies**
- **Environment Variables**
- **VPC Configuration**
- **Monitoring & Alerts**
- **Data Protection**

Question: **How do you handle state management in serverless applications?**

🔹 **1. External State Storage**

Since serverless functions don't retain state between invocations, you store state externally using:

- **Databases**:
  - **Amazon DynamoDB** (NoSQL, fast, scalable)
  - **Amazon RDS / Aurora** (Relational)
- **Object Storage**:
  - **Amazon S3** for storing files, logs, or serialized state.

2. Event-Driven State Management

Use **event sourcing** and **message queues** to manage state transitions:

- **Amazon SQS / SNS**: Queue and publish events.
- **Amazon EventBridge**: Route events between services.

API Gateway

What is Amazon API Gateway, and how does it work?

Amazon API Gateway is a fully managed service by AWS that enables developers to create, publish, maintain, monitor, and secure **APIs (Application Programming Interfaces)** at any scale.

🛠️ **How It Works**

1. **API Creation**:
    - You define **resources** (like /users, /orders) and **methods** (like GET, POST).
    - These methods are linked to **backend integrations** such as:
        - AWS Lambda functions
        - HTTP endpoints
        - AWS services (like DynamoDB, S3)
2. **Request Handling**:
    - When a client sends a request, API Gateway:
        - Validates the request
        - Applies throttling, authorization, and transformation rules
        - Forwards the request to the backend
3. **Response Processing**:
    - The backend sends a response to API Gateway
    - API Gateway can **transform** the response (e.g., format JSON)
    - Sends the final response to the client
4. **Security & Monitoring**:
    - Supports **IAM**, **Cognito**, **API keys**, and **Lambda authorizers**
    - Integrated with **CloudWatch** for logging and metrics

🔐 **Key Features**

- **Supports REST, HTTP, and WebSocket APIs**
- **Built-in throttling and caching**
- **Custom domain names and SSL support**
- **Request/response transformation using mapping templates**
- **Monitoring with AWS CloudWatch**

Question: **What are the key components of an API Gateway API?** (e.g., Resources, Methods, Integrations, Stages)

1. Resources

- Represent **individual endpoints** or paths in your API (e.g., /users, /orders).
- You can define a **hierarchy** of resources to organize your API logically.

2. Methods

- Define the **HTTP actions** (e.g., GET, POST, PUT, DELETE) that can be performed on a resource.
- Each method can have:
    - **Request validation**

- **Authorization**
- **Integration settings**
- **Response mapping**

3. Integrations

- Define how API Gateway connects to the **backend** to process requests.
- Common integration types:
  - **AWS Lambda**
  - **HTTP/HTTPS endpoints**
  - **AWS services** (e.g., DynamoDB, S3)
  - **Mock integration** (for testing)

4. Stages

- Represent **deployment environments** like dev, test, or prod.
- Each stage can have:
  - **Stage variables**
  - **Logging and metrics**
  - **Throttling and caching settings**

5. Models

- Define the **structure of request and response payloads** using JSON Schema.
- Useful for **validation** and **documentation**.

6. Authorizers

- Control **access to your API**.
- Types:
  - **IAM-based**
  - **Cognito User Pools**
  - **Lambda authorizers (custom)**

7. Usage Plans & API Keys

- Control **who can access your API** and how much they can use it.
- Define **quotas** and **rate limits** for different users or applications.

Question : **Explain the difference between REST APIs, HTTP APIs, and WebSocket APIs in API Gateway.**

🔷 **1. REST APIs**

- **Use Case**: Full-featured APIs for complex applications.
- **Features**:
  - Supports **API keys**, **usage plans**, **Lambda authorizers**, and **custom domain names**.
  - Advanced **request/response transformation** using **Velocity Template Language (VTL)**.
  - **Caching**, **throttling**, and **monitoring** via CloudWatch.
- **Best For**: Legacy systems or when you need fine-grained control and advanced features.

🔶 **2. HTTP APIs**

- **Use Case**: Lightweight, fast, and cost-effective APIs.
- **Features**:
  - Supports **JWT authorizers**, **CORS**, and **OAuth 2.0**.
  - Faster performance and lower cost than REST APIs.
  - Simplified setup and configuration.
- **Limitations**:
  - Fewer features than REST APIs (e.g., no usage plans or request transformation).
- **Best For**: Modern web/mobile apps needing speed and simplicity.

🔷 **3. WebSocket APIs**

- **Use Case**: Real-time, two-way communication (e.g., chat apps, live dashboards).
- **Features**:
  - Maintains persistent connections.
  - Supports **routes** for message handling.
  - Integrates with **Lambda** for backend processing.
- **Best For**: Applications requiring **low-latency, real-time updates**.

Question : **How does API Gateway integrate with other AWS services like Lambda, S3, or DynamoDB?**

🧩 **1. AWS Lambda Integration**

- **Use Case**: Serverless backend logic.
- **How It Works**:
  - API Gateway routes incoming HTTP requests to a **Lambda function**.
  - Lambda processes the request and returns a response.
  - API Gateway can transform both the request and response using **mapping templates**.
- **Benefits**:
  - No need to manage servers.
  - Scales automatically.
  - Ideal for microservices and event-driven architectures.

🗂️ **2. Amazon S3 Integration**

- **Use Case**: Serving static content or uploading files.
- **How It Works**:
  - API Gateway can be configured to **proxy requests** to S3 buckets.
  - You can use **IAM roles** to authorize access.
  - Common for **file uploads**, **downloads**, or **static website hosting**.
- **Benefits**:
  - Secure access to S3 via API.
  - Can restrict access using signed URLs or IAM policies.

📊 **3. Amazon DynamoDB Integration**

- **Use Case**: NoSQL database operations.
- **How It Works**:

    - API Gateway can directly integrate with DynamoDB using **AWS service integration**.
    - You define **request templates** to format the data for DynamoDB actions (e.g., PutItem, GetItem).
    - Alternatively, use Lambda to handle complex logic before interacting with DynamoDB.
- **Benefits**:
    - Fast and scalable data access.
    - Serverless and cost-effective.

🔐 **Security & Monitoring**

- API Gateway uses **IAM roles**, **Cognito**, or **Lambda authorizers** to secure access.
- Integrated with **CloudWatch** for logging, metrics, and alarms.

Question : **What are common use cases for AWS API Gateway?**

🧩 **1. Serverless Web Applications**

- API Gateway acts as the **front door** for web apps built with AWS Lambda, S3, and DynamoDB.
- Handles HTTP requests and routes them to Lambda functions for processing.

📱 **2. Mobile Backend APIs**

- Provides secure and scalable APIs for mobile apps.
- Integrates with **Amazon Cognito** for user authentication and authorization.

🔐 **3. Secure Microservices Communication**

- Enables communication between microservices using REST or HTTP APIs.
- Supports **IAM roles**, **JWT tokens**, and **Lambda authorizers** for secure access.

📦 **4. Third-Party API Exposure**

- Allows organizations to expose internal services to external developers or partners.
- Supports **rate limiting**, **API keys**, and **usage plans** to manage access.

📊 **5. Real-Time Applications**

- Use **WebSocket APIs** for chat apps, live dashboards, or gaming platforms.
- Maintains persistent connections for low-latency communication.

🧪 **6. Prototyping and Mock APIs**

- Quickly create mock integrations for testing and development.
- Use **mock integrations** to simulate backend responses.

📜 **7. Data Ingestion and Processing**

- Collects data from IoT devices or external sources.
- Routes data to services like **Kinesis**, **Lambda**, or **S3** for processing and storage.

🛡️ **8. API Gateway as a Security Layer**

- Acts as a **proxy** to protect backend services.

- Implements **throttling**, **WAF (Web Application Firewall)**, and **TLS encryption**.

Question : **How do you secure APIs built with API Gateway?** (e.g., IAM, Custom Authorizers, Cognito User Pools, API Keys)

🔐 **1. IAM (AWS Identity and Access Management)**

- **Use Case**: Internal APIs or trusted AWS users/services.
- **How It Works**:
  - IAM roles and policies control who can invoke the API.
  - Best for **service-to-service** communication within AWS.
- **Pros**: Strong security, tightly integrated with AWS.
- **Cons**: Not ideal for public-facing APIs.

🧠 **2. Lambda Authorizers (Custom Authorizers)**

- **Use Case**: Custom authentication logic (e.g., token validation, role-based access).
- **How It Works**:
  - A Lambda function runs before the API is invoked.
  - It validates headers, tokens, or parameters and returns an IAM policy.
- **Pros**: Highly flexible, supports custom auth schemes.
- **Cons**: Adds latency due to Lambda execution.

👥 **3. Amazon Cognito User Pools**

- **Use Case**: User authentication for mobile/web apps.
- **How It Works**:
  - API Gateway validates **JWT tokens** issued by Cognito.
  - Supports sign-up, sign-in, and user management.
- **Pros**: Scalable, secure, integrates with social and enterprise IdPs.
- **Cons**: Limited to JWT-based flows.

🔑 **4. API Keys and Usage Plans**

- **Use Case**: Rate limiting and access control for third-party developers.
- **How It Works**:
  - API keys are issued to clients.
  - Usage plans define **quotas** and **throttling limits**.
- **Pros**: Easy to implement, good for public APIs.
- **Cons**: Not secure for sensitive data (no user identity).

🛡️ **5. AWS WAF (Web Application Firewall)**

- **Use Case**: Protection against common web exploits (e.g., SQL injection, XSS).
- **How It Works**:
  - WAF rules inspect incoming traffic before it reaches API Gateway.
- **Pros**: Adds an extra layer of security.
- **Cons**: Requires configuration and monitoring.

Question : **Explain the purpose of API Gateway Custom Authorizers.**

🔐 **Purpose of API Gateway Custom Authorizers**

Custom Authorizers in AWS API Gateway are **Lambda functions** that you create to control access to your APIs. They allow you to implement **custom authentication and authorization logic** beyond what's available with built-in mechanisms like IAM or Cognito.

