1. **\* I have an application, how you will setup a VPC ? what all the components you use ?**

You have an application that you want to host securely and reliably in AWS. You’ll set up a VPC (Virtual Private Cloud) to provide isolated networking for your app.

🧩 Step 1: Create a VPC

* VPC (Virtual Private Cloud) – Your isolated network in AWS.
  + Example: CIDR block 10.0.0.0/16

🧩 Step 2: Create Subnets

You’ll divide your VPC into subnets for better organization and security.

| Type | Use | Example CIDR | Zone |
| --- | --- | --- | --- |
| Public Subnet(s) | For internet-facing resources like Load Balancers, Bastion Host | 10.0.1.0/24 | ap-south-1a |
| Private Subnet(s) | For application servers, backend APIs | 10.0.2.0/24 | ap-south-1a |
| Database Subnet(s) | For databases (RDS, etc.) | 10.0.3.0/24 | ap-south-1b |

👉 You typically create subnets in multiple Availability Zones (AZs) for high availability.

🧩 Step 3: Configure Internet Connectivity

* Internet Gateway (IGW):
  + Attach it to the VPC to allow outbound and inbound internet traffic for public subnets.
* Route Table (Public):
  + Add route: 0.0.0.0/0 → IGW
  + Associate with Public Subnets

🧩 Step 4: Configure Private Connectivity

* NAT Gateway:
  + Placed in a Public Subnet.
  + Allows instances in Private Subnets to access the internet (for updates, etc.) but not be exposed to inbound traffic.
* Private Route Table:
  + Add route: 0.0.0.0/0 → NAT Gateway
  + Associate with Private Subnets

🧩 Step 5: Add Security

* Security Groups (SG):
  + Virtual firewall for instances.
  + Example: Allow HTTP/HTTPS for web servers; allow DB connections from app servers only.
* Network ACLs (NACLs):
  + Optional, for stateless subnet-level security.
  + Add inbound/outbound rules to further restrict traffic.

🧩 Step 6: Add Supporting Components

* Bastion Host (Jump Box):
  + Optional, for SSH access to private instances.
* Elastic Load Balancer (ALB/NLB):
  + ALB for HTTP/HTTPS applications.
  + NLB for high-performance TCP/UDP traffic.
* Auto Scaling Group (ASG):
  + For application servers, ensures scalability and fault tolerance.
* RDS / DynamoDB / S3:
  + Database or storage layers.

🧩 Step 7: Configure DNS and Access

* Route 53:
  + To map your domain to the Load Balancer.
* VPC Endpoints (Optional):
  + For private access to AWS services (S3, DynamoDB) without using the internet.

✅ Summary of Components

| Component | Purpose |
| --- | --- |
| VPC | Isolated virtual network |
| Subnets (Public/Private) | Logical separation of resources |
| Internet Gateway | Internet access for public subnets |
| NAT Gateway | Outbound internet for private subnets |
| Route Tables | Traffic routing configuration |
| Security Groups / NACLs | Instance & subnet-level protection |
| Load Balancer (ALB/NLB) | Distribute incoming traffic |
| Auto Scaling Group | Scalability & HA |
| RDS/Database Subnet Group | Backend database tier |
| Bastion Host | Secure admin access |
| Route 53 | DNS resolution |
| VPC Endpoints | Private AWS service access |

1. **\* Where you will set the Route Table ?**

A Route Table is always associated with one or more Subnets inside your VPC.  
It defines how traffic is routed — i.e., where packets go next after leaving a subnet.

🧭 1. Public Route Table

* Purpose: For subnets that need internet access (like web servers or load balancers).
* Association: Associate it with Public Subnets.
* Route Example:
* Destination Target
* 10.0.0.0/16 local
* 0.0.0.0/0 igw-xxxxxxxx

→ 0.0.0.0/0 route points to Internet Gateway (IGW).

🧭 2. Private Route Table

* Purpose: For internal subnets (like app or DB servers) that don’t have direct internet access.
* Association: Associate it with Private Subnets.
* Route Example:
* Destination Target
* 10.0.0.0/16 local
* 0.0.0.0/0 nat-xxxxxxxx

→ 0.0.0.0/0 route points to NAT Gateway (for outbound internet only).

🧭 3. Database Route Table (Optional)

* Sometimes you create a separate route table for DB subnets (for tighter control).
  + Usually, no default route (0.0.0.0/0) — fully private, internal-only access.

✅ Summary Table

| Route Table | Associated With | Default Route Target | Internet Access |
| --- | --- | --- | --- |
| Public | Public Subnets | Internet Gateway (IGW) | ✅ Yes |
| Private | App Subnets | NAT Gateway | 🚫 Inbound, ✅ Outbound |
| DB | DB Subnets | None (local only) | 🚫 No |

1. **\* Services used in AWS ?**

1. Compute

* EC2 (Elastic Compute Cloud): Virtual servers to run applications.
* Lambda: Run code without managing servers (serverless).
* ECS / EKS: Run Docker containers (ECS = AWS-managed, EKS = Kubernetes).
* Auto Scaling: Automatically increase or decrease EC2 instances based on load.
* Elastic Beanstalk: Easy deployment for web apps — AWS manages servers for you.

2. Storage

* S3 (Simple Storage Service): Object storage for files, backups, and data.
* EBS (Elastic Block Store): Block storage for EC2 instances (like a hard drive).
* EFS (Elastic File System): Shared file system across multiple EC2 instances.
* Glacier: Archival, low-cost storage for long-term backups.

3. Networking

* VPC (Virtual Private Cloud): Isolated network environment in AWS.
* Subnets: Divide your VPC into public/private sections.
* Route 53: DNS and domain name management.
* Load Balancers (ALB/NLB/Gateway LB): Distribute traffic across servers.
* CloudFront: Content Delivery Network (CDN) for faster global access.
* Direct Connect / VPN: Secure private connection to AWS.

4. Security & Identity

* IAM (Identity and Access Management): Control user access and permissions.
* KMS (Key Management Service): Manage encryption keys.
* Secrets Manager / Parameter Store: Securely store passwords, tokens, etc.
* WAF (Web Application Firewall): Protect web apps from common attacks.
* Shield: DDoS protection.

5. Databases

* RDS (Relational Database Service): Managed databases (MySQL, PostgreSQL, etc.).
* DynamoDB: NoSQL database.
* Aurora: High-performance managed relational DB.
* ElastiCache: In-memory caching (Redis/Memcached).
* Redshift: Data warehouse for analytics.

6. Management & Monitoring

* CloudWatch: Monitor AWS resources and set alarms.
* CloudTrail: Track API activity in your AWS account.
* Config: Track configuration changes in resources.
* Systems Manager: Manage and automate tasks across servers.

7. Developer Tools

* CodeCommit: Git-based repository.
* CodeBuild: Build and test code automatically.
* CodeDeploy: Automate code deployment.
* CodePipeline: CI/CD automation.

8. Machine Learning / AI

* SageMaker: Build, train, and deploy ML models.
* Rekognition: Image and video analysis.
* Comprehend: NLP (text analysis).
* Lex: Chatbots (like Alexa).
* Polly: Text-to-speech.

9. Migration & Transfer

* DMS (Database Migration Service): Move databases to AWS.
* Snowball / Snowmobile: Physical data transfer devices.
* Migration Hub: Centralized migration tracking.

10. Analytics

* Athena: Query data directly from S3 using SQL.
* EMR: Big data processing (Hadoop, Spark).
* Kinesis: Real-time data streaming.
* Glue: ETL (Extract, Transform, Load) service.
* QuickSight: Data visualization.

1. **\* How can you make any S3 bucket as private ?**

Option 1: Using AWS Management Console

1. Go to the S3 console → https://s3.console.aws.amazon.com/s3
2. Select the bucket you want to make private.
3. Go to the Permissions tab.
4. Check these settings:
   * Block Public Access (bucket settings):
     + Click Edit → Turn ON all options:
       - Block all public access
       - Block public ACLs
       - Block new public bucket policies
       - Block cross-account public access
     + Click Save changes
   * Bucket Policy:
     + Remove any statements that allow public access (like "Principal": "\*")
   * Access Control List (ACL):
     + Ensure no public access is given (like “Everyone” or “Any authenticated AWS user”).

✅ After these steps, your bucket will be private.

Option 2: Using AWS CLI

You can use the following command to block public access:

aws s3api put-public-access-block \

--bucket your-bucket-name \

--public-access-block-configuration \

"BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=true"

Option 3: Using Bucket Policy

To explicitly deny all public access:

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "DenyPublicAccess",

"Effect": "Deny",

"Principal": "\*",

"Action": "s3:\*",

"Resource": [

"arn:aws:s3:::your-bucket-name",

"arn:aws:s3:::your-bucket-name/\*"

],

"Condition": {

"Bool": { "aws:SecureTransport": "false" }

}

}

]

}

1. **\* Cross region replication ? how you apply it ? will it copy automatically ?**

What is Cross-Region Replication (CRR)?

Cross-Region Replication (CRR) is an Amazon S3 feature that automatically copies objects from a bucket in one AWS Region to another bucket in a different Region.

It helps with:

* Disaster recovery
* Compliance (keeping backups in another region)
* Lower latency for global users

⚙️ How to Apply (Enable) Cross-Region Replication

You can enable CRR from the AWS Management Console or CLI.

Using AWS Console:

1. Go to S3 → Source Bucket → Management tab
2. Under Replication rules, click Create replication rule
3. Give a name to the rule
4. Enable the rule (choose *Enabled*)
5. Under Source, choose:
   * *This bucket* or specific prefix/folder
6. Under Destination, choose:
   * Another bucket in a different region
   * The destination bucket must have Versioning enabled
7. Optionally choose:
   * IAM Role (S3 can create one for you)
   * Whether to replicate delete markers, existing objects, etc.
8. Click Save

🪄 Will it copy automatically?

✅ Yes, but only new or changed objects after CRR is enabled.

By default:

* CRR does not copy existing objects — only new uploads.
* However, you can choose “Replicate existing objects” (if available) to copy existing data as a one-time bulk replication.

📦 Important Prerequisites

* Versioning must be enabled on both source and destination buckets.
* Destination bucket must be in another region.
* Proper IAM Role with replication permissions must exist.
* Block Public Access or KMS encryption can affect replication if not properly configured.

💡 Example (AWS CLI):

aws s3api put-bucket-replication \

--bucket source-bucket-name \

--replication-configuration '{

"Role": "arn:aws:iam::123456789012:role/s3-replication-role",

"Rules": [

{

"Status": "Enabled",

"Priority": 1,

"DeleteMarkerReplication": { "Status": "Disabled" },

"Filter": {},

"Destination": {

"Bucket": "arn:aws:s3:::destination-bucket-name",

"StorageClass": "STANDARD"

}

}

]

}'

🧠 Summary

| Feature | Description |
| --- | --- |
| Purpose | Replicate objects across regions |
| Automatic? | Yes — for new objects after setup |
| Existing data replicated? | No (unless you choose to replicate existing) |
| Requirements | Versioning enabled on both buckets |
| Use case | Backup, disaster recovery, compliance |

1. **\* S3 transfer accereration ?**

S3 Transfer Acceleration (TA) — is a feature in Amazon S3 that speeds up uploads and downloads of objects to S3 over long distances by using Amazon CloudFront’s globally distributed edge locations.

🔍 How it Works

* When you enable Transfer Acceleration on an S3 bucket:
  + Data is first uploaded to the nearest AWS edge location (using CloudFront’s global network).
  + Then it is securely routed through AWS’s optimized network backbone to your S3 bucket.
* This results in faster data transfer, especially when users are geographically distant from the bucket’s region.

🧭 Example

Let’s say your S3 bucket is in us-east-1 (N. Virginia) but your users are in India.  
Without acceleration → data travels over the public internet.  
With acceleration → data goes to the nearest edge location (like Mumbai or Singapore) → travels inside AWS’s fast network → reaches the S3 bucket faster.