🧠 **How They Work**

1. **Client Request**: A client sends a request to your API with a token (e.g., JWT, OAuth token, API key).
2. **Custom Authorizer Triggered**: API Gateway calls your Lambda authorizer before forwarding the request to the backend.
3. **Token Validation**: The Lambda function validates the token and returns an IAM policy that either **allows or denies** access.
4. **Policy Enforcement**: API Gateway uses the returned policy to decide whether to proceed with the request.

✅ **Use Cases**

- Validating **third-party tokens** (e.g., Auth0, Okta).
- Implementing **role-based access control**.
- Supporting **custom headers or query parameters** for authentication.
- Adding **fine-grained access control** based on user attributes.

📌 **Example Scenario**

You have an API that should only be accessible to users with a valid JWT token issued by a third-party identity provider. A **custom authorizer** can:

- Decode the token
- Verify its signature and expiration
- Check user roles or permissions
- Return an IAM policy to allow or deny access

Question : **How can you implement rate limiting and throttling in API Gateway?**

🔑 **1. Usage Plans and API Keys**

✅ **Purpose:**

Control access and usage for different clients or applications.

🛠️ **How to Set Up:**

1. Create an **API Key**.
2. Create a **Usage Plan** that includes:
    - **Rate Limit**: Requests per second.
    - **Burst Limit**: Maximum number of requests allowed in a short burst.
    - **Quota**: Total requests allowed per day/week/month.
3. Associate the API Key with the Usage Plan.

📌 **Example:**

- Rate: 100 req/sec
- Burst: 200
- Quota: 10,000 req/day

⚙️ **2. Stage-Level Throttling**

✅ **Purpose:**

Apply default throttling settings across all methods in a stage (e.g., dev, prod).

🛠️ **How to Set Up:**

- In the **stage settings**, configure:
  - **Rate Limit**
  - **Burst Limit**

This applies to all methods unless overridden.

🔧 **3. Method-Level Throttling**

✅ **Purpose:**

Fine-tune throttling for specific endpoints.

🛠️ **How to Set Up:**

- In the method settings (e.g., GET /users), override the stage-level throttling.

🛡️ **4. AWS WAF (Web Application Firewall)**

✅ **Purpose:**

Advanced protection against abuse, bots, and attacks.

🛠️ **How to Set Up:**

- Attach a **WAF Web ACL** to your API Gateway.
- Define rules to block or rate-limit traffic based on:
  - IP addresses
  - Request patterns
  - Geographic location

📊 **Monitoring and Alerts**

- Use **Amazon CloudWatch** to monitor:
  - Throttled requests
  - Latency
  - Error rates
- Set up **CloudWatch Alarms** to alert on unusual traffic patterns.

Question : How do you handle different stages of an API using API Gateway? (e.g., dev, test, prod)

How API Gateway Handles Different Stages

1. **Stages in API Gateway**
   - API Gateway allows you to define **stages** (e.g., dev, test, prod) for an API deployment.
   - Each stage has its **own URL endpoint**, such as:
   - https://{api-id}.execute-api.{region}.amazonaws.com/dev
   - https://{api-id}.execute-api.{region}.amazonaws.com/test
   - https://{api-id}.execute-api.{region}.amazonaws.com/prod
2. **Stage Variables**
   - You can define **key-value pairs** at the stage level (like environment configs).
   - Example:
     - lambdaAlias=dev → maps API requests to the **dev version** of a Lambda.

        - lambdaAlias=prod → maps API requests to the **production version**.
3. **Separate Deployment Versions**
    - Each deployment can be linked to a **stage**.
    - You deploy new changes to dev first, test them, then promote the same deployment to test or prod.
4. **Usage Plans & API Keys**
    - You can associate **different usage plans** with different stages.
    - Example:
        - dev → relaxed throttling, no billing.
        - prod → stricter quotas, metered usage.
5. **Custom Domain Names & Base Path Mapping**
    - You can map custom domains:
        - api.example.com/dev → points to dev stage.
        - api.example.com/prod → points to prod stage.
    - Helps avoid exposing the raw API Gateway URL.
6. **Best Practices**
    - Keep **separate stages** for dev/test/prod.
    - Use **stage variables + Lambda aliases/versions** to switch between environments without changing code.
    - Apply **different IAM roles, WAF rules, throttling policies** per stage for isolation and security.
    - Automate deployments using **CI/CD pipelines (e.g., CodePipeline + SAM/CloudFormation)**.

✅ **Example in real-world**:

- Developer pushes code → pipeline deploys to **API Gateway dev stage** → testers verify → same deployment promoted to **prod stage** with stricter throttling & logging.

Question : Explain how caching works in API Gateway and its benefits.

How Caching Works in API Gateway

1. **Where caching happens**
    - API Gateway provides an **integrated cache at the stage level**.
    - It caches responses for **method requests** (GET, POST, etc.) so repeated requests don't always hit your backend (Lambda, EC2, etc.).
2. **Cache Size**
    - You can allocate cache size from **0.5 GB to 237 GB**.
    - Cache is **provisioned per stage** and incurs extra cost.
3. **Cache Key**
    - Cache entries are identified by a **cache key**, which includes:
        - Request path (/users)
        - Query string parameters (?id=123)
        - Headers (optional, configurable)
        - Request body (optional)

🔑 You can configure which parameters/headers are included in the cache key.

4. **TTL (Time to Live)**
    - Each cached response has a **TTL** (default = 300 seconds, max = 3600 seconds).
    - After TTL expiry, the next request fetches a fresh response from the backend.
5. **Cache Invalidation**
    - Cache can be **manually invalidated** (e.g., after a deployment).
    - Clients can also force bypassing cache using Cache-Control: max-age=0 header (if allowed).

Benefits of Caching in API Gateway

1. **Improved Performance** 🚀
    - Reduces latency because API Gateway serves responses directly from cache instead of calling backend services.
2. **Lower Backend Load** ⚡
    - Backend (Lambda, EC2, DynamoDB, etc.) gets fewer requests → saves compute and database read costs.
3. **Reduced Costs** 💰
    - Since fewer requests reach the backend, you save on **Lambda execution time**, **DynamoDB reads**, or **EC2 usage**.
4. **Consistent Responses** 📌
    - Clients receive cached responses quickly and consistently for frequently accessed data.
5. **Configurable Control** 🔧
    - You can control **per-method caching**, **TTL**, and **cache keys** (query params/headers) to fine-tune caching behavior.

✅ **Example:**

- Suppose your API GET /products?id=101 fetches product details from DynamoDB.
- Without caching → every request queries DynamoDB.
- With caching → API Gateway stores the response for id=101 in cache. Next requests return the cached result instantly, until TTL expires.

Question : How can you implement request validation in API Gateway?

Ways to Implement Request Validation in API Gateway

1. Enable Request Validation in API Gateway

- API Gateway has a **built-in request validation feature**.
- You can configure it to validate:
    1. **Parameters only** (query string, headers, path params)
    2. **Body only** (against a JSON schema defined in a request model)
    3. **Both parameters and body**

✅ Example:

- You define a **Model** (e.g., JSON schema for POST /users).
- Attach a **Request Validator** to the method (say, “Validate body”).

- API Gateway will reject requests that don't match the schema before hitting your Lambda/EC2 backend.

---

2. Models & JSON Schema

- Models in API Gateway are defined using **JSON Schema**.
- Example schema for POST /user:

```
{
 "type": "object",
 "properties": {
  "username": { "type": "string" },
  "email": { "type": "string", "format": "email" }
 },
 "required": ["username", "email"]
}
```

- If the client sends { "username": "avadhut" } (missing email), API Gateway returns **400 Bad Request** automatically.

---

3. Method Request Parameters

- You can mark **query params, path params, or headers** as **required**.
- Example: /orders?orderId=123 → if orderId is missing, API Gateway rejects the request.

---

4. Custom Request Validation with Lambda Authorizer

- For advanced use cases (e.g., checking token format, business rules), you can use a **Lambda Authorizer** to validate requests before they reach the backend.
- Example: Validate Authorization header format or decode JWT claims.

Benefits of Request Validation

1. **Stops bad requests early** — avoids wasting backend resources.
2. **Consistent error handling** — clients get 400 errors with clear messages.
3. **Improves security** — prevents malformed payloads and injection attempts.
4. **Cost savings** — fewer invalid requests hit Lambda, EC2, or databases.

✅ **Example in Real World**

- POST /register expects { "username": "John", "email": "john@email.com" }.
- If the request is missing email, API Gateway itself blocks it with 400 error — Lambda never runs.

Question : Describe the process of deploying an API Gateway API.

Process of Deploying an API in API Gateway

1. Create or Update Your API

- Define **resources** (e.g., /users, /orders) and **methods** (GET, POST, etc.).
- Configure **integration type** (Lambda, HTTP, Mock, or AWS service).
- Optionally configure:
  - **Request/response models**
  - **Request validation**
  - **Authorizers** (IAM, Cognito, Lambda)

---

2. Test the API in the Console

- Use the API Gateway console "**Test**" feature to make sure each method works before deployment.
- At this stage, the API is **not yet publicly accessible** — it only exists as a definition.

---

3. Create a Deployment

- In API Gateway, you must create a **Deployment** to make your API available.
- A deployment is essentially a **snapshot** of your API configuration at a point in time.

---

4. Associate Deployment with a Stage

- Every deployment must be linked to a **Stage** (e.g., dev, test, prod).
- Each stage gets a **unique invoke URL**, like:
- https://{api-id}.execute-api.{region}.amazonaws.com/prod
- Stages have settings like:
    - **Stage variables** (for environment-specific configs, e.g., Lambda alias = prod/dev).
    - **Caching** (enabled/disabled, TTL).
    - **Logging/Tracing** (CloudWatch, X-Ray).
    - **Throttling and quotas**.

---

5. Optional – Map a Custom Domain

- You can map your API to a **custom domain name**, such as:
    - https://api.example.com/dev
    - https://api.example.com/prod
- This makes the API more user-friendly and easier to manage.

---

6. Test the Deployed API

- Send requests via Postman, curl, or your application.
- Verify **authentication, validation, caching, throttling** behave as expected.

---

7. Promote Through Stages

- Common workflow:
    - Deploy new version → dev stage.
    - Run automated/manual tests.
    - Promote the same deployment → test stage.
    - Finally, deploy → prod stage for customers.

Best Practices

- Automate deployments using **CI/CD pipelines** (AWS CodePipeline + CloudFormation/SAM/Terraform).
- Use **stage variables** + **Lambda aliases** for environment isolation.
- Enable **CloudWatch logging** for each stage.
- Use **deployment history** to roll back if needed.

✅ **Example in Real World**

- A team creates POST /orders API with Lambda integration.
- They deploy to dev stage first.

- QA team tests API in dev.
- After approval, they redeploy the same config to prod, where clients access it via https://api.company.com/orders.

Question : How do you handle request and response transformations using mapping templates in API Gateway?

What Are Mapping Templates?

- A **mapping template** is written in **Velocity Template Language (VTL)**.
- It lets you **modify request/response payloads** between:
    - **Client → API Gateway → Backend** (request transformation)
    - **Backend → API Gateway → Client** (response transformation)

---

1. Request Transformation (Client → Backend)

You use mapping templates in the **Method Request → Integration Request**.

- Common use cases:
    - Rename/reshape request body fields.
    - Combine query parameters, headers, and body into one payload.
    - Add metadata (stage variables, context info like requestId, callerIp).

✅ **Example:** Client sends:

```
{
 "fname": "John",
 "lname": "Doe"
}
```

You want backend Lambda to receive:

```
{
 "fullName": "John Doe",
 "requestId": "$context.requestId"
}
```

👉 Mapping template (VTL):

```
{
 "fullName": "$input.json('$.fname') $input.json('$.lname')",
 "requestId": "$context.requestId"
}
```

---

2. Response Transformation (Backend → Client)

You use mapping templates in **Integration Response → Method Response**.

- Common use cases:
    - Mask sensitive fields before sending back to client.
    - Standardize error responses.
    - Change backend field names to client-friendly ones.

✅ **Example:** Backend returns:

```
{
 "user_id": 123,
 "user_email": "john@example.com",
 "internalFlag": true
}
```

You want client to receive:

```
{
 "id": 123,
 "email": "john@example.com"
}
```

👉 Mapping template:

```
{
 "id": $input.json('$.user_id'),
 "email": $input.json('$.user_email')
}
```

---

3. Context & Stage Variables

- You can inject **request context variables**:
  - $context.identity.sourceIp → client IP
  - $context.httpMethod → HTTP method
  - $context.stage → current stage (dev/test/prod)
- You can inject **stage variables**:
  - $stageVariables.lambdaAlias → used for environment-specific Lambda versions

---

4. Benefits

1. **Decouple client and backend** → frontend doesn't need to change if backend format changes.
2. **Reduce backend logic** → simple transformations are handled at the API Gateway layer.
3. **Consistent API responses** → even if multiple backends differ.
4. **Security** → mask/remove sensitive data before sending to clients.

✅ **Real-world Example**

- Mobile app sends { "lat": 19.07, "lng": 72.87 }
- Backend expects { "location": "19.07,72.87" }
- API Gateway request mapping template combines them into a single string → saves rewriting backend code.

Question : How can you monitor and log API Gateway activity using CloudWatch?

How to Monitor & Log API Gateway with CloudWatch

1. Enable CloudWatch Logs

- Go to **API Gateway → Stage settings**.
- Enable:
  - **CloudWatch Logs** → Captures execution logs for each request.
  - **Log level** → ERROR / INFO / DEBUG.
  - **Full request/response data** (optional; use carefully, can expose sensitive data).

👉 Logs appear in **CloudWatch Logs Groups** under:
/aws/apigateway/{api-id}/stages/{stage-name}

---

2. Enable CloudWatch Metrics

- API Gateway automatically publishes **metrics per stage** to CloudWatch.

- Common metrics:
  - 4XXError → Client-side errors (bad requests).
  - 5XXError → Backend/integration errors.
  - Latency → Time between request and response.
  - IntegrationLatency → Time API Gateway waits for backend.
  - Count → Number of requests.
  - CacheHitCount / CacheMissCount → Cache performance.

👉 You can create **CloudWatch Dashboards** or **Alarms** on these metrics.

---

3. Access Logs

- Access logs record **who called the API** and **what they sent**.
- Configure an **Access Log format** with variables like:
- $context.identity.sourceIp - $context.httpMethod $context.resourcePath $context.status
- Example log line:
- 192.168.1.10 - GET /orders 200

---

4. Tracing with X-Ray (Optional)

- You can enable **AWS X-Ray tracing** for deeper visibility.
- Shows request path across API Gateway, Lambda, DynamoDB, etc.
- Helps find bottlenecks.

---

5. Best Practices

- **Enable ERROR logging** at minimum → avoids cost explosion from full request logs.
- Use **log filters** in CloudWatch to find specific requests (@requestId).
- Set **CloudWatch Alarms** for:
  - High 5XXError rate.
  - Latency spikes.
  - Sudden drop in Count.
- Send CloudWatch Logs to **S3 or Elasticsearch/OpenSearch** for long-term analysis.

✅ **Example in Real World**

- API /checkout → sudden spike in 5XXError.
- CloudWatch metrics alarm triggers SNS alert.
- Dev checks CloudWatch logs: finds “Timeout” error in Lambda integration.
- Root cause → DB queries too slow.
- Solution → optimize DB, increase Lambda timeout.

Question : How would you troubleshoot a 5xx error in API Gateway integrated with Lambda?

How to Troubleshoot 5xx Errors in API Gateway + Lambda

1. Understand the Error Types

- **502 Bad Gateway** → API Gateway couldn’t understand the response from Lambda (often malformed response, or timeout).
- **503 Service Unavailable** → API Gateway capacity issue (rare) or throttling.
- **504 Integration Timeout** → Lambda didn’t respond within the timeout period (default max = 29s).

- **500 Internal Server Error** → Lambda crashed, unhandled exception, or permission issue.

---

2. Check CloudWatch Metrics

- Look at **API Gateway metrics** in CloudWatch:
  - 5XXError → spikes show backend issue.
  - Latency & IntegrationLatency → see if Lambda is slow.
- Look at **Lambda metrics**:
  - Duration → is it hitting the timeout?
  - Errors → count of unhandled exceptions.
  - Throttles → exceeding concurrency limits?

---

3. Check CloudWatch Logs

- **API Gateway Execution Logs**
  - Enable logs at the stage level (INFO or ERROR).
  - Look for requestId to trace failing calls.
- **Lambda Function Logs**
  - Check logs for exceptions, stack traces, or incorrect return payloads.

👉 Example of a common issue:
Lambda returns:
{ "status": 200, "body": "OK" }
But API Gateway expects:
{ "statusCode": 200, "body": "OK" }
→ This mismatch triggers a **502 error**.

---

4. Validate Integration Setup

- Check **Integration Request/Response mapping templates**:
  - Are you passing the request correctly to Lambda?
  - Is the response mapping extracting the right fields?
- Ensure **Lambda permissions** (execution role has correct AWSLambdaBasicExecutionRole, and API Gateway is allowed to invoke the Lambda).

---

5. Common Root Causes & Fixes

Error Type	Likely Cause	Fix
502 Bad Gateway	Malformed Lambda response (missing statusCode/body), or Lambda returned binary data without proper config	Fix Lambda to return correct JSON format ({statusCode, body})
504 Timeout	Lambda taking too long (slow DB query, external API delay)	Increase Lambda timeout (max 15 min), optimize code, add retries/backoff
500 Internal Server Error	Lambda unhandled exception (e.g., null reference, JSON parse error)	Add proper error handling & logging in Lambda
Throttling (429/503)	Too many requests, Lambda hitting concurrency limits	Increase Lambda concurrency limit, use DLQ/SQS buffering, implement throttling

---

6. Test in Isolation

- Run Lambda **directly** from AWS Console with the same input.
  - If it fails → issue is in Lambda code.
  - If it works → issue is in API Gateway mapping or integration.

---

7. Use X-Ray for Deep Tracing

- Enable **AWS X-Ray** on API Gateway + Lambda.
- You'll see detailed traces of request path, including where timeouts or errors occur.

✅ **Example in Real World**

- You deploy a new Lambda function behind API Gateway.
- Clients start getting **502 Bad Gateway**.
- CloudWatch logs show:
- "errorMessage": "statusCode is not defined"
- Root cause → Lambda returned { "status": 200 } instead of { "statusCode": 200 }.
- Fix → Correct Lambda response format → issue resolved.

Question : Explain how to set up a custom domain name for an API Gateway endpoint.

Steps to Set Up a Custom Domain Name in API Gateway

1. Register a Domain

- Register a domain name with **Route 53** (or another provider).
  Example: api.mycompany.com

---

2. Create or Import an SSL/TLS Certificate

- API Gateway requires an **SSL certificate** in **AWS Certificate Manager (ACM)**.
- Certificate must be in the **same AWS Region** as your API Gateway.
- Steps:
  - Go to **ACM → Request a certificate**.
  - Add domain name (e.g., api.mycompany.com).
  - Validate via **DNS** (recommended) or **Email**.

---

3. Configure Custom Domain in API Gateway

- In the **API Gateway console**:
  - Go to **Custom domain names → Create custom domain name**.
  - Enter domain (e.g., api.mycompany.com).
  - Choose **Endpoint type**:
    - **Regional** (default, used with CloudFront implicitly).
    - **Edge-optimized** (older, creates CloudFront distribution).
    - **Private** (inside VPC).
  - Select the SSL certificate from ACM.