⚙️ How to Enable

1. Go to S3 Console → Select your bucket.
2. Click Properties tab.
3. Scroll to Transfer acceleration → Click Edit → choose Enable.
4. After enabling, use the accelerated endpoint URL:
5. https://<bucketname>.s3-accelerate.amazonaws.com

Example:

https://mybucket.s3-accelerate.amazonaws.com

💡 CLI Example

To upload using Transfer Acceleration:

aws s3 cp file.txt s3://mybucket/ --endpoint-url https://s3-accelerate.amazonaws.com

📊 Use Cases

* Large file uploads from far geographic locations.
* Applications requiring low latency and high throughput transfers.
* Media uploads, backups, or data ingestion from global users.

⚠️ Points to Note

* Extra cost per GB for accelerated transfers.
* Works best when there’s significant distance between client and S3 region.
* Must be enabled per bucket.
* Not supported for all AWS Regions (e.g., some China regions).

1. **\* Read replica**

✅ Read Replica — is a copy of a primary database (DB) that allows read-only queries, helping to improve performance and scalability of your database workloads.

sPurpose

Read replicas are used to:

* Offload read traffic from the main (primary) database.
* Increase performance for read-heavy applications.
* Provide high availability and disaster recovery options.

How It Works

1. You have a Primary DB instance (also called source DB) — it handles both reads and writes.
2. You create a Read Replica — it gets a real-time asynchronous copy of data from the primary DB.
3. Applications can read from the replica, but cannot write to it.
4. If the primary DB fails, a read replica can be promoted to become a new primary.

Supported Databases (in AWS RDS)

* Amazon RDS for:
  + MySQL
  + MariaDB
  + PostgreSQL
  + Oracle
  + SQL Server (limited)
  + Amazon Aurora (automatic replication)

How to Create a Read Replica (AWS Console)

1. Go to RDS Console → Databases.
2. Select your primary DB instance.
3. Choose Actions → Create read replica.
4. Specify:
   * Replica instance name
   * Region (same or different)
   * DB instance class, storage, etc.
5. Click Create read replica.

CLI Example

aws rds create-db-instance-read-replica \

--db-instance-identifier mydb-replica \

--source-db-instance-identifier mydb-primary

Cross-Region Read Replica

You can create read replicas in a different AWS region.

* Useful for global applications or disaster recovery.

Promotion

If the primary DB fails:

aws rds promote-read-replica --db-instance-identifier mydb-replica

Now it becomes a standalone primary DB, allowing read and write operations.

Key Points

| Feature | Description |
| --- | --- |
| Replication type | Asynchronous |
| Writes | Not allowed |
| Reads | Allowed |
| Promotion | Possible (manual or automatic in Aurora) |
| Use case | Scaling reads, backup, failover |

Example Use Case

* Your application has heavy read traffic (like a news site or analytics app).
* You create 3 read replicas and use a load balancer to distribute reads among them.
* The primary handles writes only.

1. **\* Suppose you create Auto scaling group and in launch conf you specify as 3 instances, then the all the 3 instances are running in full capacity ?**

Scenario

You created an Auto Scaling Group (ASG)  
and in the Launch Configuration, you set:

Desired capacity = 3

So, the Auto Scaling Group will launch 3 EC2 instances.

Now, do all 3 run at full capacity?

No — not necessarily.

The desired capacity (3) just means:

* The ASG will maintain 3 instances running at all times.
* It doesn’t mean the instances are running at *100% CPU or resource utilization*.

Explanation

* Each EC2 instance runs your application or workload.
* How much “capacity” (CPU, memory, network, etc.) they use depends on your application’s traffic or load.
* If the traffic is low, all 3 instances may be mostly idle.
* If traffic increases, the CPU/memory usage may go up — Auto Scaling can then launch more instances (scale out).

ASG Parameters

| Parameter | Meaning |
| --- | --- |
| Minimum capacity | The minimum number of instances to keep running. |
| Desired capacity | The current number of instances you want running (default start count). |
| Maximum capacity | The upper limit of instances ASG can create. |

📊 Example

| Parameter | Value |
| --- | --- |
| Min | 2 |
| Desired | 3 |
| Max | 6 |

➡️ ASG starts with 3 instances.  
➡️ If load increases, CloudWatch triggers scale out to maybe 4–6 instances.  
➡️ If load drops, it may scale in to 2–3 instances.

🔁 Conclusion

So:

❌ Not all 3 instances will always run at “full capacity”.  
✅ They will all run, but their utilization depends on workload and scaling policies.

1. **\* will the applciation fails or it will be running ? Do you think AWS will provide new instance ?**

Scenario Recap

You have an Auto Scaling Group (ASG) with:

* Desired capacity: 3 instances
* Min: 3
* Max: 6

Now, imagine one of the 3 EC2 instances fails (for example: system crash, AZ failure, or termination).

✅ What Happens Next

1. ASG continuously monitors instance health (via EC2 status checks or ELB health checks).
2. If it detects an instance as unhealthy or stopped,  
   → ASG terminates that bad instance.
3. Then it automatically launches a new instance to replace it.

So, the total count (3 instances) stays constant — application remains running.

⚙️ In Short

| Condition | What AWS Does | Application Impact |
| --- | --- | --- |
| One instance fails | ASG launches a new instance automatically | ✅ App keeps running (minimal downtime) |
| Whole AZ fails | If ASG spans multiple AZs, it launches new instances in other healthy AZs | ✅ App stays available |
| All instances unhealthy (and only one AZ used) | Temporary downtime until ASG launches new healthy instances | ⚠️ Possible short interruption |

📊 Example

If you deployed a web app in ASG with desired=3  
and 1 instance crashes, AWS will:

* Detect unhealthy state
* Terminate the failed instance
* Launch a new one (using the Launch Template/Configuration)

You’ll again have 3 running healthy instances.

🧩 Key AWS Feature

This ability comes from:

Auto Healing (a built-in feature of Auto Scaling Groups)

✅ Final Answer

The application will not fail entirely — Auto Scaling will automatically replace the failed instance to maintain the desired count.  
✅ AWS provides a new instance automatically.

1. **\* How to you manage credentials ?**

Credential management means storing, using, and protecting your AWS access keys, passwords, and tokens securely, so that unauthorized users cannot access your AWS resources.

1. Use IAM Roles (Best Practice)

* Never hardcode access keys in code or configuration files.
* Instead, assign IAM Roles to:
  + EC2 instances
  + Lambda functions
  + ECS tasks
* The role automatically provides temporary credentials to the resource.

Example:  
When an EC2 instance has an IAM role attached, the application can call AWS APIs without storing credentials — AWS automatically handles authentication via the instance metadata service.

2. Use AWS Secrets Manager / Systems Manager Parameter Store

* Store database passwords, API keys, tokens securely.
* They provide:
  + Encryption with AWS KMS
  + Access control via IAM policies
  + Automatic rotation of secrets

Example:  
Store your RDS password in Secrets Manager, and fetch it securely in your application at runtime.

3. Use AWS CLI or SDK Profiles (for developers)

* Store credentials in ~/.aws/credentials using:
* aws configure
* Use multiple named profiles for different accounts (e.g., dev, test, prod).
* Always ensure this file is not shared or uploaded to GitHub or public repos.

4. Use Temporary Security Credentials

* Use STS (Security Token Service) to generate temporary access keys with limited time validity.
* Ideal for short-term access (for CI/CD or federated logins).

Example:

aws sts get-session-token --duration-seconds 3600

5. Enforce IAM Best Practices

* Apply least privilege — give only necessary permissions.
* Rotate credentials regularly.
* Use MFA (Multi-Factor Authentication) for root and IAM users.
* Disable or delete unused access keys.

6. Never Do These

❌ Don’t hardcode AWS keys in:

* Source code
* Config files
* Docker images
* Git repositories

❌ Don’t share IAM user credentials.

🧠 Summary Table

| Best Practice | Purpose |
| --- | --- |
| IAM Roles | Grant temporary access to AWS services without hardcoding keys |
| Secrets Manager / SSM | Securely store passwords and API keys |
| STS Tokens | Provide short-term access |
| IAM Policies | Restrict access with least privilege |
| MFA | Extra protection for users |
| Key Rotation | Reduce risk if a key is compromised |

Example Answer (Interview Style)

“I manage AWS credentials using IAM Roles and AWS Secrets Manager.  
I never hardcode access keys; instead, I attach roles to EC2 or Lambda.  
For sensitive data like database passwords, I use Secrets Manager with KMS encryption.  
I also follow best practices like enabling MFA, key rotation, and least privilege IAM policies.”

1. **\* If credentials icon is not available? What you will do?**

Context

You’re in the AWS Management Console (for example, IAM, EC2, or S3), and you expect to see a “Security credentials” tab or icon —  
but it’s missing or greyed out.

Possible Causes & Fixes

1. You’re Not Logged in as the Root User

* The “Security credentials” option (for access keys, passwords, etc.) is available only for:
  + Root users, and
  + IAM users (for their own credentials)
* If you’re using an IAM role or federated SSO login, that option won’t appear.

✅ Fix:

* Log in as the IAM user directly (not as a role).
* Or ask the admin to manage credentials for you.

2. Missing IAM Permissions

If your IAM policy doesn’t allow credential management, you won’t see the option.

✅ Fix:

* Ask the AWS Administrator to add permissions such as:
* {
* "Effect": "Allow",
* "Action": [
* "iam:CreateAccessKey",
* "iam:ListAccessKeys",
* "iam:DeleteAccessKey",
* "iam:UpdateAccessKey"
* ],
* "Resource": "arn:aws:iam::<account-id>:user/<username>"
* }
* After permissions are updated, sign out and sign in again.

3. You’re in the Wrong Console Page

Sometimes users look in EC2 → IAM Role page or S3, expecting “credentials” there.  
But credentials are managed only from:

AWS Console → IAM → Users → (your username) → Security credentials

✅ Fix:

* Go to the IAM service directly.
* Open your user profile → Security credentials tab.

4. Using AWS Organization Account

If you’re in an AWS Organization, your admin may have disabled access key creation or restricted console options.

✅ Fix:

* Contact your Org admin to confirm if credential management is restricted by Service Control Policy (SCP).

5. Browser / UI Issue

Sometimes the console UI glitches due to browser cache or outdated session.

✅ Fix:

* Clear cache or use Incognito mode.
* Try another browser or region endpoint (e.g., us-east-1).

🧩 Example Interview Answer

“If the credentials icon is missing, I’ll first check if I’m logged in as the correct IAM user.  
If I’m assuming a role or my policy doesn’t include IAM actions, I’ll contact the admin to enable permissions.  
I’ll also verify from the IAM console directly under my user’s Security credentials **tab, and ensure there’s no Organization-level restriction.”**

1. **\* You are working on EC2 and that goes down and how do you resolve this ?**

Step 1: Identify the Problem

First, confirm why the EC2 instance is down —  
it could be stopped, terminated, crashed, or unreachable.

Check from the AWS Console:

* Go to EC2 → Instances → Instance state
* Note the state:
  + 🟥 *Stopped*
  + 🟧 *Terminated*
  + 🟢 *Running but not reachable*

Possible Scenarios & Solutions

🔹 Scenario 1: Instance is “Stopped”

Cause: Stopped manually, by auto-scaling policy, or due to scheduled maintenance.

Solution:

* Simply start it again:
* aws ec2 start-instances --instance-ids <instance-id>
* If it fails to start, check:
  + IAM permissions
  + Billing (account suspension)
  + Quota limits (EC2 limits reached)

🔹 Scenario 2: Instance is “Terminated”

Cause:

* Auto-scaling policy replaced it
* Spot instance interruption
* Accidental termination

Solution:

* You can’t restart a terminated instance ❌
* Instead:
  1. Launch a new instance from the same AMI (Amazon Machine Image).
  2. Attach the old EBS volume (if it exists) to recover data.
  3. Reassociate Elastic IP (if used).

🔹 Scenario 3: Instance is “Running” but Not Reachable

Symptoms: SSH/RDP not working, timeout, or unreachable website.

Possible causes and fixes:

| Cause | Fix |
| --- | --- |
| 🔸 Security Group blocking inbound SSH (port 22) or RDP (3389) | Edit SG → allow inbound from your IP |
| 🔸 Network ACL blocking inbound/outbound | Check VPC → Network ACL rules |
| 🔸 Route table missing IGW route | Ensure 0.0.0.0/0 → Internet Gateway |
| 🔸 OS or application crash | Use EC2 serial console or SSM Session Manager |
| 🔸 EBS volume corruption | Detach volume → attach to another instance → repair filesystem |

🔹 Scenario 4: High CPU or Memory Usage

Symptoms: Slow response, freezing, unresponsive SSH.