---

4. Set Up Base Path Mappings

- A **Base Path Mapping** connects the custom domain to a specific API + stage.
  - Example:
    - https://api.mycompany.com/v1 → API Gateway → prod stage.
    - https://api.mycompany.com/dev → API Gateway → dev stage.

👉 You can also map the root (/) if you don't want /v1 in the path.

---

5. Update DNS in Route 53

- API Gateway provides a **target domain name** (like d-abcdef123.execute-api.us-east-1.amazonaws.com).
- In **Route 53**:
    - Create a **CNAME or Alias record** pointing api.mycompany.com → API Gateway target domain.
    - Use **Alias** if available (better, integrates with CloudFront).

---

6. Test the Custom Domain

- Now you can access API via:
- https://api.mycompany.com/orders

instead of
https://abcdef123.execute-api.us-east-1.amazonaws.com/prod/orders

---

Best Practices

- Use **Regional endpoints** unless you need global edge optimization.
- Always use **ACM-managed certificates** (free auto-renewal).
- Apply **WAF (Web Application Firewall)** if exposing to the internet.
- Use **Base Path versioning** (/v1, /v2) for smoother upgrades.

✅ **Example in Real World**

- Company sets up api.company.com.
- /v1 → maps to prod stage.
- /dev → maps to dev stage.
- DNS in Route 53 points to API Gateway domain → clients now always use a friendly domain.

Question : Discuss how API Gateway can be used with a VPC Link.

What is VPC Link in API Gateway?

- A **VPC Link** allows API Gateway to securely connect to **resources inside a VPC** (like private ALBs, NLBs, or EC2 instances).
- Normally, API Gateway is **public** (internet-facing), but many backends (databases, services) are in **private subnets** — VPC Link bridges that gap.

How It Works

1. **Create a VPC Link**
    - In API Gateway, you create a VPC Link resource.
    - It establishes a **managed ENI (Elastic Network Interface)** in your VPC.
    - That ENI routes traffic from API Gateway to your **private VPC resources**.
2. **Integration with API Gateway**
    - When configuring **Integration Request** for a method:
        - Choose **VPC Link** as the integration type.
        - Select target **NLB** or **ALB** inside your VPC.
    - API Gateway → VPC Link → Load Balancer → private service.
3. **Client Access Flow**

- Client calls API Gateway endpoint (https://api.company.com/orders).
- API Gateway forwards request via **VPC Link**.
- Request reaches private ALB/NLB → forwards to EC2, ECS, or microservice.
- Response returns back through VPC Link → API Gateway → client.

When to Use VPC Link

- You have **microservices** or **internal APIs** running in private subnets.
- You don't want to expose backend services directly to the internet.
- You want **secure connectivity** between API Gateway and VPC workloads without opening security groups broadly.

Benefits

- 🔒 **Security** → Backend stays private inside the VPC.
- ⚡ **Performance** → Uses AWS internal network (low latency, high throughput).
- 🔗 **Scalability** → Works well with ALBs/NLBs handling multiple private services.
- 🛡️ **Compliance** → Backend never exposed to public internet, useful for finance/healthcare workloads.

Limitations

- API Gateway → VPC Link → only supports **NLB/ALB** targets (not directly EC2 IPs).
- **VPC Link is regional** → can't span regions.
- Additional **cost per VPC Link** (hourly + data processing charges).

Real-World Example

- A company runs its **Order Service** on ECS (private subnets, behind an NLB).
- They expose /orders API via API Gateway → VPC Link → NLB → ECS tasks.
- Clients access the API securely, but backend remains private inside the VPC.

✅ In interviews, you can summarize like this:
"VPC Link in API Gateway allows private integration with services inside a VPC. I'd use it when I want API Gateway to securely connect to private ALBs/NLBs without exposing my backend directly to the internet."

Question : How would you implement blue/green or canary deployments with API Gateway?

1. Blue/Green Deployment with API Gateway

In **blue/green**, you maintain **two separate environments** (blue = current, green = new).

How to implement in API Gateway:

- **Separate stages**:
  - prod-blue → current production stage.
  - prod-green → new deployment (updated Lambda, integration, or mapping templates).
- **Custom domain base path mapping**:
  - https://api.company.com/ → points to prod-blue.
  - Once tested, remap it to prod-green.
- **Switch traffic instantly** by updating the base path mapping.

👉 Benefit: **Zero downtime cutover** (but it's an all-or-nothing switch).

---

2. Canary Deployment with API Gateway

In **canary deployments**, you release new changes to a **small percentage of traffic** before rolling out fully.

How to implement in API Gateway:

- Enable **canary settings** on a stage (e.g., prod).
- Specify:
  - **Traffic weight** → e.g., 10% of requests go to new deployment, 90% to old.
  - **Stage variables** for canary → point to new Lambda alias/version, backend service, or config.
- Monitor CloudWatch metrics (latency, 5XX errors).
- If stable → gradually increase traffic to 50%, then 100%.
- If issues → instantly roll back by setting canary traffic weight to 0%.

👉 Benefit: **Safe, gradual rollout** with easy rollback.

---

3. Canary with Lambda Aliases (Best Practice)

- Use **Lambda versions + aliases** (prod, green) along with API Gateway canaries.
- Canary stage variables map to **different Lambda aliases**.
- Example:
  - 90% → Lambda alias prod-v1 (stable).
  - 10% → Lambda alias prod-v2 (new release).

This gives even finer control — you can shift both **API Gateway routing** and **Lambda traffic weighting**.

Best Practices

- Always monitor **CloudWatch metrics & logs** during rollout (look at 5XXError, Latency, Count).
- Use **automated alarms** (SNS alerts) to auto-roll back on failure.
- For blue/green → run **automated smoke tests** before cutover.
- For canary → start with very low % (1–5%) before scaling up.

✅ **Example in Real World**

- You run orders API on API Gateway + Lambda.
- Deploy new version → set **canary = 5% traffic**.
- Monitor CloudWatch → no errors after 1 hour.
- Increase canary to 50%, then 100%.
- If errors spike → roll back canary to 0% instantly.

AWS API Gateway Interview Q&A (Brief)

Basics

1. **What is AWS API Gateway?**
   → A fully managed service that allows you to create, publish, secure, and monitor REST, HTTP, and WebSocket APIs at scale.
2. **What types of APIs does API Gateway support?**
   → **REST APIs, HTTP APIs (lighter, cheaper), and WebSocket APIs** (for real-time apps).

3. **What are common use cases for API Gateway?**
   → Expose Lambda functions, connect to microservices, provide REST/HTTP APIs, real-time WebSocket APIs, handle request/response transformation, and secure backend services.

Core Features

4. **How do stages work in API Gateway?**
   → Stages (dev, test, prod) represent different environments. Each stage has its own URL, settings (caching, throttling, variables).
5. **What are stage variables?**
   → Key-value pairs used like environment variables (e.g., point to different Lambda aliases or endpoints).
6. **What is a deployment in API Gateway?**
   → A snapshot of API configuration that must be deployed to a stage to make it accessible.
7. **How does caching work in API Gateway?**
   → You can enable stage-level caching (0.5–237 GB). Responses are cached by request parameters/headers for a set TTL → reduces latency & backend load.
8. **How do you secure APIs in API Gateway?**
   → Methods: **IAM policies, Cognito User Pools, Lambda Authorizers, API Keys + usage plans, and WAF**.

Integrations

9. **What integration types are supported?**
   → **Lambda, HTTP(S) endpoints, Mock, AWS services (like S3, DynamoDB), and VPC Link (for private ALB/NLB)**.
10. **How does VPC Link work?**
    → Allows API Gateway to securely connect to private resources inside a VPC (via NLB/ALB).

Monitoring & Logging

11. **How do you monitor API Gateway?**
    → Using **CloudWatch metrics** (4XX/5XX errors, latency, count), **execution/access logs**, and **X-Ray tracing**.
12. **What's the difference between execution logs and access logs?**
    → Execution logs → internal API Gateway behavior (errors, mapping issues).
    Access logs → request/response details (IP, method, status code).

Deployments

13. **How do you do blue/green deployments with API Gateway?**
    → Use separate stages (blue, green) and switch base path mapping (custom domain).
14. **How do you do canary deployments?**
    → Enable canary release on a stage, send a % of traffic to new deployment, monitor, then shift more traffic gradually.

Advanced

15. **How do request/response transformations work?**
    → Using **mapping templates (VTL)** to reshape payloads, rename fields, inject context/stage variables.
16. **How do you handle request validation in API Gateway?**
    → Enable **request validators + models (JSON schema)** to validate parameters or request body before hitting backend.
17. **How do you troubleshoot 5XX errors?**
    → Check **CloudWatch logs & metrics**, validate Lambda response format ({ statusCode, body }), inspect mapping templates, and check integration timeouts.
18. **How do you set up a custom domain in API Gateway?**
    → Get SSL cert in ACM → create custom domain in API Gateway → map base path to stage → update Route 53 DNS.
19. **What's the difference between REST API and HTTP API in API Gateway?**

- REST API → more features (request validation, transformations, usage plans), but higher cost.
- HTTP API → faster, cheaper, simpler, good for Lambda/HTTP backends.

SNS

Fundamental Concepts

Q1: What is Amazon SNS?

→ Amazon Simple Notification Service (SNS) is a fully managed **pub/sub messaging service** that enables applications to send messages to multiple subscribers (like email, SMS, Lambda, SQS, mobile push) simultaneously.

Q2: How does Amazon SNS work?

→ SNS follows a **publisher-subscriber model**:

- Publishers send messages to an **SNS topic**.
- SNS **fans out** the message to all subscribers using configured protocols (HTTP, email, SMS, etc.).

Q3: What are SNS topics?

→ A **topic** is a logical access point for message publishing. Publishers send messages to a topic, and all subscribers to that topic receive the message.

Q4: What is an SNS subscription?

→ A **subscription** is an endpoint (e.g., email, Lambda, SQS, SMS, HTTP URL) that receives messages published to a topic.

Q5: What are the benefits of using Amazon SNS?

- Fully managed & serverless.
- Scalable to millions of subscribers.
- Multiple delivery protocols supported.
- Cost-effective (pay-per-request).
- Easy integration with other AWS services.

Technical Details and Use Cases

Q6: What protocols does SNS support?

→ **HTTP/S, Email, Email-JSON, SMS, Amazon SQS, AWS Lambda, Application (mobile push)**.

Q7: How do you secure messages in SNS?

- **Encryption at rest**: KMS-managed keys.
- **Encryption in transit**: TLS for message delivery.
- **Access control**: IAM policies & topic policies.
- **Message filtering**: Deliver only relevant messages to subscribers.

Q8: What is message filtering in SNS?

→ Subscribers define **filter policies** (based on message attributes). Only messages that match the filter are delivered.

✅ Example:

- Topic publishes events with attribute eventType.
- Subscriber wants only "orderCreated".
- They define filter policy:
- { "eventType": ["orderCreated"] }

Q9: How does SNS handle retries & failures?

- For **HTTP/S endpoints** → retries with exponential backoff.
- For **unreachable endpoints** → retries for hours, then message is dropped or sent to a **DLQ (dead-letter queue)** if configured.
- For **SQS/Lambda** → guaranteed delivery until acknowledgment.

Q10: Real-world scenario for SNS?

→ **E-commerce order system**: When an order is placed:

- SNS publishes event to OrderTopic.
- Subscribers:
  - Lambda (send confirmation email).
  - SQS (log order in data warehouse).
  - SMS (notify customer).

Q11: How does SNS integrate with AWS services?

- **SQS** → durable queue-based consumption.
- **Lambda** → event-driven processing.
- **CloudWatch** → trigger alerts/monitor failures.
- **Kinesis Firehose** → stream data to storage.

Q12: Difference between SNS & SQS?

- **SNS** = Pub/Sub → pushes messages to many subscribers.
- **SQS** = Queue → stores messages for consumers to poll.

👉 Use SNS when you need **fan-out** (1-to-many).

👉 Use SQS when you need **decoupled async processing** (1-to-1).

Advanced & Scenario-Based

Q13: How to prevent duplicate notifications in SNS?

- Use **MessageDeduplicationId** in FIFO SNS (FIFO topics introduced in 2020).
- Subscribers (e.g., SQS FIFO) ensure **exactly-once processing**.
- Or add **idempotency checks** in subscriber logic.

Q14: Scalability & Reliability features of SNS?

- Auto-scales to millions of messages/sec.
- Globally distributed endpoints (high availability).
- Redundancy across multiple AZs.
- Retry + DLQ support for reliability.

Q15: How to monitor SNS activity?

- **CloudWatch metrics**: NumberOfMessagesPublished, NumberOfNotificationsDelivered, NumberOfNotificationsFailed.
- **CloudWatch Logs**: Enable delivery status logging (e.g., for SMS/HTTP).
- **DLQ monitoring** for failed deliveries.

Q16: Key considerations for high-volume SNS design?

- Use **FIFO topics** if order & deduplication matter.
- Apply **filtering** to reduce unnecessary traffic.

- Enable **DLQs** for failed messages.
- Monitor retries to avoid subscriber overload.
- Consider **regional setup** for latency.

Q17: How to use SNS for mobile push notifications?

- Create an **SNS application** for each platform (APNS for iOS, GCM/FCM for Android).
- Register **device tokens** as platform endpoints.
- Publish messages to the **application endpoint** or topic → SNS delivers to mobile devices.

SES

Fundamental Concepts

Q1: What is Amazon SES? Explain its purpose and core functionalities.

→ Amazon Simple Email Service (SES) is a **cloud-based email sending service** designed for:

- **Transactional emails** (e.g., order confirmations, OTPs).
- **Marketing emails** (e.g., newsletters, promotions).
- **Bulk email campaigns**.

Core functionalities:

- Send, receive, and track emails.
- High deliverability with reputation management.
- Integrates with other AWS services (SNS, S3, Lambda, CloudWatch).

Q2: What are the key benefits of using Amazon SES?

- **Cost-effective**: Pay only for what you send.
- **Highly scalable**: Can handle millions of emails/day.
- **High deliverability**: Built-in support for SPF, DKIM, DMARC.
- **Reputation management**: Bounce & complaint handling, suppression list.
- **Security**: IAM policies, TLS, domain/email verification.

Q3: What types of emails can you send with SES?

- **Transactional** → OTPs, order confirmations.
- **Marketing** → promotions, newsletters.
- **Bulk** → campaign-style mass emails.

Q4: How does SES help with email deliverability and reputation?

- Supports **SPF, DKIM, DMARC** for authentication.
- **Feedback loops** for bounces & complaints.
- **Automatic suppression list** prevents sending to invalid/complaining addresses.
- IP warm-up & dedicated IPs for reputation management.

Technical Aspects

Q5: How do you send emails using SES?

- **SMTP interface** (works with standard mail clients).
- **AWS SDKs (boto3, etc.)** for programmatic sending.
- **SES API** for direct integration.

Q6: Explain domain/email verification in SES. Why is this necessary?

- Verification ensures you **own the sender identity**.
- You add a TXT record in DNS (for domains) or confirm via email link (for email).
- Prevents **email spoofing** and helps with **trust & deliverability**.

Q7: Difference between verified email vs. verified domain?

- **Verified email** → You can send emails *from that specific address*.
- **Verified domain** → You can send emails from **any address under that domain**.

Q8: How do you handle bounces & complaints in SES?

- **SNS integration**: SES sends notifications about bounces/complaints to an SNS topic.
- **Suppression list**: SES automatically blocks emails to problematic addresses.
- **Lambda/SQS**: Can process bounce/complaint events automatically.

Q9: How can you monitor email sending activity & performance in SES?

- **CloudWatch metrics** (send rate, delivery attempts, bounces, complaints).
- **SES event publishing** → SNS, Kinesis Firehose, CloudWatch Logs for detailed tracking.
- **Configuration sets** → capture metrics per campaign/application.

Q10: What are sending quotas in SES & how to increase them?

- Two main limits:
  - **Sending rate** → emails per second.
  - **Sending quota** → emails per 24 hours.
- By default → sandbox mode (only verified addresses).
- To increase → submit **SES sending limit request** in AWS Support.

Q11: How do you ensure security in SES?

- **IAM policies**: Restrict who/what can send emails.
- **TLS**: Enforce TLS for SMTP connections.
- **Domain authentication**: SPF/DKIM to prevent spoofing.
- **IAM roles + least privilege principle**.

Advanced Scenarios & Best Practices

Q12: Example scenario where you'd use SES?

→ **SaaS application sending signup confirmations & OTPs**.

- SES sends transactional emails.
- Bounce/complaint handling via SNS → Lambda → DynamoDB log.
- CloudWatch monitors deliverability.

Q13: How would you build a robust email notification system with SES + AWS services?

- **SES** → send emails.
- **SNS** → handle bounce/complaint notifications.
- **SQS/Lambda** → process notifications asynchronously.
- **CloudWatch** → monitor metrics.
- **S3** → archive logs & messages.

Q14: Best practices for maximizing email deliverability in SES?

- Verify domains & use **SPF/DKIM/DMARC**.
- Use **dedicated IPs** for large-scale sending.
- **Warm up IPs** gradually for reputation.
- Avoid spammy content & follow unsubscribe regulations.
- Monitor **bounce/complaint rates** (<0.1% complaint rate recommended).

Q15: How to troubleshoot if emails are landing in spam or not delivered?

- Check **SPF, DKIM, DMARC** setup.
- Verify **SES suppression list**.
- Inspect **CloudWatch metrics** for bounces/complaints.
- Review **email content** (avoid spam triggers).

- Warm up dedicated IPs if needed.

Q16: What are configuration sets in SES?

→ A **configuration set** is a set of rules that you can attach to emails for tracking & control.

Use cases:

- Event publishing (deliveries, bounces, opens, clicks).
- Apply sending limits per campaign.
- Monitor engagement metrics separately for different apps.

Q17: How do you handle email receiving with SES?

- Update **MX records** in DNS to point to SES.
- SES can then:
    - Store incoming emails in **S3**.
    - Trigger a **Lambda function** for processing.
    - Send to **SNS/SQS** for workflows.

Cloud Watch

Fundamental Concepts

Q1: What is AWS CloudWatch, and what is its primary purpose?

→ Amazon CloudWatch is a **monitoring and observability service** for AWS resources, applications, and on-premises systems.

Primary purpose: Collect, monitor, and analyze metrics/logs, set alarms, trigger automated actions, and visualize system health.

Q2: What are the core components of CloudWatch?

- **Metrics** → Numerical data (e.g., CPUUtilization, Latency).
- **Logs** → Log data from AWS resources & applications.
- **Alarms** → Threshold-based alerts on metrics.
- **Events (EventBridge)** → React to AWS resource state changes.
- **Dashboards** → Custom visualizations of metrics & logs.

Q3: Difference between CloudWatch vs. CloudTrail?

- **CloudWatch** → Performance & operational monitoring (metrics, logs, alarms, dashboards).
- **CloudTrail** → Governance, compliance, and security auditing (records **who did what** in AWS).

  👉 Use **CloudWatch** for *resource health/performance monitoring*.

  👉 Use **CloudTrail** for *auditing API calls & security*.

Q4: How does CloudWatch collect metrics from AWS services?

- AWS services automatically publish **default metrics** to CloudWatch at 1-min or 5-min intervals.
- You can also use the **CloudWatch Agent** (for EC2, on-premises, custom metrics).

Q5: What are custom metrics, and how can you publish them?

- **Custom metrics** → User-defined metrics (e.g., app response time, queue depth).
- Publish methods:
  - AWS SDK / CLI (put-metric-data).
  - CloudWatch Agent.
  - Embedded metric format (JSON in logs).