Solution:

* Use CloudWatch metrics to check CPU, memory, or I/O.
* If it’s overloaded:
  + Reboot the instance
  + Resize it (Change instance type to larger size)
  + Use Auto Scaling or Load Balancer to distribute traffic

🔹 Scenario 5: Underlying Host Failure (AWS-side)

Sometimes AWS may detect a hardware failure on the host where your EC2 runs.

Solution:

* AWS automatically tries to recover or migrate your instance.
* You can also manually recover it:
  + Go to EC2 → Actions → Instance settings → Recover instance

*(Works for EBS-backed instances only.)*

🧰 Useful Commands

Check instance status:

aws ec2 describe-instance-status --instance-id <id>

Reboot instance:

aws ec2 reboot-instances --instance-ids <id>

Start instance:

aws ec2 start-instances --instance-ids <id>

Step-by-Step Troubleshooting Summary

| Step | Action |
| --- | --- |
| 1️⃣ | Check instance state (running, stopped, terminated) |
| 2️⃣ | Review EC2 and System Status checks |
| 3️⃣ | Verify security group, NACL, and route table |
| 4️⃣ | Check CloudWatch metrics for CPU/network |
| 5️⃣ | Use EC2 serial console or SSM to log in if SSH fails |
| 6️⃣ | Recover or relaunch instance if necessary |
| 7️⃣ | Attach old EBS volume to recover data if needed |

🧠 Example Interview Answer

“If my EC2 instance goes down, I first check its state in the AWS console.  
If it’s stopped, I start it; if it’s terminated, I relaunch from the last AMI and reattach the EBS volume.  
If it’s running but unreachable, I verify the security group, NACL, and route table.  
For system issues, I use CloudWatch logs, EC2 serial console, or SSM Session Manager.  
This ensures minimal downtime and quick recovery.”

1. **\* You want to give a access to a EC2 and you have only pvt key and how do you enable him to access the EC2 ?**

Scenario

* You have an existing EC2 instance (Linux/Ubuntu, for example).
* You have the private key (.pem) used to connect.
* You need to give another person (user B) access to this instance.
* But you don’t want to share your private key (and shouldn’t).

🛠️ Steps to Give Access Securely

🔹 Step 1: SSH into the EC2 Instance (using your private key)

Use your existing key to connect:

ssh -i mykey.pem ec2-user@<public-ip>

🔹 Step 2: Add a New User (for the other person)

Once inside the EC2 instance:

sudo adduser newuser

Then add that user to the sudo group (optional):

sudo usermod -aG sudo newuser

🔹 Step 3: Create SSH Directory for the New User

sudo mkdir /home/newuser/.ssh

sudo chmod 700 /home/newuser/.ssh

🔹 Step 4: Add Their Public Key

Ask the other person to send their public key (not private key).  
Example: id\_rsa.pub

Then copy that public key into:

sudo nano /home/newuser/.ssh/authorized\_keys

Paste their public key in that file.

Set permissions:

sudo chmod 600 /home/newuser/.ssh/authorized\_keys

sudo chown -R newuser:newuser /home/newuser/.ssh

🔹 Step 5: Test Access

Now the other person can connect using their own private key:

ssh -i theirkey.pem newuser@<public-ip>

✅ They now have access — securely — without sharing your key.

If You Only Have the Private Key (and They Have No Key Pair Yet)

You can help them generate one:

1. On their local machine:
2. ssh-keygen -t rsa -b 4096 -C "newuser@example.com"

This creates:

* + Private key → ~/.ssh/id\_rsa
  + Public key → ~/.ssh/id\_rsa.pub

1. They send you only the id\_rsa.pub file.
2. You follow Steps 3–4 above to place that public key on the EC2 instance.

⚠️ Never Do This

🚫 Never share your private key (.pem) with anyone.  
🚫 Never email or upload the .pem file to others.  
🚫 Avoid adding multiple people under the same SSH key — always use individual keys.

🧠 Example Interview Answer

“I’d SSH into the EC2 using my private key, create a new Linux user for them, and place their public key in the user’s ~/.ssh/authorized\_keys file.  
This way, they can log in with their own private key securely, and I don’t have to share mine.  
I’d also make sure proper permissions are set and restrict SSH access via security groups.”

1. **\* EKS ?**

Excellent — this one’s about EKS (Elastic Kubernetes Service) — a very common AWS interview question for DevOps and cloud engineers.

Let’s go step-by-step

EKS stands for Amazon Elastic Kubernetes Service.  
It is a fully managed Kubernetes service provided by AWS that makes it easy to run, manage, and scale containerized applications using Kubernetes on AWS — without needing to install or manage your own Kubernetes control plane.

In Simple Terms

EKS = Managed Kubernetes by AWS  
AWS takes care of the control plane (master nodes) — you manage only the worker nodes and pods.

How EKS Works

1. Control Plane

* Managed by AWS.
* Includes:
  + kube-apiserver
  + etcd (Kubernetes database)
  + Controller Manager
  + Scheduler
* Automatically scales and is highly available across multiple AZs.

2. Worker Nodes

* EC2 instances (or Fargate) that actually run your containers/pods.
* You manage these nodes or use AWS Fargate for serverless nodes.

3. Networking

* Uses VPC CNI plugin to assign ENIs (Elastic Network Interfaces) and IPs to pods.
* Integrates with AWS Load Balancers (ALB/NLB) for traffic routing.

🏗️ Architecture Overview

+------------------------------------------------------+

| Amazon EKS Control Plane |

| (Managed by AWS - API Server, etcd, Controllers) |

+------------------------------------------------------+

| |

----------|--------------------|-------------

| |

+------------------+ +------------------+

| Worker Node 1 | | Worker Node 2 |

| (EC2 / Fargate) | | (EC2 / Fargate) |

| Pods, Services | | Pods, Services |

+------------------+ +------------------+

Key Components in EKS

| Component | Description |
| --- | --- |
| Cluster | A group of worker nodes + managed control plane |
| Node Group | A set of EC2 instances running Kubernetes pods |
| Pod | Smallest deployable unit in Kubernetes |
| Deployment / Service | How you define and expose apps |
| kubeconfig | Used to connect kubectl to EKS cluster |

How to Create an EKS Cluster (Basic Steps)

Option 1: AWS Console

1. Go to EKS → Create cluster
2. Configure:
   * Cluster name
   * IAM role
   * VPC and subnets
3. Wait for control plane creation
4. Add Node Group (EC2 or Fargate)
5. Connect via kubectl

Option 2: AWS CLI / eksctl

Using eksctl (simplest way):

eksctl create cluster \

--name my-cluster \

--region us-east-1 \

--nodegroup-name my-nodes \

--nodes 3 \

--node-type t3.medium \

--managed

Benefits of EKS

| Benefit | Description |
| --- | --- |
| Fully Managed Control Plane | AWS handles upgrades, HA, and patches |
| Scalable | Auto-scaling for both pods and nodes |
| Secure | Integrated with IAM, VPC, KMS, CloudTrail |
| Flexible | Supports EC2, Fargate, on-prem via EKS Anywhere |
| Open Source Compatibility | 100% upstream Kubernetes — no lock-in |

Integration with Other AWS Services

* ECR → for container images
* CloudWatch → for logging/metrics
* ALB/NLB → for load balancing
* IAM → for access control
* Auto Scaling → for worker nodes
* Fargate → for serverless compute

EKS vs ECS (Common Interview Follow-up)

| Feature | EKS | ECS |
| --- | --- | --- |
| Type | Kubernetes-based | AWS-native container service |
| Management | Kubernetes API | AWS Console/CLI |
| Portability | Multi-cloud, open-source | AWS-only |
| Control Plane | Managed by AWS (EKS) | Fully managed by AWS |
| Complexity | Higher | Easier |

Example Interview Answer

“EKS is Amazon’s managed Kubernetes service that lets you run containerized workloads on AWS without managing the control plane.  
It integrates with IAM, CloudWatch, and ELB, and supports both EC2 and Fargate worker nodes.  
I can deploy applications using kubectl, and AWS handles the Kubernetes master nodes, upgrades, and scaling automatically.”

1. **\* VPC ? suppose I have a server, need to download the packages from the internet, but sever should not be accessed directly from outside ? how you can do that**

“Suppose I have a server in a VPC that needs to download packages from the internet,  
but it should not be accessed directly from outside (internet) —  
how can you design that?”

🧠 Goal

* ✅ Server can access the internet (outbound) → to download updates/packages
* 🚫 Server cannot be accessed from the internet (inbound) → for security

💡 Solution: Use a Private Subnet + NAT Gateway

🏗️ Architecture Overview

+------------------------------+

| Internet |

+--------------+---------------+

|

(Public IP)

|

+----------------+

| NAT Gateway |

| (in Public Subnet)

+----------------+

|

(Private traffic routing)

|

+------------------------------+

| Private Subnet (No IGW) |

| EC2 Instance (no public IP)|

+------------------------------+

|

+----------------+

| Route Table |

| 0.0.0.0/0 → NAT Gateway |

+----------------+

⚙️ Step-by-Step Setup

🔹 1. Create a VPC

* Example CIDR: 10.0.0.0/16

🔹 2. Create 2 Subnets

| Subnet | Type | Example CIDR | Purpose |
| --- | --- | --- | --- |
| Public Subnet | Public | 10.0.1.0/24 | For NAT Gateway |
| Private Subnet | Private | 10.0.2.0/24 | For your EC2 instance |

🔹 3. Create and Attach an Internet Gateway

* Attach IGW to your VPC.
* Public subnet’s route table → add route:
* Destination: 0.0.0.0/0
* Target: Internet Gateway

🔹 4. Launch a NAT Gateway in the Public Subnet

* Allocate an Elastic IP.
* NAT Gateway uses that IP to access internet.

🔹 5. Update Route Table of Private Subnet

Add a route:

Destination: 0.0.0.0/0

Target: NAT Gateway

✅ Now, private EC2 instance can connect to the internet *through* the NAT Gateway.  
But no one from the internet can connect *to* that EC2.

🔹 6. Launch Your EC2 in the Private Subnet

* Do not assign a public IP.
* Access it using:
  + Bastion Host (Jump Server) in the public subnet
  + or SSM Session Manager (recommended for secure access)

💻 Result

| Direction | Allowed? | Path |
| --- | --- | --- |
| EC2 → Internet (outbound) | ✅ Yes | via NAT Gateway |
| Internet → EC2 (inbound) | ❌ No | blocked (no public IP, no IGW) |

🧩 Alternative: Using AWS Systems Manager (SSM)

If you don’t want a bastion host:

* Attach SSM agent + IAM role to EC2.
* Connect via Session Manager (no SSH needed, no inbound ports open).

✅ Secure  
✅ No NAT needed (for updates you still need internet access)  
✅ No public IP exposure

🧠 Example Interview Answer

“I’ll place the server in a private subnet with no public IP.  
To allow it to download packages, I’ll route its outbound traffic through a NAT Gateway in a public subnet.  
This gives the server internet access for updates, but it can’t be accessed directly from the internet.  
For admin access, I can use a bastion host or AWS Systems Manager Session Manager.”

1. **\* 2 AWS accounts, my EC2 of 1 account should talk to my another EC2 ? How ?**

🧠 Goal

You have:

* Account A → EC2 instance (say EC2-A)
* Account B → EC2 instance (say EC2-B)  
  and you want network communication between them (for example, SSH, API calls, database access, etc.)

🏗️ There are 3 main ways to achieve this depending on the use case 👇

Option 1️⃣ — VPC Peering (Most Common)

🧩 Concept:

VPC Peering connects two VPCs privately using AWS internal network,  
even if they are in different accounts or regions (with some limits).