Metrics and Alarms

Q6: Common metrics to monitor for EC2 with CloudWatch?

- **CPUUtilization**
- **NetworkIn/Out**
- **DiskReadOps / DiskWriteOps**
- **StatusCheckFailed (Instance & System)**

Q7: How do you create a CloudWatch Alarm? Key components?

Steps: Choose metric → Define threshold → Configure evaluation period → Attach action (SNS,

Auto Scaling, etc.).
Key components:

- **Metric** (what to monitor).
- **Threshold** (e.g., CPU > 80%).
- **Period** (interval for evaluation).
- **Evaluation periods** (number of consecutive periods).
- **Actions** (send SNS, scale EC2, stop instance, etc.).

Q8: States of a CloudWatch Alarm?

- **OK** → Metric within threshold.
- **ALARM** → Metric breaches threshold.
- **INSUFFICIENT_DATA** → Not enough data.

Q9: How can CloudWatch Alarms automate actions?

- Trigger **Auto Scaling** (e.g., add/remove EC2).
- Notify via **SNS**.
- Execute **EC2 actions** (stop, reboot, terminate).
- Trigger **Lambda** for custom automation.

Q10: What is the concept of percentiles in CloudWatch metrics?

- Percentiles show distribution of values (e.g., **p90 latency** → 90% of requests are faster than this).
- Useful for understanding performance beyond averages.

Logs and Events

Q11: How do you enable CloudWatch Logs?

- **EC2** → Install/enable CloudWatch Agent.
- **Lambda** → Logs automatically published.
- **VPC Flow Logs** → Configure Flow Logs to CloudWatch.
- **ECS/EKS** → Configure logging drivers to send logs to CloudWatch.

Q12: How to analyze/filter log data in CloudWatch Logs Insights?

- Use **Logs Insights query language** to search, filter, and aggregate logs.
- Example: fields @timestamp, @message | filter @message like /ERROR/ | sort @timestamp desc.

Q13: What are CloudWatch Events (EventBridge)?

- Real-time stream of AWS service events.
- Used to **automate tasks** based on resource changes.
  Example: Trigger a Lambda when an EC2 instance state changes to "stopped".

Q14: Example scenario for CloudWatch Events?

→ Automatically trigger a Lambda to snapshot an EBS volume when an **EC2 instance is stopped**.

Advanced Topics & Scenarios

Q15: How would you monitor a serverless app (Lambda + API Gateway) with CloudWatch?

- **Metrics**:

    - Lambda → Invocations, Duration, Errors, Throttles.
    - API Gateway → Latency, 4XX/5XX errors, CacheHitRate.
- **Logs**: Enable Lambda & API Gateway logs in CloudWatch.
- **Alarms**: On error rates, latency, or throttles.
- **Dashboards**: Visualize metrics for end-to-end monitoring.

Q16: How would you troubleshoot app performance using CloudWatch?

- Start with **metrics** (CPU, memory, latency, error rate).
- Check **logs** for detailed error messages.
- Use **Logs Insights** to search failures.
- Set **alarms** for recurring issues.
- Correlate with **CloudTrail** if security/API misconfig is suspected.

Q17: How to monitor custom apps or on-prem with CloudWatch Agent?

- Install CloudWatch Agent.
- Configure JSON to collect CPU, memory, disk, or app logs.
- Publish to CloudWatch as custom metrics/logs.

Q18: How do you use CloudWatch Dashboards?

- Build **custom dashboards** with multiple widgets (graphs, numbers, text).
- Monitor multiple resources (EC2, RDS, Lambda) in one view.
- Share dashboards for team visibility.

Q19: How to ensure cost-effective logging/monitoring with CloudWatch?

- Use **log retention policies** (auto-expire old logs).
- Store long-term logs in **S3** instead of CloudWatch.
- Filter/aggregate logs before ingestion.
- Monitor only critical metrics with **alarms**.

Q20: How does CloudWatch integrate with other AWS services?

- **Auto Scaling** (scale based on alarms).
- **SNS** (notifications).
- **SQS/Lambda** (event-driven automation).
- **EventBridge** (workflow automation).
- **CloudTrail** (security + compliance correlation).

Cloud Font

Beginner Level

1. What is Amazon CloudFront?
Amazon CloudFront is a **Content Delivery Network (CDN)** that delivers content (websites, videos, APIs) with **low latency** using a global network of edge locations.
👉 Example: Netflix uses CloudFront to stream movies faster worldwide.

2. How does CloudFront deliver content faster?
It stores (caches) content in **edge locations** near users. Requests are served from the nearest edge instead of the origin.
👉 Example: A user in India watching an e-commerce site sees cached product images from a Mumbai edge location instead of US servers.

3. What is an "origin" in CloudFront?
Origin = the main source of content CloudFront fetches from.
👉 Examples: **S3 bucket (static website), EC2 instance (custom app), or an ALB.**

4. What are cache behaviors in CloudFront?
Cache behaviors define **how CloudFront handles requests** (e.g., caching, forwarding headers, routing by URL path).
👉 Example: /images/* can be cached longer, while /api/* always fetches fresh data.

5. Purpose of Geo Restriction in CloudFront?
Restricts access to content based on users' location.
👉 Example: A streaming platform blocks access to videos in countries without a license.

6. How can you invalidate cached content?
By creating an **invalidation request** (e.g., /index.html). Used when content changes but cache still serves old data.
👉 Example: Updating a company homepage after a product launch.

Intermediate Level

1. Difference: CloudFront vs S3 vs EC2

- **S3** = object storage.
- **EC2** = compute (runs apps).
- **CloudFront** = accelerates delivery of content from S3/EC2 globally.
  👉 Example: Store website files in S3 → CloudFront serves them quickly worldwide.

2. How to secure content?

- **Signed URLs/Cookies** = restrict who can access.
- **OAI/OAC** = restrict S3 bucket so only CloudFront can fetch objects.
  👉 Example: Paid e-learning site gives signed URL valid for 1 hour.

3. CloudFront Functions vs Lambda@Edge

- **CloudFront Functions** = lightweight, for viewer requests/responses (URL rewrites, redirects).
- **Lambda@Edge** = full compute, supports complex logic at edge.
  👉 Example: Redirect HTTP→HTTPS (Functions) vs. Resize image on demand (Lambda@Edge).

4. Optimizing for dynamic content

- Use **Cache-Control headers**, short TTLs, forward only required headers/query strings.
  👉 Example: Caching API responses for 60s to balance freshness and speed.

5. Setup CloudFront for S3 website

- Create S3 bucket → enable static hosting.
- Create CloudFront distribution → set S3 as origin.
- Point DNS (Route53) to CloudFront.
  👉 Example: Hosting a personal portfolio website.

6. CloudFront + WAF

CloudFront integrates with **AWS WAF** to block SQL injection, XSS, bad IPs.
👉 Example: Protecting an online store from bot traffic and DDoS.

Advanced Level

1. Advanced caching strategies

- **Cache key normalization** = ignore query strings/headers that don't affect response.
- **Origin Shield** = extra caching layer near origin to reduce load.
  👉 Example: Ignore utm_campaign in cache key so marketing URLs don't break cache.

2. Troubleshooting performance issues

- Check **CloudWatch metrics** (cache hit ratio, latency, 4xx/5xx).
- Use **CloudFront access logs** for request analysis.
  👉 Example: High 5xx errors → origin overloaded.

3. Handling different content types

- Use multiple cache behaviors:
  - /static/* → cache long.
  - /api/* → short TTL, forward headers.
  - /videos/* → streaming optimized.
    👉 Example: E-commerce site serving product images, APIs, and video demos.

4. Real-time logs use case

Track live requests for **fraud detection or analytics**.
👉 Example: Monitor suspicious login attempts in real time.

5. Blue/Green or A/B testing

- Use **weighted cache behaviors** or Lambda@Edge to split traffic between versions.
  👉 Example: Send 10% users to new homepage design.

6. Cost implications & optimization

- Costs based on data transfer & requests.
- Optimize by: using **cache TTLs**, compressing objects, limiting invalidations.

  👉 Example: Reduce cost by caching images longer instead of hitting S3 often.

7. Role in Serverless (API Gateway + Lambda)

CloudFront sits in front of **API Gateway + Lambda** to provide **global caching, security, and DDoS protection**.

👉 Example: A serverless REST API used by a mobile app worldwide.

Certificate Manger

Beginner Level

1. Purpose of ACM?
Manages SSL/TLS certificates to enable **HTTPS** and secure connections without manual certificate handling.
👉 Example: Enabling HTTPS on an e-commerce website hosted with ALB.

2. Types of certificates ACM can provision?

- **Public certificates** (internet-facing apps).
- **Private certificates** (internal apps, within VPC).
  👉 Example: Public cert for myshop.com, private cert for internal HR app.

3. How ACM simplifies management?
Handles provisioning, renewal, deployment automatically → no manual file uploads.
👉 Example: Automatic renewal for CloudFront SSL certs every year.

4. Public vs Private certificates?

- **Public**: Trusted by browsers, for public websites.
- **Private**: Used inside organization (not trusted by browsers).
  👉 Example: Public for myapp.com, private for internal API in VPC.

5. How to request a public certificate?
Request via **ACM Console/CLI** → validate domain (DNS/Email).
👉 Example: Adding a DNS CNAME record in Route 53 to prove domain ownership.

6. What is domain validation?
Process of proving domain ownership via **DNS record** or **Email approval**.
👉 Example: ACM asks you to create a CNAME record in Route53 to validate www.example.com.

7. Which AWS services integrate with ACM?
CloudFront, ALB/NLB, API Gateway, Elastic Beanstalk, App Runner.
👉 Example: Attach ACM cert to ALB serving a WordPress app on EC2.

Intermediate Level

1. Certificate renewal in ACM?
ACM auto-renews public certs before expiry. Private certs depend on Private CA settings.
👉 Example: Public cert on CloudFront renewed automatically without downtime.

2. Importing third-party certificate?
Upload certificate, private key, and chain via console/CLI.
👉 Example: Importing DigiCert SSL cert for a legacy banking application.

3. Steps to associate ACM cert with ELB/CloudFront

- Request/import cert in ACM.
- Go to ELB/CloudFront settings → choose **HTTPS listener** → select ACM cert.
  👉 Example: Adding HTTPS to an Application Load Balancer for a web app.

4. Benefits of private certificates in VPC?
Secures internal apps/services (low cost, automatic lifecycle management).
👉 Example: Encrypt traffic between microservices in a private VPC.

5. Certificate revocation in ACM?
For private certs → ACM PCA can revoke. Public certs cannot be revoked by ACM.
👉 Example: Revoking a compromised internal API cert issued by PCA.

6. Role of AWS PCA with ACM?
ACM + PCA = issue and manage private certs at scale for internal apps.
👉 Example: Issue TLS certs for IoT devices in a factory network.

7. Security considerations

- Use **least privilege IAM** for cert access.
- Monitor expiration.
- Use OIDC/SAML for internal apps with private certs.
  👉 Example: Restrict only DevOps team to attach ACM certs to ELBs.

Advanced Level

1. ACM + CloudFormation integration
You can define ACM certs in CloudFormation templates for auto-provision + attach to resources.
👉 Example: CI/CD pipeline automatically provisions SSL cert for each new CloudFront distribution.

2. When to use PCA with ACM (internal apps)?
Scenario: Corporate intranet with microservices across VPCs.
Steps:

- Setup AWS PCA.
- Use ACM to issue private certs.
- Attach certs to internal ALBs/NLBs.
  👉 Example: HR payroll system accessible only inside the corporate VPN.

3. Troubleshooting validation/deployment issues

- Check domain validation DNS records.
- Use **ACM console → Validation status**.

- CloudWatch logs for CloudFront/ALB errors.
  👉 Example: CloudFront stuck on “Pending validation” because DNS CNAME was not added.

4. Multi-account / multi-region implications

- ACM certs are **region-specific** (except CloudFront).
- Need to request/import in each region/account.
  👉 Example: Banking app running in us-east-1 and ap-south-1 → request certs separately.

5. Monitoring cert health/expiration

- **ACM Console** shows expiry.
- CloudWatch Events + SNS alerts before expiry.
  👉 Example: Send Slack alert 30 days before a cert expires.

6. Certificate pinning relevance

Pinning = client trusts only a specific cert/key → reduces MITM risk.

👉 Example: A mobile banking app pins ACM-issued cert to avoid fake cert attacks.

7. ACM for microservices architecture

- Use ACM (with PCA) to issue private certs for **service-to-service TLS**.
- Use Service Mesh (App Mesh) or mTLS for authentication.
  👉 Example: A Kubernetes cluster where all microservices communicate securely via ACM-managed private certs.

Route 53

Beginner Level

1. What is Amazon Route 53, and why is it used?
Route 53 is a **highly available DNS (Domain Name System) service** by AWS. It routes end users to applications.
👉 Example: Mapping www.myshop.com to an ELB running an e-commerce site.

2. What are the main features of Route 53?

- Domain registration
- DNS routing (public & private hosted zones)
- Health checks & failover
- Integration with AWS services
  👉 Example: Register mywebsite.com, host DNS, and point traffic to CloudFront.

3. What is a hosted zone in Route 53?
Container for DNS records of a domain.
👉 Example: A hosted zone for myapp.com stores A, CNAME, MX records.

4. Public vs Private hosted zone?

- **Public**: Resolves domains on the internet.
- **Private**: Resolves names within a VPC.
  👉 Example: Public → myapp.com for customers, Private → db.internal.local in VPC.

5. Types of records in Route 53?
A, AAAA, CNAME, MX, TXT, Alias, etc.
👉 Example: A record → point app.myshop.com to EC2 IP.

6. What is TTL in Route 53?
Time DNS resolvers cache a record before re-checking.
👉 Example: Set TTL=60s for frequently updated API endpoint.

7. How does Route 53 provide high availability?
Globally distributed DNS servers + health checks + failover.
👉 Example: If one server fails, DNS query is routed to nearest healthy server.

8. Can Route 53 be a registrar too?
Yes, it can register and manage domains.
👉 Example: Buy myportfolio.com directly in Route 53.

9. Difference between Alias and CNAME?

- Alias → AWS resources (ELB, S3, CloudFront), works at root domain.
- CNAME → maps one name to another, not allowed at root.
  👉 Example: Alias for myapp.com → CloudFront, CNAME for blog.myapp.com → Medium.

10. How does Route 53 integrate with CloudFront/ELB?
You create an Alias record pointing to CloudFront/ELB.
👉 Example: www.myapp.com Alias → CloudFront distribution.

Intermediate Level

11. Routing policies in Route 53?

- Simple, Weighted, Latency, Failover, Geolocation, Geoproximity, Multi-Value Answer.
  👉 Example: Weighted routing for A/B testing new app versions.

12. Latency-based routing?
Routes users to the region with lowest latency.
👉 Example: US users → us-east-1, India users → ap-south-1.

13. Weighted routing?
Distributes traffic across resources based on percentages.
👉 Example: 80% traffic to old site, 20% to new site.

14. Geolocation vs Geoproximity?

- Geolocation = based on user's location (country/continent).
- Geoproximity = based on resource location + bias setting.
  👉 Example: Direct EU users to Frankfurt (Geolocation), shift more EU traffic to Paris with bias (Geoproximity).

15. Failover routing?
Primary resource active, secondary used if primary fails.
👉 Example: If primary EC2 in us-east-1 is down → traffic goes to backup EC2 in us-west-2.

16. What is a health check?
Monitors endpoints (HTTP/HTTPS/TCP) and integrates with routing.
👉 Example: Health check fails for API → Route 53 removes unhealthy endpoint.

17. Can Route 53 work with private VPCs?
Yes, using **Private Hosted Zones**.
👉 Example: db.local resolving only inside company VPC.

18. Configure Route 53 with S3 static site?
Create S3 static website → Route 53 Alias record pointing to S3 bucket.
👉 Example: myblog.com → S3 bucket hosting HTML pages.

19. DNS failover in Route 53?
Uses health checks to reroute traffic to a healthy endpoint.
👉 Example: If Mumbai server fails, failover to Singapore server.

20. Migrate DNS records from another provider?
Export/import records → create hosted zone → update domain registrar's name servers to Route

53.
👉 Example: Moving DNS from GoDaddy to Route 53.

Advanced Level

21. Multi-region app design with Route 53?
Use latency or geolocation routing + health checks to distribute traffic globally.
👉 Example: Banking app active-active in US & EU with latency routing.

22. Route 53 health checks vs ELB health checks?
- Route 53 → DNS level, independent of AWS.
- ELB → instance-level, inside AWS.
  👉 Example: Route 53 can check a 3rd-party endpoint, ELB cannot.

23. What is Route 53 Resolver?
Service for DNS resolution in hybrid environments (VPC ↔ on-prem).
👉 Example: On-prem users resolve internal.aws.local hosted in VPC.

24. Route 53 for hybrid cloud?
Use **Resolver inbound/outbound endpoints** to resolve DNS across AWS + on-prem.
👉 Example: AWS app needs to resolve on-prem Oracle DB hostname.

25. Split-horizon DNS in Route 53?
Same domain resolves differently based on requester's location (internal vs external).
👉 Example: app.company.com → public IP for customers, private IP for employees.

26. DNSSEC support in Route 53?
Yes, secures domain records with cryptographic signatures.
👉 Example: Prevents attackers from spoofing login.mybank.com.

27. Disaster recovery with Route 53?
Use failover routing + health checks to redirect to backup region.
👉 Example: DR site in Singapore becomes active if India region fails.

28. Route 53 + AWS Global Accelerator?
Global Accelerator routes traffic over AWS backbone (TCP/UDP), while Route 53 routes via DNS.
👉 Example: Gaming app uses Global Accelerator for low latency, Route 53 for DNS resolution.

29. Monitoring DNS queries?
Enable Route 53 query logs to CloudWatch or S3.
👉 Example: Track suspicious DNS traffic patterns for a DDoS attack.

30. Cost optimization strategies?
- Use Alias (free queries) instead of CNAME.
- Consolidate hosted zones.

- Optimize TTL to reduce lookups.
  👉 Example: Set higher TTL for static images to reduce DNS costs.

IAM

Beginner Level

1. What is AWS IAM, and why is it used?
IAM manages **access to AWS services/resources** securely using users, groups, and roles.
👉 Example: Allow a developer to access only S3 but not EC2.

2. What are IAM users, groups, and roles?
- **User**: Individual identity with credentials.
- **Group**: Collection of users with same policies.
- **Role**: Temporary access, assumed by users/services.
  👉 Example: EC2 instance assumes an IAM role to read S3 data.

3. What is an IAM policy?
JSON document defining **permissions** (Allow/Deny).
👉 Example: A policy granting s3:GetObject on mybucket/*.

4. Difference between IAM roles and IAM users?
- **User** → Long-term credentials.
- **Role** → Temporary credentials, assumed when needed.
  👉 Example: Lambda assumes a role to write logs to CloudWatch.

5. What are IAM access keys?
A pair (AccessKeyId, SecretAccessKey) for programmatic access (CLI/SDK).
👉 Example: DevOps engineer using AWS CLI with access keys.

6. What is the IAM root user?
The initial AWS account user with **full permissions**. Should be protected (MFA).
👉 Example: Use root only to close account or set up billing.

7. What is MFA in IAM?
Multi-Factor Authentication adds a second factor (e.g., OTP, hardware token) for login security.
👉 Example: Admin logs in with password + Google Authenticator code.