✅ Steps:

1. Create a VPC Peering Connection
   * From Account A → Request peering with Account B’s VPC ID.
2. Accept the Peering Request
   * Log into Account B → Accept the connection.
3. Update Route Tables
   * In both VPCs, add routes to each other’s CIDR ranges through the peering connection.
   * Example:
     + VPC A route: 10.1.0.0/16 → Peering Connection
     + VPC B route: 10.0.0.0/16 → Peering Connection
4. Modify Security Groups
   * Allow inbound traffic from the peer VPC’s CIDR range.

✅ Result:  
EC2-A in Account A can talk to EC2-B in Account B privately (no internet).

🚫 Limitations:

* No transitive routing (A↔B, B↔C doesn’t mean A↔C).
* Both sides need proper route tables & security rules.

Option 2️⃣ — PrivateLink (VPC Endpoint Service)

🧩 Concept:

If you want one-way private access (for example, Account A’s app accesses Account B’s service/API securely),  
you can use PrivateLink.

✅ Steps:

1. In Account B, create a VPC Endpoint Service using a Network Load Balancer in front of EC2-B.
2. Share the Service Name with Account A.
3. In Account A, create an Interface VPC Endpoint and connect to that service.
4. Update Security Groups and NLB targets.

✅ Result:  
EC2-A privately accesses the application in Account B over AWS backbone,  
without peering or internet gateway.

🏆 Best for:

* One-directional access
* SaaS or API-style communication
* Secure and controlled sharing

Option 3️⃣ — Transit Gateway (For Complex Multi-Account Setup)

If you have many VPCs across multiple accounts, instead of multiple peerings,  
you can use a Transit Gateway (TGW).

✅ Steps:

1. Create a Transit Gateway in one account.
2. Share it across accounts using AWS Resource Access Manager (RAM).
3. Attach both VPCs (Account A and B) to the TGW.
4. Update routes to send traffic via TGW.

✅ Result:  
All attached VPCs (across accounts) can communicate via TGW.  
Easier than managing multiple peerings.

Example Interview Answer

“If I need EC2 in one AWS account to communicate with another EC2 in a different account,  
I can use VPC Peering to create a private connection between the two VPCs and update the route tables and security groups accordingly.  
For one-way, service-based communication, I’d use PrivateLink,  
and for multiple accounts or large-scale setups, I’d use Transit Gateway.”

⚡ Summary

| Method | Use Case | Direction | Internet Needed? | Notes |
| --- | --- | --- | --- | --- |
| VPC Peering | Simple 2-way communication | Two-way | ❌ | Common & easy |
| PrivateLink | One-way service access | One-way | ❌ | Secure & scalable |
| Transit Gateway | Multi-account network | Two-way | ❌ | Enterprise-grade |

1. **\* IAM ? I have 10 instances, how do you give access to user for only 5 resources ?**

Goal

You want:

* A user to access only 5 specific EC2 instances
* And not access the remaining 5

🔹 Step-by-Step Solution

1️⃣ Identify the 5 instances

Let’s say the instance IDs are:

i-01, i-02, i-03, i-04, i-05

2️⃣ Create a Custom IAM Policy

AWS IAM policies can restrict permissions by resource ARN.  
So you specify only those 5 instance ARNs in the policy.

✅ Example Policy (Allow access only to 5 instances)

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ec2:StartInstances",

"ec2:StopInstances",

"ec2:DescribeInstances"

],

"Resource": [

"arn:aws:ec2:ap-south-1:123456789012:instance/i-01",

"arn:aws:ec2:ap-south-1:123456789012:instance/i-02",

"arn:aws:ec2:ap-south-1:123456789012:instance/i-03",

"arn:aws:ec2:ap-south-1:123456789012:instance/i-04",

"arn:aws:ec2:ap-south-1:123456789012:instance/i-05"

]

},

{

"Effect": "Allow",

"Action": "ec2:DescribeInstances",

"Resource": "\*"

}

]

}

📝 Explanation:

* Allows start/stop actions only on the listed 5 instances.
* Allows DescribeInstances for all — so they can see info, but not control others.

3️⃣ Attach the Policy

Attach this policy to:

* A specific IAM User, or
* A Group (if multiple users need same limited access), or
* An IAM Role (if access is via EC2 or a service)

4️⃣ (Optional) Add Tag-Based Access Control

Instead of listing instance IDs, you can use tags for flexibility.

Example:

Tag the 5 instances with:

Key: Access

Value: Allowed

Then create a tag-based IAM policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ec2:StartInstances",

"ec2:StopInstances",

"ec2:DescribeInstances"

],

"Resource": "\*",

"Condition": {

"StringEquals": {

"ec2:ResourceTag/Access": "Allowed"

}

}

}

]

}

✅ Now the user can operate only on instances that have the tag Access=Allowed.

💡 Best Practice

Use tag-based IAM policies — scalable and easier to maintain than listing individual ARNs.

💻 Example Interview Answer

“I’d create a custom IAM policy that grants EC2 access only to the specific instance ARNs or to instances with a specific tag like Access=Allowed.  
Then, I’d attach that policy to the IAM user or group.  
This way, the user can manage only those 5 instances and not the rest.”

⚡ Summary Table

| Approach | Method | Scalable? | Use Case |
| --- | --- | --- | --- |
| Resource-based | List specific instance ARNs | ❌ | Few resources |
| Tag-based | Use tags + IAM conditions | ✅ | Many instances, dynamic setup |

1. **\* Cross region Route53 ?**

Scenario

You have applications deployed in multiple AWS regions — for example:

* One EC2 or load balancer in us-east-1 (Virginia)
* Another in ap-south-1 (Mumbai)

You want Route 53 to:

* Route users to the nearest or healthy region
* Or fail over to another region if one region goes down

Goal

Use Route 53 to provide cross-region DNS routing —  
meaning your DNS can intelligently direct traffic between AWS regions.

🏗️ How You Can Do It

Option 1️⃣ — Latency-Based Routing (LBR)

Concept:

Users are automatically routed to the region that provides lowest network latency.

✅ Example:

| Region | Resource | Route 53 Record Type |
| --- | --- | --- |
| us-east-1 | ALB / EC2 public IP | A / Alias |
| ap-south-1 | ALB / EC2 public IP | A / Alias |

Each record is marked as Latency record:

* us-east-1 → Latency region = US East (N. Virginia)
* ap-south-1 → Latency region = Asia Pacific (Mumbai)

🔹 Route 53 checks latency for each region and routes users accordingly.

✅ Use when you want users served from the closest region for best performance.

Option 2️⃣ — Failover Routing

🧩 Concept:

Use Primary and Secondary endpoints in different regions.  
If primary fails, Route 53 automatically sends traffic to the secondary.

✅ Steps:

1. Create two records for the same domain:
   * Primary: points to region 1 (e.g., us-east-1)
   * Secondary: points to region 2 (e.g., ap-south-1)
2. Associate health checks to monitor the primary endpoint.

If health check fails → Route 53 routes traffic to the secondary (cross-region failover).

✅ Use when you want disaster recovery setup across regions.

Option 3️⃣ — Geolocation Routing

🧩 Concept:

Send users to different regions based on their geographic location (continent/country).

Example:

* India traffic → ap-south-1
* US traffic → us-east-1
* Europe traffic → eu-west-1

✅ Use when you want to localize user experiences or meet data residency needs.

Option 4️⃣ — Multi-Value Answer Routing

Route 53 can return multiple IPs from different regions for load distribution or redundancy.  
If one IP’s health check fails, Route 53 removes it from DNS responses.

🧰 Health Checks

To enable cross-region failover, you must:

* Create Route 53 health checks for endpoints (ALB, EC2 public IP, or API)
* Health checks monitor HTTP/HTTPS or TCP status
* Combine with Failover or Multi-value routing

💻 Example Interview Answer

“For cross-region routing, I can use Route 53’s Latency-based routing to send users to the nearest region or Failover routing to route traffic to another region if one region goes down.  
Each region would host a load balancer or EC2 endpoint, and Route 53 health checks monitor availability.  
This ensures high availability and low latency across regions.”

⚡ Summary Table

| Routing Policy | Purpose | Regions | Needs Health Check? |
| --- | --- | --- | --- |
| Latency-Based | Send to nearest region | Multi | Optional |
| Failover | Active-Passive cross-region DR | Two | ✅ Yes |
| Geolocation | Route by user’s location | Multi | Optional |
| Multi-Value | Load distribution + health checks | Multi | ✅ Yes |

1. **\* Health checks in Route53**

Definition

A Route 53 health check continuously monitors the health and availability of your application’s endpoints (like EC2, Load Balancer, API, or website).

If the endpoint fails the check, Route 53 can:

* Stop routing traffic to that endpoint
* Failover to another region or backup resource

Purpose

Used for:

* ✅ High availability
* ✅ Cross-region failover
* ✅ Automatic DNS routing based on health

`

⚙️ How It Works

1. Route 53 Health Checkers (AWS global monitoring servers) periodically send requests to your endpoint.
2. If the endpoint fails to respond correctly (or at all), it is marked unhealthy.
3. Route 53 updates the DNS response — removing or redirecting traffic from that unhealthy endpoint.

🧩 Types of Health Checks

| Type | Description | Example |
| --- | --- | --- |
| Endpoint health check | Checks the health of a web server or resource using HTTP, HTTPS, or TCP | Check EC2 or ALB endpoint |
| CloudWatch alarm check | Uses a CloudWatch metric instead of direct network requests | Check CPUUtilization < 80% |
| Calculated health check | Combines multiple health checks using AND/OR logic | Healthy only if both region checks pass |

Protocols Supported

* HTTP
* HTTPS (with or without SSL validation)
* TCP

Health Check Configuration Options

| Setting | Description | Example |
| --- | --- | --- |
| Endpoint | DNS name or IP to check | app.example.com |
| Request Interval | How often Route 53 sends requests (10 or 30 seconds) | 30s |
| Failure Threshold | No. of consecutive failures before marking unhealthy | 3 |
| String Match | Expect a specific keyword in response | “Success” |
| Inverted Check | Invert logic (healthy if endpoint fails) | Rare cases |
| Alarm Integration | Use CloudWatch alarms | For metrics-based checks |

Where Health Checks Are Used

| Use Case | How It Works |
| --- | --- |
| Failover routing | If primary endpoint fails → traffic moves to secondary |
| Multi-value routing | Unhealthy IPs removed from DNS rotation |
| Latency-based routing | Can skip unhealthy region endpoints |
| Weighted routing | Health determines if weight applies or not |

Example: Failover Setup

Let’s say:

* Primary Region: us-east-1 → ALB endpoint
* Secondary Region: ap-south-1 → ALB endpoint

Steps:

1. Create 2 records in Route 53 for app.example.com
   * Primary → ALB in us-east-1
   * Secondary → ALB in ap-south-1
2. Create a health check for the primary endpoint
3. Mark routing type:
   * Primary = Failover (Primary)
   * Secondary = Failover (Secondary)

✅ If health check fails → traffic automatically shifts to secondary region.

🔐 Key Points to Remember

* Health checks are independent of AWS region (global service).
* Default check interval = 30 seconds (10s is more frequent, higher cost).
* 3 consecutive failures mark endpoint unhealthy.
* Supports public endpoints (private endpoints require CloudWatch integration).

Example Interview Answer

“In Route 53, health checks monitor the health of endpoints using HTTP, HTTPS, or TCP.  
If a health check fails, Route 53 can automatically remove that endpoint from DNS responses or trigger failover to a healthy one.  
I can also combine multiple checks or use CloudWatch alarms for application-level monitoring.”

⚡ Summary Table

| Feature | Description |
| --- | --- |
| Purpose | Monitor endpoint health for DNS-based routing |
| Check Types | Endpoint, CloudWatch, Calculated |
| Protocols | HTTP, HTTPS, TCP |
| Use Cases | Failover, Multi-value, Latency-based routing |
| Fail Threshold | Default 3 failures |
| Interval | 10 or 30 seconds |

1. **\* DualStack configuration in Route53**

Definition

DualStack in Route 53 means your DNS configuration supports both IPv4 and IPv6 addresses for the same domain name — allowing users to connect using either protocol depending on what their network supports.

In simple words:

DualStack = IPv4 (A record) + IPv6 (AAAA record)

Why It’s Needed

* The internet is gradually shifting to IPv6 (since IPv4 addresses are limited).
* Many modern devices and networks use IPv6 by default.
* AWS provides “DualStack” so your applications are reachable by both IPv4 and IPv6 clients.

🏗️ How It Works in Route 53

When you enable DualStack, Route 53 creates DNS records for both protocols:

| Record Type | Description | Example |
| --- | --- | --- |
| A Record | Maps hostname → IPv4 address | example.com → 192.0.2.10 |
| AAAA Record | Maps hostname → IPv6 address | example.com → 2001:db8::1234 |

When a client (browser, device, or API) makes a DNS request:

* If it supports IPv6, it uses the AAAA record.
* If not, it uses the A record (IPv4).

✅ Both records can point to the same resource (like an ALB or CloudFront).

🌐 Where “DualStack” Appears in AWS

1️⃣ – In Load Balancer DNS Names

When you create an Application Load Balancer (ALB) or Network Load Balancer (NLB), AWS gives it a DNS name like:

dualstack-myapp-alb-123456789.ap-south-1.elb.amazonaws.com

* The prefix dualstack means:
  + The ALB/NLB supports both IPv4 and IPv6.
  + You can create Route 53 records (A and AAAA) pointing to it.

So, if your hosted zone has:

* A record → Alias to that dualstack ALB
* AAAA record → Alias to that same ALB

→ You’ve created a DualStack setup.

2️⃣ – In CloudFormation or CLI Configuration

When defining alias targets to ELB in Route 53, you’ll often see:

dualstack.<elb-name>.<region>.elb.amazonaws.com

That ensures IPv6 compatibility.

3️⃣ – For CloudFront

CloudFront distributions natively support IPv6.  
You can just enable IPv6 in the distribution settings, and Route 53 automatically handles both A and AAAA records.

💻 Example Configuration in Route 53

| Record Name | Type | Alias Target / Value | Purpose |
| --- | --- | --- | --- |
| example.com | A | dualstack-myapp-alb-xyz.ap-south-1.elb.amazonaws.com | IPv4 |
| example.com | AAAA | dualstack-myapp-alb-xyz.ap-south-1.elb.amazonaws.com | IPv6 |

✅ Result:  
Users connecting via IPv4 or IPv6 can reach the same ALB endpoint.

🔐 Key Notes

| Feature | IPv4 | IPv6 |
| --- | --- | --- |
| Record Type | A | AAAA |
| Supported by | EC2, ALB, NLB, CloudFront, S3 (with static hosting) | ✅ |
| Internet Gateway Type | IGW | Egress-only IGW (for outbound IPv6) |

💬 Example Interview Answer

“In Route 53, DualStack means supporting both IPv4 and IPv6 access.  
For example, if I have an Application Load Balancer, AWS provides a DNS name starting with *dualstack*.  
In Route 53, I can create both A and AAAA alias records pointing to that ALB.  
This way, IPv4 clients use A record and IPv6 clients use AAAA — ensuring global compatibility.”

⚡ Summary Table

| Term | Meaning | Example |
| --- | --- | --- |
| DualStack | IPv4 + IPv6 support | dualstack-alb-name.region.elb.amazonaws.com |
| A Record | IPv4 address mapping | 192.0.2.10 |
| AAAA Record | IPv6 address mapping | 2001:db8::1234 |
| Used in | ALB, NLB, CloudFront, S3 | ✅ |
| Benefit | Both IPv4 & IPv6 users can connect | 🌍 |

1. **\* what is S3 and why customer choose S3 ?**

🧠 Definition:

Amazon S3 (Simple Storage Service) is an object storage service provided by AWS.  
It is used to store and retrieve any amount of data — at any time, from anywhere — via the web.

In simple words:  
S3 is like a scalable, highly durable online storage drive for all your data — backups, logs, images, videos, documents, etc.

📦 How S3 Stores Data

* Data is stored as objects inside buckets.
  + Bucket → like a folder (globally unique name)
  + Object → file + metadata
  + Each object has a key (name) and a version ID

Example:  
s3://my-company-bucket/finance/report2025.pdf

Why Customers Choose S3 (Key Benefits)

| Feature | Description |
| --- | --- |
| 1️⃣ Scalability | Automatically scales to store unlimited data — no need to provision storage. |
| 2️⃣ Durability | Designed for 99.999999999% (11 nines) durability — data replicated across multiple Availability Zones. |
| 3️⃣ Availability | Up to 99.99% availability (depending on storage class). |
| 4️⃣ Security | Supports IAM policies, Bucket policies, encryption (SSE-S3, SSE-KMS), access control lists, and Block Public Access. |
| 5️⃣ Cost-effective | Pay only for what you use — multiple storage classes (Standard, IA, Glacier, etc.) optimize cost. |
| 6️⃣ Easy Access | Access via console, CLI, SDK, APIs, and S3 URLs from anywhere. |
| 7️⃣ Data Management | Features like Lifecycle rules, Versioning, Replication, and Event notifications simplify management. |
| 8️⃣ Integration | Integrates with almost every AWS service — EC2, Lambda, CloudFront, Athena, Glue, etc. |

💡 Common Use Cases

| Use Case | Description |
| --- | --- |
| Backup & Restore | Store backups from on-prem or EC2 instances |
| Static Website Hosting | Host HTML/CSS/JS websites directly from S3 |
| Big Data & Analytics | Data lake for Athena, EMR, Redshift Spectrum |
| Disaster Recovery | Cross-region replication for high availability |
| Application Storage | Store media files, logs, or documents for apps |
| Archive | Long-term storage with Glacier / Deep Archive |

💬 Example Interview Answer

“Amazon S3 is an object storage service that lets you store and retrieve unlimited data securely and durably.  
Customers choose S3 because it’s scalable, cost-effective, highly durable, and integrates easily with other AWS services.  
It’s used for backups, static website hosting, big data storage, and disaster recovery.”

⚡ Summary

| Feature | Description |
| --- | --- |
| Type | Object Storage |
| Durability | 99.999999999% (11 nines) |
| Access | Console, CLI, SDK, API |
| Key Advantage | Scalable, Secure, Cost-effective |
| Example Use | Backups, Hosting, Data Lake |

1. **\* When the jar is created, how will you push it to S3?**

🧠 Scenario

You have a Java application that builds a .jar file (for example using Maven, Gradle, or Jenkins).  
Once the JAR is created, you need to upload (push) it to Amazon S3 — for storage, deployment, or sharing.

💡 There Are 3 Main Ways to Push the JAR to S3

Option 1️⃣ — Using AWS CLI (Most Common & Simple)

After your build process creates the .jar, you can use:

aws s3 cp myapp.jar s3://my-bucket-name/artifacts/

📌 Explanation:

* aws s3 cp → copy file to S3
* myapp.jar → local JAR file
* s3://my-bucket-name/artifacts/ → S3 bucket path (folder)

✅ Example with full command:

aws s3 cp target/myapp-1.0.jar s3://devops-artifacts/java-apps/myapp-1.0.jar

You can also upload entire folders:

aws s3 sync target/ s3://devops-artifacts/builds/

Option 2️⃣ — Automate via Jenkins Pipeline

If you’re using Jenkins for CI/CD:

1. Build your Java project using Maven/Gradle.
2. Add a post-build step or stage in Jenkinsfile to upload to S3.

📄 Example Jenkinsfile:

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'mvn clean package'

}

}

stage('Upload to S3') {

steps {

withAWS(region: 'ap-south-1', credentials: 'aws-creds') {

sh 'aws s3 cp target/myapp.jar s3://my-bucket/artifacts/myapp.jar'

}

}

}

}

}

✅ Result:  
Every time Jenkins builds a new JAR, it automatically uploads it to S3.

Option 3️⃣ — Using SDK / Programmatic Upload

If you want to upload from within your application or script:

Example: Python (Boto3)

import boto3

s3 = boto3.client('s3')

s3.upload\_file('target/myapp.jar', 'my-bucket', 'artifacts/myapp.jar')

Example: Java (AWS SDK for Java)

AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();

s3.putObject("my-bucket", "artifacts/myapp.jar", new File("target/myapp.jar"));

✅ Use this if your build or deployment process is part of an automated workflow or microservice.

🔐 Best Practices

| Practice | Description |
| --- | --- |
| Use IAM Roles | If running from EC2, Jenkins, or CodeBuild, attach a role with s3:PutObject permission |
| Use Versioning in S3 | To keep track of each build version |
| Use Environment Variables | For bucket name, region, and credentials |
| Use Lifecycle Policies | To archive or delete old artifacts automatically |

💬 Example Interview Answer

“Once the JAR is created during the build, I upload it to S3 using the AWS CLI command  
aws s3 cp target/myapp.jar s3://<bucket>/artifacts/.  
In a CI/CD pipeline like Jenkins, I automate this step by adding a post-build stage with AWS credentials.  
This ensures every build artifact is versioned and stored securely in S3 for deployment.”

⚡ Summary

| Method | Tool | Automation Level | Use Case |
| --- | --- | --- | --- |
| Manual | AWS CLI | Low | Simple upload |
| CI/CD | Jenkins / CodePipeline | High | Continuous integration |
| SDK | Python / Java code | Medium | Programmatic uploads |

1. **\* DynamoDB**

Amazon DynamoDB is a fully managed NoSQL database service provided by AWS.  
It offers high performance, automatic scaling, and serverless architecture for applications that need low-latency data access.

In simple words:  
DynamoDB is a fast and flexible key-value & document database that automatically scales as your application grows — no servers to manage.

⚙️ Key Features

| Feature | Description |
| --- | --- |
| 1️⃣ Fully Managed | AWS handles provisioning, scaling, patching, and replication — you only focus on your data. |
| 2️⃣ NoSQL Database | Stores data as key-value pairs or JSON documents (non-relational). |
| 3️⃣ High Performance | Single-digit millisecond response time, even at scale. |
| 4️⃣ Scalable | Automatically scales read and write capacity. Supports On-Demand or Provisioned modes. |
| 5️⃣ Highly Available & Durable | Data replicated across multiple Availability Zones in a region. |
| 6️⃣ Serverless | No infrastructure to manage. You pay only for what you use. |
| 7️⃣ Secure | Integrated with IAM, encryption (KMS), and VPC endpoints. |
| 8️⃣ Integrated with AWS Services | Works well with Lambda, API Gateway, CloudWatch, Data Pipeline, etc. |

🧩 Basic Structure

| Concept | Description |
| --- | --- |
| Table | Collection of data (like a table in RDBMS). |
| Item | A single record (like a row). |
| Attribute | A column or field in an item. |
| Primary Key | Unique identifier for each item. Two types: |
| → Partition Key | (e.g., UserID) — hashed for data distribution |
| → Composite Key | Partition Key + Sort Key (e.g., UserID + OrderDate) |

📄 Example

Table: Users

| UserID (PK) | Name | Email | City |
| --- | --- | --- | --- |
| 101 | Rekha | rekha@email.com | Chennai |
| 102 | Rohan | rohan@email.com | Bangalore |

⚡ Example Use Cases

| Use Case | Description |
| --- | --- |
| Web Apps / Mobile Apps | Store user profiles, session info, preferences |
| IoT Applications | Store sensor data with high velocity |
| Gaming | Real-time leaderboard data |
| E-Commerce | Product catalogs, orders, and cart data |
| Serverless Apps | Backend database for AWS Lambda + API Gateway |

🔄 Modes of Capacity

| Mode | Description | When to Use |
| --- | --- | --- |
| Provisioned Mode | You manually set Read/Write Capacity Units (RCUs/WCUs) | Predictable workloads |
| On-Demand Mode | DynamoDB automatically scales capacity | Unpredictable traffic patterns |

🔐 Security

* Integrated with IAM for access control
* Encryption at rest with AWS KMS
* VPC endpoints for private connectivity

🧠 Extra Features (for advanced interviews)

| Feature | Description |
| --- | --- |
| DynamoDB Streams | Captures real-time data changes for triggers (e.g., AWS Lambda). |
| Global Tables | Multi-region replication for disaster recovery or global apps. |
| TTL (Time to Live) | Automatically delete expired items. |
| DAX (DynamoDB Accelerator) | In-memory caching for microsecond latency. |

💬 Example Interview Answer

“Amazon DynamoDB is a fully managed, serverless NoSQL database service from AWS.  
It provides single-digit millisecond latency and scales automatically without managing servers.  
It’s ideal for applications like gaming, IoT, and mobile apps that need high throughput and low latency.  
Features like Streams, DAX, and Global Tables make it powerful for modern, distributed applications.”