8. What are IAM managed policies?
- **AWS-managed** (predefined by AWS).
- **Customer-managed** (custom).
  👉 Example: Attach AmazonS3FullAccess to a group.

9. What is the IAM principle of least privilege?
Grant only the **minimum permissions** needed.
👉 Example: Grant EC2 operator only ec2:StartInstances, not full EC2 access.

10. How is IAM global?
IAM is **not region-specific**, applies across the whole AWS account.
👉 Example: User created in IAM can access S3 buckets in any region (if allowed).

Intermediate Level

11. Difference between inline and managed policies?

- **Inline** → Embedded in a single user/role.
- **Managed** → Reusable, attachable across users/roles.
  👉 Example: Inline for one-off special case, managed for team-wide S3 access.

12. What is an IAM trust policy?
Defines **who can assume a role**.
👉 Example: Trust policy allows EC2 to assume a role.

13. How do IAM roles work with EC2?
EC2 assumes role → gets temporary credentials from STS → access AWS services.
👉 Example: EC2 app reading from DynamoDB without storing credentials.

14. What is AWS STS (Security Token Service)?
Issues temporary credentials for IAM roles, federated users.
👉 Example: SSO login to AWS via corporate Active Directory.

15. How do you restrict access to a specific IP address in IAM?
Use a **condition** in policy with aws:SourceIp.
👉 Example: Allow S3 access only from office IP.

16. What is IAM identity federation?
Allows external identities (Google, Active Directory, Okta) to access AWS using roles.
👉 Example: Employees log in with corporate SSO to AWS console.

17. How do IAM policies get evaluated?

- Default = Deny
- Explicit Allow = grants access
- Explicit Deny = overrides everything
  👉 Example: User allowed S3 read but explicit deny → blocked.

18. What are service-linked roles?
Special IAM roles **pre-created by AWS services** for integration.
👉 Example: CloudWatch creates service-linked role for alarms.

19. What is IAM credential report?
CSV report listing all users and their credentials (passwords, access keys, MFA).
👉 Example: Check inactive users or unused access keys.
20. How do you enforce strong password policies in IAM?
Configure IAM account password policy (length, complexity, rotation).
👉 Example: Enforce 12-char password + rotate every 90 days.

Advanced Level

21. How to use IAM with cross-account access?
- Create a role in Account A.
- Allow Account B to assume it via trust policy.
  👉 Example: Centralized logging account → other accounts assume role to push logs.

22. What are permission boundaries in IAM?
Advanced guardrails: define the **maximum permissions** a role/user can get.
👉 Example: A developer can only create resources in dev-* environment.

23. Difference: SCPs (Service Control Policies) vs IAM policies?
- **IAM policies** → Apply to users/roles in an account.
- **SCPs** → Apply at AWS Organizations level (across accounts).
  👉 Example: SCP prevents creating resources outside us-east-1.

24. What is IAM Access Analyzer?
Finds **resources shared externally** (like S3 buckets or KMS keys).
👉 Example: Detect if S3 bucket is accidentally public.

25. How do you troubleshoot IAM "Access Denied" errors?
- Check IAM policy simulator
- Verify explicit denies
- Check SCPs & permission boundaries
  👉 Example: User denied S3 access due to missing permission in boundary policy.

26. What are IAM session policies?
Policies passed at **STS AssumeRole** call to further restrict temporary creds.
👉 Example: Contractor assumes role but restricted to one S3 bucket.

27. How do you secure IAM best practices?
- Use MFA on root & admin
- Rotate keys
- Least privilege
- Monitor with CloudTrail
  👉 Example: Detect root login without MFA using CloudWatch alarms.

28. How does IAM integrate with AWS Organizations?
Organizations use **SCPs** to limit IAM permissions across accounts.
👉 Example: Block IAM users from disabling CloudTrail in child accounts.

29. Explain identity-based vs resource-based policies.

- **Identity-based** → Attached to users, groups, roles.
- **Resource-based** → Attached directly to resources.

  👉 Example: S3 bucket policy (resource-based) allows cross-account access.

30. How would you design IAM for a large enterprise?

- Use SSO + Identity Federation
- Organize accounts with Organizations & SCPs
- Use roles for cross-account access
- Apply least privilege + permission boundaries

  👉 Example: Enterprise with 50 AWS accounts uses Okta SSO + IAM roles for developers.

Event Bridge

Beginner Level

1. What is AWS EventBridge?
A **serverless event bus service** that connects applications using events. It routes events from AWS services, SaaS apps, or custom apps.
👉 Example: When a new file is uploaded to S3, EventBridge triggers a Lambda.

2. Difference between EventBridge and CloudWatch Events?
EventBridge is the **evolved version** of CloudWatch Events with advanced routing, schema registry, SaaS integration.
👉 Example: EventBridge integrates with Salesforce → CloudWatch Events does not.

3. What is an event in EventBridge?
A JSON record that represents a change in a system.
👉 Example: {"detail-type": "EC2 Instance State-change Notification", "state": "stopped"}

4. What is an event bus in EventBridge?
A pipeline that receives and routes events.
👉 Example: Default bus for AWS services, custom bus for internal apps.

5. What are event rules in EventBridge?
Rules filter and route events to targets.
👉 Example: A rule sends EC2 stop events to SNS for alerting.

6. What are event targets?
Destinations for events (Lambda, SNS, SQS, Step Functions, etc.).
👉 Example: Event triggers Step Function to start an order workflow.

7. What is the schema registry in EventBridge?
Stores event structures (schemas) for discovery and validation.
👉 Example: Developer downloads schema for PutItem events from DynamoDB.

8. Use case of EventBridge?

- Event-driven microservices
- Automation
- SaaS integration
  👉 Example: Automatically create a Jira ticket when a support email arrives via EventBridge.

Intermediate Level

9. How does EventBridge deliver events reliably?

- At-least-once delivery
- Retry mechanism (24 hours)

- DLQ support via SQS/SNS
  👉 Example: Failed Lambda invocation goes to DLQ for later processing.

10. Difference between EventBridge and SNS?

- **SNS** = Pub/Sub messaging, fan-out, push notifications.
- **EventBridge** = Advanced routing, event transformation, SaaS integrations.
  👉 Example: Use SNS for mobile push alerts, EventBridge for microservice orchestration.

11. Can EventBridge filter events?

Yes, rules use **event patterns** (JSON match).

👉 Example: Match only S3 events where eventName = PutObject.

12. How do you transform events in EventBridge?

Using **input transformer** to modify event before sending to target.

👉 Example: Extract only username from event and send to Lambda.

13. How do you secure EventBridge?

- IAM policies
- Resource policies (cross-account)
- Encryption (KMS)
  👉 Example: Allow only Account B to put events on Account A's bus.

14. Can EventBridge integrate with SaaS apps?

Yes, EventBridge integrates with SaaS like Zendesk, Datadog, PagerDuty.

👉 Example: When a PagerDuty incident is triggered, EventBridge routes it to Slack.

15. What is a partner event bus?

Dedicated event bus for SaaS partner apps.

👉 Example: Zendesk event bus for support ticket creation events.

16. How do you replay events in EventBridge?

Use **Event Replay** (archive + replay).

👉 Example: Reprocess old order events after fixing a bug in Lambda.

17. What is cross-account event delivery?

EventBridge can send/receive events between AWS accounts.

👉 Example: Central logging account collects events from all AWS accounts.

18. How does EventBridge scale?

EventBridge is fully managed and auto-scales to handle millions of events/sec.

👉 Example: Streaming IoT device events at scale.

19. How do you monitor EventBridge?

Use **CloudWatch metrics** (Invocations, DeadLetterInvocations, FailedInvocations).

👉 Example: CloudWatch alarm on high FailedInvocations.

20. How does EventBridge integrate with Step Functions?
Events trigger Step Functions workflows for orchestration.
👉 Example: "OrderPlaced" event → Step Function → payment + shipping microservices.

Advanced Level

21. Difference between EventBridge vs SQS + SNS architecture?

- **SNS + SQS** = basic pub/sub + queue.
- **EventBridge** = event routing, filtering, transformation, SaaS integration.
  👉 Example: Complex event-driven app → EventBridge; simple fan-out → SNS+SQS.

22. How would you design an event-driven microservices system with EventBridge?

- Each service publishes events to EventBridge bus.
- Rules route events to specific consumers.
  👉 Example: E-commerce → "OrderPlaced" → payment, inventory, notification services.

23. How to handle event duplication in EventBridge?

- At-least-once delivery means duplicates possible → design idempotent consumers.
  👉 Example: Payment service ignores duplicate OrderPaid events.

24. What are advanced event patterns in EventBridge?
Supports:

- Matching arrays
- Wildcards
- Complex JSON paths
  👉 Example: Match EC2 events where state = stopped OR terminated.

25. How does EventBridge support event-driven integration across regions?
You can **send events cross-region** using event bus policies.
👉 Example: US-East events trigger processing in AP-South.

26. How do you archive and replay events in EventBridge?
Enable event archiving → store events → replay later.
👉 Example: Replay all OrderPlaced events after fixing downstream bug.

27. EventBridge vs Kinesis?

- **EventBridge**: Discrete events, routing, SaaS integration.
- **Kinesis**: Real-time streaming, analytics, ordering.
  👉 Example: EventBridge for app events, Kinesis for video/log streams.

28. How do you implement DLQ (Dead Letter Queue) in EventBridge?
Attach **SQS/SNS as DLQ** for failed event deliveries.
👉 Example: Failed Lambda events sent to SQS → DevOps retries later.

29. What are EventBridge Pipes?
Feature to connect event sources → filter/transform → send to targets directly.
👉 Example: Pipe DynamoDB stream → filter inserts → send to Lambda.

30. Design a multi-account, multi-region event bus system with EventBridge.

- Centralized event bus in one account.
- Other accounts forward events via **cross-account policies**.
- Multi-region failover with replication.

  👉 Example: Enterprise logging all security events into a single account for compliance.