⚡ Summary Table

| Property | DynamoDB |
| --- | --- |
| Type | NoSQL (Key-Value / Document) |
| Managed By | AWS (Fully Managed) |
| Scalability | Automatic |
| Speed | Millisecond latency |
| Best For | High-scale, low-latency apps |
| Example Services | Lambda, API Gateway, CloudWatch |

1. **\* vpc architecture**

🧠 Definition

VPC (Virtual Private Cloud) is a logically isolated section of AWS Cloud  
where you can launch AWS resources (like EC2, RDS, etc.) inside your own custom network.

It’s like your own private data center inside AWS — with full control over IP ranges, subnets, routing, and security.

🏗️ Typical VPC Architecture Diagram (Conceptually)

🌍 Internet

│

┌──────────────────────┐

│ Internet Gateway │

└──────────┬───────────┘

│

┌──────────┴──────────┐

│ VPC (10.0.0.0/16) │

└───────────────────────────────┘

├───────────────┬────────────────┤

│ Public Subnet │ Private Subnet │

│ (10.0.1.0/24) │ (10.0.2.0/24) │

│ │ │

EC2 (Web Server) │ EC2 (App DB)

↓ │ ↓

Route Table → IGW │ NAT Gateway → IGW

│ │

Security Groups + NACLs

🔹 Main Components of a VPC

| Component | Description |
| --- | --- |
| 1️⃣ VPC | The virtual network (you define CIDR block, e.g., 10.0.0.0/16). |
| 2️⃣ Subnets | Divide VPC into smaller networks:  • Public Subnet → Internet accessible  • Private Subnet → Internal-only |
| 3️⃣ Internet Gateway (IGW) | Allows resources in public subnet to access the internet. |
| 4️⃣ Route Tables | Define how network traffic is directed (e.g., public route → IGW, private route → NAT). |
| 5️⃣ NAT Gateway / NAT Instance | Lets private instances access the internet (for updates/downloads) without being exposed. |
| 6️⃣ Security Groups | Instance-level firewall — controls inbound/outbound traffic. |
| 7️⃣ Network ACLs (NACLs) | Subnet-level firewall — adds stateless filtering. |
| 8️⃣ Elastic IPs (EIP) | Static public IP addresses for EC2/NAT. |
| 9️⃣ VPC Peering / Transit Gateway | Connect multiple VPCs privately. |
| 🔟 Endpoints (S3 / DynamoDB) | Access AWS services privately without internet. |

🌍 Example VPC Setup

| Resource | Configuration |
| --- | --- |
| VPC CIDR | 10.0.0.0/16 |
| Public Subnet | 10.0.1.0/24 → Connected to IGW |
| Private Subnet | 10.0.2.0/24 → Connected to NAT |
| Route Tables | Public → 0.0.0.0/0 via IGW  Private → 0.0.0.0/0 via NAT |
| Security Group | Allow HTTP/HTTPS to web server, MySQL between app and DB |
| NACL | Allow specific port ranges only |
| EC2 Instances | Web Server (public) + App/DB Server (private) |

🔐 Security Layers

| Layer | Purpose |
| --- | --- |
| Security Groups | Instance-level protection (stateful). |
| NACLs | Subnet-level protection (stateless). |
| IAM Policies | Control access to resources. |

💬 Example Interview Answer

“A VPC is an isolated virtual network in AWS.  
In a typical VPC architecture, we create public subnets (for web servers) and private subnets (for databases or internal apps).  
The public subnet connects to the Internet Gateway, while private subnets use a NAT Gateway for outbound internet access.  
We manage routing via Route Tables, and control traffic using Security Groups and NACLs.  
This architecture provides both security and scalability for cloud deployments.”

🧠 Best Practices

✅ Use multiple Availability Zones for high availability  
✅ Keep databases in private subnets  
✅ Enable Flow Logs for monitoring  
✅ Use VPC Endpoints for private S3/DynamoDB access  
✅ Use least privilege in Security Groups

⚡ Summary

| Component | Description |
| --- | --- |
| VPC | Your private cloud network |
| Subnets | Logical divisions (public/private) |
| IGW | Enables internet connectivity |
| NAT Gateway | Private subnet internet access |
| Route Tables | Control traffic flow |
| Security Groups | Stateful firewalls |
| NACLs | Stateless firewalls |

1. **\* why we need ELB & how you select which type of loadbalancer suitable for my application.**

🧠 Part 1: Why We Need ELB (Elastic Load Balancer)

💡 Definition:

Elastic Load Balancer (ELB) automatically distributes incoming traffic across multiple EC2 instances, containers, or IP addresses in one or more Availability Zones.

In simple terms:  
ELB ensures your application is always available, scalable, and fault-tolerant by spreading traffic evenly among healthy targets.

🚀 Why We Need ELB

| Reason | Description |
| --- | --- |
| 1️⃣ High Availability | ELB routes traffic only to healthy instances, ensuring uptime. |
| 2️⃣ Fault Tolerance | If one instance fails, ELB automatically sends traffic to another healthy one. |
| 3️⃣ Scalability | Works seamlessly with Auto Scaling — as instances increase/decrease, ELB adjusts automatically. |
| 4️⃣ Security | Supports SSL/TLS termination, WAF (Web Application Firewall), and security group integration. |
| 5️⃣ Flexibility | Supports multiple protocols (HTTP, HTTPS, TCP, UDP, gRPC, WebSockets). |
| 6️⃣ Monitoring | Integrated with CloudWatch, providing latency and health check metrics. |
| 7️⃣ Simplifies DNS | Provides a single DNS endpoint (no need to manage IPs manually). |

🧩 Example:

Let’s say you have 3 EC2 instances running your web app.  
Without ELB → You must manually manage which instance gets requests.  
With ELB → It automatically balances requests and ensures continuous service even if one instance fails.

🧠 Part 2: How to Choose Which ELB Type for Your Application

AWS provides 3 main types of Load Balancers (plus a 4th newer one).  
Let’s see them one by one 👇

| Load Balancer | Layer | Best For | Protocols | Use Case |
| --- | --- | --- | --- | --- |
| 1️⃣ Application Load Balancer (ALB) | Layer 7 (HTTP/HTTPS) | Web applications, microservices | HTTP, HTTPS, WebSocket | Content-based routing (e.g., /api → one target, /app → another) |
| 2️⃣ Network Load Balancer (NLB) | Layer 4 (TCP/UDP) | High-performance, low-latency apps | TCP, UDP, TLS | Gaming, IoT, financial systems |
| 3️⃣ Gateway Load Balancer (GWLB) | Layer 3 (Network) | Security appliances | All traffic | Firewalls, IDS/IPS, inspection tools |
| 4️⃣ Classic Load Balancer (CLB) | Layer 4/7 | Legacy workloads | HTTP, HTTPS, TCP | Old applications (not recommended for new projects) |

⚙️ How to Choose the Right One

| Requirement | Choose |
| --- | --- |
| You need path-based or host-based routing | ✅ ALB |
| You need TLS termination and HTTP headers-based routing | ✅ ALB |
| You need to handle millions of connections per second | ✅ NLB |
| You need static IP support | ✅ NLB |
| You need network-level appliance integration (firewall, IPS) | ✅ GWLB |
| You have legacy apps using TCP/HTTP | ✅ CLB (only if required) |

📄 Example Scenarios

| Scenario | Load Balancer Type |
| --- | --- |
| Hosting a web app with frontend & APIs | ALB |
| Real-time chat or gaming server | NLB |
| Deploying a firewall or security service in AWS | GWLB |
| Old EC2-based app (pre-2016) | CLB |

💬 Example Interview Answer

“We use Elastic Load Balancers to distribute incoming traffic evenly across multiple instances for high availability and fault tolerance.  
It ensures only healthy instances receive requests and scales automatically with demand.

If my application is web-based and needs path-based routing (e.g., /api vs /app), I’d choose Application Load Balancer.  
If it’s a high-performance or TCP-based service like gaming or IoT, I’d go with Network Load Balancer.

For inspecting or filtering network traffic using firewalls, I’d use Gateway Load Balancer.”

⚡ Summary

| Type | Layer | Best For | Key Feature |
| --- | --- | --- | --- |
| ALB | 7 | HTTP/HTTPS | Smart routing, microservices |
| NLB | 4 | TCP/UDP | High performance, static IP |
| GWLB | 3 | Security appliances | Packet-level traffic inspection |
| CLB | 4/7 | Legacy apps | Basic load balancing |

1. **\* crosszone Loadbalancer**

Definition:

Cross-Zone Load Balancing is an Elastic Load Balancer (ELB) feature that allows the load balancer to distribute incoming traffic evenly across all registered targets in all Availability Zones (AZs).

In simple terms:  
Without it — traffic in one Availability Zone goes only to instances in that zone.  
With it — traffic from any AZ is distributed across all healthy instances in all AZs.

🌍 Why It’s Needed

When you deploy an ELB across multiple Availability Zones (for high availability), you might not have an equal number of EC2 instances in each zone.  
Without cross-zone balancing, one zone could be overloaded while another is underutilized.

⚙️ Example Scenario

| Availability Zone | Instances |
| --- | --- |
| us-east-1a | 2 EC2 instances |
| us-east-1b | 1 EC2 instance |

🔸 Without Cross-Zone Load Balancing

Each load balancer node in an AZ only sends traffic to instances in its own AZ.  
So:

* 50% of requests → sent to us-east-1a
* 50% of requests → sent to us-east-1b

Since us-east-1b has only 1 instance, that instance gets 50% of total traffic, while us-east-1a’s 2 instances share the other 50%.  
👉 Unbalanced load.

🔹 With Cross-Zone Load Balancing

Now, each ELB node can distribute traffic to instances in all AZs.  
So:

* All 3 instances (in both AZs) share the total load equally (≈33% each).  
  👉 Balanced load, better utilization, improved performance.

🧩 Support by Load Balancer Type

| Load Balancer Type | Cross-Zone Balancing Support | Default |
| --- | --- | --- |
| Application Load Balancer (ALB) | ✅ Always enabled, no extra cost | Enabled |
| Network Load Balancer (NLB) | ✅ Optional (can be turned ON/OFF) | Disabled by default, may incur cross-AZ data charges |
| Gateway Load Balancer (GWLB) | ❌ Not supported | – |
| Classic Load Balancer (CLB) | ✅ Optional (can be turned ON/OFF) | Disabled by default |

💰 Cost Consideration

* For ALB → No extra cost.
* For NLB or CLB → Enabling cross-zone load balancing may incur inter-AZ data transfer charges, since traffic may cross Availability Zones.

💬 Example Interview Answer

“Cross-zone load balancing allows the load balancer to distribute incoming traffic evenly across all instances in all Availability Zones.  
Without it, each load balancer node can only route traffic to instances in its own zone, which can cause uneven load if zones have different numbers of instances.

For example, if Zone A has 2 instances and Zone B has 1, enabling cross-zone balancing ensures all 3 share traffic equally.  
It’s always enabled for ALB, and optional for NLB and CLB.”

⚡ Summary

| Feature | Description |
| --- | --- |
| Purpose | Distribute traffic evenly across all AZs |
| ALB | Always ON |
| NLB/CLB | Optional |
| Benefit | Better resource utilization & performance |
| Drawback | May incur cross-AZ data charges (NLB/CLB) |

1. **\* cloudwatch & cloudtrail**

Definition

| Service | Purpose | Type |
| --- | --- | --- |
| Amazon CloudWatch | Monitors AWS resources and applications in real time. | Performance & operational monitoring |
| AWS CloudTrail | Records API calls and actions made in your AWS account. | Security & auditing |

☁️ 1️⃣ Amazon CloudWatch

💡 What It Does:

* Collects and tracks metrics, logs, and events from AWS services and custom apps.
* Helps monitor performance, utilization, and health.

📊 Common Metrics:

| Resource | Example Metrics |
| --- | --- |
| EC2 | CPU utilization, disk I/O, network traffic |
| RDS | Free storage, connections, read/write latency |
| Lambda | Invocation count, duration, errors |
| ELB | Request count, latency, healthy hosts |

⚙️ Key Features

* Alarms → Trigger actions when thresholds are crossed  
  (e.g., Auto Scaling, SNS notifications)
* Dashboards → Visualize metrics
* Logs → Store, search, and analyze logs
* Events / Rules → Trigger actions automatically (like restarting an instance)

🔔 Example Use Case:

Auto Scale EC2 instances when CPU utilization exceeds 80%.

🔍 2️⃣ AWS CloudTrail

💡 What It Does:

* Tracks “who did what and when” in your AWS account.
* Records API calls (via console, CLI, SDK, or other AWS services).

🧾 What It Records:

| Data Type | Example |
| --- | --- |
| User Identity | Who made the call (IAM user, role, root) |
| Event Time | When it happened |
| API Action | What action was taken (e.g., RunInstances, DeleteBucket) |
| Service | Which AWS service was used |
| Source IP | Where the request came from |

⚙️ Key Features

* Logs API activity across all AWS services.
* Stores logs in S3, and can also send to CloudWatch Logs or EventBridge.
* Supports multi-region and organization-level trails.
* Useful for security auditing, compliance, and incident investigation.

🔔 Example Use Case:

Detect who deleted an S3 bucket or stopped a production EC2 instance.

⚖️ CloudWatch vs CloudTrail — Comparison Table

| Feature | CloudWatch | CloudTrail |
| --- | --- | --- |
| Purpose | Monitor performance and operations | Audit API activity and governance |
| Type | Monitoring service | Logging & auditing service |
| Tracks | Metrics, logs, events | API calls and user activity |
| Data Source | Performance data from AWS services | Management & data plane API calls |
| Storage | CloudWatch Logs | S3 bucket |
| Real-Time Alerts | Yes (Alarms, dashboards) | Yes (via EventBridge) |
| Used For | Monitoring, scaling, troubleshooting | Security auditing, compliance, forensics |
| Example Question Answered | “Is my EC2 CPU high?” | “Who stopped my EC2 instance?” |

💬 Example Interview Answer

“CloudWatch is used for monitoring AWS resources and applications — it collects metrics, logs, and events, and can trigger alarms or Auto Scaling actions.

CloudTrail, on the other hand, is used for auditing and governance — it records who performed what actions on AWS services via console, CLI, or SDK.

In short, CloudWatch answers 'what is happening', while CloudTrail answers 'who did what'.”

⚡ Summary

| Service | Key Role |
| --- | --- |
| CloudWatch | Monitors performance & health |
| CloudTrail | Tracks API activity & security events |
| Both Together | Give full visibility + accountability of your AWS environment |

1. **\* How many aws account you have managed?**

🟢 Beginner / Small Environment:

“I’ve managed 2 to 3 AWS accounts — typically one for development, one for testing, and one for production.  
Each account had separate IAM roles, billing, and networking setups for isolation and security.”

🟡 Intermediate (Team-Level Projects):

“I’ve managed around 5–8 AWS accounts across different projects.  
We followed a multi-account strategy using AWS Organizations, where each account represented a different environment or business unit.  
I used AWS IAM roles, Organizations SCPs (Service Control Policies), and CloudWatch + CloudTrail aggregation for centralized management and monitoring.”

🔵 Advanced (Enterprise or Multi-Project):

“I’ve managed 10+ AWS accounts under a single AWS Organization.  
Each account was designated for specific functions — for example,

* Security account for centralized logging & GuardDuty
* Shared services account for networking (VPC, Transit Gateway)
* Application accounts for Dev, QA, and Prod.

I implemented cross-account access using IAM roles, consolidated billing, and AWS Control Tower for governance and compliance.”

**30 \* client has different data centers on different locations. if your client has 50 aws accounts now there is a requirement to establish a connectivity so that the application which is hosted in data centers servers can easily communicate with applications (or) resources which are hosted on different vpc and diff aws accounts so how you are going to establish the connection**

* Client has multiple data centers (on-prem) in different locations.
* Client also has ~50 AWS accounts, each with their own VPCs.
* Need to establish connectivity between:
  + On-premise data centers ⇄ AWS VPCs
  + VPCs across different AWS accounts and regions
* Goal: All applications and resources should communicate securely and efficiently.

🧠 Step-by-Step Answer

Step 1️⃣ – Connect On-Premise Data Centers to AWS

You have two main options for hybrid connectivity:

🟢 Option 1: AWS Direct Connect

* Provides a dedicated private network connection between on-prem data centers and AWS.
* Offers high bandwidth, low latency, and consistent performance.
* You can connect Direct Connect to:
  + A Direct Connect Gateway → which can attach to multiple VPCs across multiple AWS accounts/regions.

🧩 Example:

Each data center connects to AWS using a Direct Connect line, then all VPCs from 50 accounts attach to a Direct Connect Gateway.

🟣 Option 2: Site-to-Site VPN

* Encrypted connection over the internet.
* Used as:
  + Backup connection for Direct Connect
  + Or for smaller environments / testing
* VPN terminates on Virtual Private Gateway (VGW) or Transit Gateway

Step 2️⃣ – Interconnect Multiple AWS Accounts / VPCs

With 50 AWS accounts, you can’t efficiently do 1:1 VPC peering — it becomes too complex.

So use:

🟢 Option: AWS Transit Gateway (TGW)

* Acts as a central hub to connect multiple VPCs and on-prem networks.
* Simplifies networking — hub-and-spoke architecture.
* Supports cross-account attachments, cross-region peering, and Direct Connect integration.
* You can share it across accounts using AWS Resource Access Manager (RAM).

🧩 Example:

Each AWS account’s VPC is attached to a central Transit Gateway in a shared networking account.  
The Transit Gateway is also connected to a Direct Connect Gateway → which connects to the on-prem data centers.

Step 3️⃣ – Manage Routing and Security

* Use Transit Gateway route tables to control how traffic flows between VPCs and on-prem networks.
* Use Security Groups and NACLs to allow only necessary communication.
* Optionally use AWS Network Firewall or 3rd-party appliances for inspection and filtering.

Step 4️⃣ – Centralized Management

* Create a Shared Services / Networking Account:
  + Hosts the Transit Gateway and Direct Connect Gateway
  + Shared to other accounts using AWS RAM
* Manage DNS centrally using Route 53 Private Hosted Zones and Route 53 Resolver rules
* Monitor all network traffic using VPC Flow Logs and CloudWatch

🧩 Architecture Summary

[ On-Prem Data Centers ]

│

(Direct Connect / VPN)

│

[ Direct Connect Gateway ]

│

[ Transit Gateway (Hub) ]

├── VPC (Account 1)

├── VPC (Account 2)

├── ...

└── VPC (Account 50)

All communication between on-prem and AWS accounts flows through Transit Gateway, ensuring centralized control and scalable connectivity.

💬 Example Interview Answer (2-minutes concise)

“Since the client has multiple data centers and around 50 AWS accounts, I would set up a hub-and-spoke network using AWS Transit Gateway.  
Each VPC in those 50 accounts would be attached to the Transit Gateway, which acts as the central communication hub.

For on-prem connectivity, I would use AWS Direct Connect with a Direct Connect Gateway that links to the Transit Gateway.  
This setup provides secure, scalable, and low-latency connectivity between on-prem data centers and all AWS VPCs.

Using AWS Resource Access Manager (RAM), I’d share the Transit Gateway with all accounts.  
Routing is controlled through TGW route tables, and we can enforce security using security groups, NACLs, and optionally AWS Network Firewall.

This architecture minimizes complexity and provides a centralized, highly available hybrid network.”

⚡ Bonus (to impress interviewer):

You can add:

“For resilience, I’ll add a VPN as a backup to Direct Connect, and enable Transit Gateway route propagation for automatic updates.  
DNS resolution between on-prem and AWS is managed through Route 53 Resolver endpoints.”

**31 \* customer has 50 (or) 100 odd aws accounts they are looking for a solution so that each and every single vpc can communicate with each other how you are going to establish that connectivity?**

Customer situation:

* Has 50–100 AWS accounts, each with one or more VPCs.
* Each VPC needs to communicate with every other VPC (full-mesh communication).
* Needs a scalable, secure, and centralized solution.

🧠 Possible Solutions

Let’s evaluate the main options AWS provides 👇

### 🅰️ Option 1: VPC Peering (Not Recommended for Large Scale)

* You can connect two VPCs directly using VPC Peering.
* Works well for few VPCs (2–10).
* But not scalable for 50 or 100 VPCs.

🧮 Why not scalable:

* Peering is one-to-one.
* Each VPC would need peering with 99 others → ~4,950 connections (n\*(n−1)/2).
* Huge management and routing complexity.

✅ Good for: Small setups  
❌ Not suitable for: 50–100 accounts

### 🅱️ Option 2: Transit VPC (Legacy Design)

* Earlier solution before AWS Transit Gateway existed.
* Used a central VPC with VPN connections (IPSec tunnels) to all other VPCs.
* Required third-party appliances (like Cisco CSR, Palo Alto).

✅ Centralized  
❌ Expensive, complex, hard to manage  
⚠️ Mostly replaced by Transit Gateway

### 🅲️ Option 3: AWS Transit Gateway (Recommended & Modern Solution ✅)

Best, scalable, and most commonly used approach today.

🚀 AWS Transit Gateway (TGW) Solution

💡 How it works

* Acts as a central hub that connects multiple VPCs and on-premises networks.
* Each VPC attaches to the TGW (hub-and-spoke model).
* TGW handles routing between all connected networks.

🧩 Key Features

| Feature | Description |
| --- | --- |
| Hub-and-Spoke Model | All VPCs connect to one TGW instead of full mesh |
| Cross-Account Sharing | Share TGW using AWS Resource Access Manager (RAM) |
| Cross-Region Peering | Connect TGWs across regions |
| Centralized Control | Routing and security managed centrally |
| Scalable | Thousands of attachments supported |
| Integrates with Direct Connect / VPN | For hybrid connectivity with on-prem data centers |

🗺️ Architecture Design

+----------------------+

| AWS Transit Gateway |

+----------+-----------+

|

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

| | | | |

[VPC-A] [VPC-B] [VPC-C] [VPC-D] [VPC-E]

(Account1) (Account2) (Account3) (Account4) (Account5)

| | | | |

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

Central Routing via TGW

🧰 Implementation Steps

1. Create a Central “Network” Account
   * Host the AWS Transit Gateway (TGW) here.
   * Enable AWS Organizations for account management.
2. Share TGW using AWS Resource Access Manager (RAM)
   * Share it with all other AWS accounts (50–100).
3. Attach each VPC to the TGW
   * From each account, attach the VPC to the shared TGW.
   * Update route tables to send inter-VPC traffic via TGW.
4. Configure TGW Route Tables
   * Define which VPCs can communicate with which others.
   * Can create separate route tables for isolation (Prod vs Dev).
5. (Optional) Peering across regions
   * If VPCs exist in multiple regions, peer TGWs across regions.
6. (Optional) Connect to On-Prem
   * Attach Direct Connect Gateway or VPN to the TGW for hybrid setups.

🔐 Security and Governance

* Use Security Groups, NACLs, and Network Firewall for traffic control.
* Use SCPs (Service Control Policies) under AWS Organizations for governance.
* Enable VPC Flow Logs and CloudWatch for monitoring.

💬 Example Interview Answer

“If the customer has around 50 or 100 AWS accounts and wants all VPCs to communicate, I would use an AWS Transit Gateway (TGW).

The TGW acts as a central hub, and each VPC from all accounts connects to it as a spoke.  
Using AWS Resource Access Manager (RAM), I can share the TGW with multiple accounts.

This design eliminates the need for hundreds of peering connections and provides a scalable, secure, and centrally managed network.

We can control communication using Transit Gateway route tables, and optionally connect on-prem data centers through Direct Connect or VPN.

This is the recommended AWS architecture for multi-account, multi-VPC connectivity.”

⚡ Bonus Points to Impress

You can add:

* For multi-region setup, use TGW peering.
* For monitoring, enable Transit Gateway Flow Logs.
* For cost control, segment VPCs using multiple route tables.

✅ In short:

| Option | Suitable For | Scalability |
| --- | --- | --- |
| VPC Peering | Small setup (2–10 VPCs) | ❌ |
| Transit VPC | Legacy setups | ⚠️ |
| Transit Gateway | Large multi-account architecture | ✅✅✅ |

**32 \* customer has a physical side production environment (or) workloads are running on aws there is 1 particular Ip/CIDR which is continously hitting their environment multiple times in a day they are suspecting that a malicious activity they are asking you to block any kind of a traffic from a Ip/CIDR how you will going to block them?**

* Customer has workloads running in AWS (production).
* One IP/CIDR is continuously sending traffic (potential malicious activity).
* Need to block that IP/CIDR from accessing any resource in the AWS environment.

🧠 Possible AWS-Level Solutions

There are multiple layers where you can block an IP or CIDR.  
You choose based on where the threat is coming and which service is exposed.

Let’s go layer by layer 👇

🧩 Option 1️⃣: Network Firewall / WAF (Best Practice for Application Layer)

1. AWS WAF (Web Application Firewall)

* Works at Layer 7 (HTTP/S).
* Protects Application Load Balancers (ALB), API Gateway, or CloudFront.
* You can block requests from specific IPs or CIDR ranges.

✅ Steps:

1. Go to AWS WAF → Web ACLs.
2. Create a Rule → IP match condition.
3. Specify the malicious IP/CIDR (e.g., 203.0.113.0/24).
4. Set Action = Block.
5. Associate this Web ACL with the ALB / API Gateway / CloudFront.

✅ Use Case:  
If the suspicious traffic is hitting your web application endpoints.

🧩 Option 2️⃣: AWS Network Firewall (Best for VPC-level blocking)

* Works at Layer 3 & 4 (Network/Transport).
* Can block IPs, ports, or protocols inside your VPC.
* Integrated with Transit Gateway or VPC subnets.

✅ Steps:

1. Deploy AWS Network Firewall in the VPC or central Transit Gateway.
2. Create stateless or stateful rule groups.
3. Add a rule to drop traffic from the malicious IP/CIDR.
4. Associate firewall policy with the VPC subnet(s) or TGW attachment.

✅ Use Case:  
If the malicious IP is sending non-HTTP traffic (like SSH, RDP, database connections).

🧩 Option 3️⃣: Network ACL (NACL) — Quick, Simple Method

* Works at subnet level (Layer 3/4).
* Can deny specific IP or CIDR before reaching EC2.

✅ Steps:

1. Go to VPC → Network ACLs.
2. Identify the NACL associated with the target subnet.
3. Add an Inbound rule to deny that specific IP/CIDR.  
   Example:
4. Rule #: 100
5. Type: ALL Traffic
6. Source: 203.0.113.0/24
7. Action: DENY
8. Ensure you have a “Allow” rule for legitimate traffic.

✅ Use Case:  
When the malicious IP hits a specific subnet or EC2 instance directly.

🧩 Option 4️⃣: Security Groups (Limited Option)

* Security Groups are stateful and only allow traffic; they can’t explicitly deny.  
  ❌ So, you cannot block an IP using a Security Group directly.  
  ✅ But you can remove rules that allow public access (e.g., remove 0.0.0.0/0 from inbound SSH).

🧩 Option 5️⃣: Route 53 Firewall (For DNS blocking)

* If the malicious IP is DNS-based or domain-based traffic,  
  you can use Route 53 Resolver DNS Firewall to block domain names before resolution.

✅ Use Case:  
If you suspect your environment is reaching out to malicious domains.

🛡️ Best-Practice Answer (Short, Interview Version)

“Since the customer is seeing repeated hits from a suspicious IP/CIDR, I would first analyze the traffic using VPC Flow Logs or CloudWatch metrics to confirm the pattern.

Then, depending on the exposure type:

* If it’s a web application, I’ll use AWS WAF with an IP match rule to block that IP/CIDR.
* If it’s network-level traffic, I’ll apply a Network ACL deny rule or use AWS Network Firewall for centralized control.

For large-scale or multi-account environments, I’ll integrate AWS Firewall Manager to push the blocking rule across all accounts.

This ensures the malicious IP is blocked immediately and consistently across the entire AWS infrastructure.”

⚙️ Bonus: Multi-Account / Organization-Level Protection

If you have many AWS accounts (e.g., 50–100):

* Use AWS Firewall Manager.
* It allows you to centrally manage and apply WAF / Network Firewall policies across all accounts in the AWS Organization.

✅ Example:

One rule in Firewall Manager → blocks malicious IP across all accounts’ ALBs / CloudFront / TGWs automatically.

⚡ Summary Table

| Layer | AWS Service | Use Case | Can Block IP/CIDR? |
| --- | --- | --- | --- |
| Application (L7) | AWS WAF | Web apps (HTTP/S) | ✅ |
| Network (L3/L4) | AWS Network Firewall | VPC-level filtering | ✅ |
| Subnet (L3/L4) | Network ACL (NACL) | Simple IP block | ✅ |
| Instance | Security Group | Allow only trusted IPs | ⚠️ (no explicit deny) |
| DNS | Route 53 DNS Firewall | Block domain-based threats | ✅ |
| Org-wide | Firewall Manager | Apply across accounts | ✅✅✅ |

🧩 Example Real-World Implementation

[Malicious IP]

↓

[Internet Gateway]

↓

[AWS WAF / Network Firewall]

↓

[VPC Subnets + NACL Rules]

↓

[EC2 / ALB / Application]

Traffic from the malicious IP is dropped at the earliest layer (edge).

**33 \* CIDR for 15 ec2 instances , 2 subnets 1 private and 1 public , write CIDR ranges**

Nice — here’s a simple, interview-friendly CIDR plan you can use immediately.

Requirements recap

* Total EC2s: 15
* Subnets: 1 public, 1 private
* Keep room for growth and AWS reservation rules

Recommended VPC & subnet CIDRs

* VPC: 10.0.0.0/24 (256 addresses)
* Public subnet: 10.0.0.0/26 → covers 10.0.0.0 - 10.0.0.63 (64 addresses)
* Private subnet: 10.0.0.64/26 → covers 10.0.0.64 - 10.0.0.127 (64 addresses)

Why: each /26 gives 64 addresses (AWS reserves 5 per subnet), so you get 59 usable addresses per subnet — plenty for 15 instances and future growth.

Usable addresses (after AWS reservation)

* Per subnet usable ≈ 59 (64 − 5 reserved by AWS)
* Total usable in both subnets ≈ 118

Example placement (suggestion)

* Public subnet: Load balancers, bastion/NAT, public-facing services — e.g., 4 EC2s
* Private subnet: App/Web/DB servers — e.g., 11 EC2s

You can adjust the split (e.g., /25 for larger subnets or /28 for very small) — but /26 is a practical, interview-safe default for 15 instances.

**34 \* you have created one server in a private subnet now you have given the Ip address to me. I am the person who will install some softwares (or) configurations on that server. I am siting infront of my laptop i am complaining you that i am not able to access the machinefrom my laptop how you can resolve it?**

Excellent scenario 👏 — this is a classic AWS networking interview question about connectivity to private subnets.

Let’s break it down clearly 👇

🧩 Scenario Recap

* A server (EC2 instance) is created in a private subnet.
* You shared its private IP with someone (your teammate) sitting outside AWS (on their laptop).
* They cannot access the server.

🔍 Root Cause

A private subnet in AWS does not have direct internet access —  
it’s isolated for security reasons.  
Instances inside a private subnet:

* Cannot be accessed directly from the internet.
* Cannot receive inbound SSH / RDP traffic directly.

🛠️ How to Resolve It

✅ Option 1: Use a Bastion Host (Jump Server)

Best practice for secure access.

1. Launch a bastion host (jump server) in a public subnet (same VPC).
2. Assign it a public IP / Elastic IP.
3. Allow SSH from your laptop’s IP to the bastion host:
4. Inbound rule: SSH (22) → Source: <your IP>
5. From your laptop, SSH into the bastion:
6. ssh -i mykey.pem ec2-user@<bastion-public-ip>
7. Then SSH into the private instance:
8. ssh ec2-user@<private-ip>

➡️ You are now inside the private subnet via the bastion host.

✅ Option 2: Use AWS Systems Manager (SSM) Session Manager

More secure — no SSH keys, no open ports.

1. Attach the AmazonSSMManagedInstanceCore IAM role to the private instance.
2. Ensure it has internet access (via NAT Gateway) or VPC endpoint for SSM.
3. From AWS Console → Systems Manager → Session Manager → Start session.
4. You can now directly connect to the private EC2 from the console.

➡️ No need for bastion or SSH access at all.

✅ Option 3: Temporary Access via VPN or Direct Connect

If your organization has a VPN connection or Direct Connect setup to the VPC:

* You can connect your laptop to the VPN.
* Once connected, your laptop is part of the same private network → can access the private IP.

✅ Option 4: Move Instance to Public Subnet (Not Recommended)

If it’s only for quick testing (and not production):

* Move EC2 to a public subnet.
* Attach a public IP or Elastic IP.
* Open port 22 (SSH) to your IP.
* Then you can access it directly.

⚠️ Not secure — use only in dev/test environments.

🔒 Summary Table

| Method | Description | Security Level |
| --- | --- | --- |
| Bastion Host | Jump server in public subnet for SSH | 🔒 Secure |
| SSM Session Manager | Browser-based access (no public IP) | 🟢 Most Secure |
| VPN / Direct Connect | Corporate network access | 🔒 Secure |
| Public Subnet Access | Direct internet access | ⚠️ Least secure |

✅ Example Real-World Setup

* Public subnet → Bastion host (with Elastic IP)
* Private subnet → Application / Database servers
* Security Groups:
  + Bastion → SSH allowed from admin IPs
  + Private EC2 → SSH allowed only from bastion’s security group

🧠 In short:

“Since the EC2 is in a private subnet, it has no public IP — so I’ll create a Bastion Host or use AWS Systems Manager to connect securely.”

**35 \* how you can access a machine without a .pem key?**

Short answer: A few safe ways — depending on your environment and permissions — let you access an instance even if you lost the .pem file. Below I list the most common methods, ordered from safest/easiest to more invasive, with concrete steps and commands you can run right away.

⚠️ Warning: many of these require AWS console/CLI permissions (EC2 modify, IAM role attach, SSM, or ability to stop/modify instances). Don’t use them on systems you don’t own or have explicit permission to manage.

1) Recommended — AWS Systems Manager (Session Manager) (no key required)

Requirements: SSM Agent installed on the instance (most Amazon Linux/Windows AMIs have it), and the instance has an IAM role with AmazonSSMManagedInstanceCore.

Steps:

1. Attach or confirm an instance role with AmazonSSMManagedInstanceCore.
2. In AWS Console: *Systems Manager → Session Manager → Start session* and pick the instance.  
   OR using AWS CLI:

aws ssm start-session --target i-0123456789abcdef0

This provides an interactive shell — no SSH key needed.

2) EC2 Instance Connect (temporary SSH key / browser connect)

Works for supported AMIs (Amazon Linux 2, certain Ubuntu). Lets you push a one-time public key or use the browser-based SSH:

* Console: EC2 → Instances → Connect → EC2 Instance Connect (browser).
* Or push key via CLI:

aws ec2-instance-connect send-ssh-public-key \

--instance-id i-0123456789abcdef0 \

--instance-os-user ec2-user \

--availability-zone us-east-1a \

--ssh-public-key file://mykey.pub

# then ssh ec2-user@PUBLIC\_IP -i path/to/private-key-corresponding-to-mykey.pub

You do need the private key for the pair you just generated locally, but you do not need the original .pem for the instance.

3) Add a new SSH key by modifying the root volume (most universally works)

Use if SSM / Instance Connect aren't available.

High-level steps:

1. Stop the instance.
2. Detach its root EBS volume (e.g. /dev/xvda).
3. Attach the volume to a helper instance as /dev/sdf.
4. On helper instance:

sudo mkdir /mnt/recover

sudo mount /dev/xvdf1 /mnt/recover # adjust device name/partition as needed

# add your public key to the user's authorized\_keys

sudo mkdir -p /mnt/recover/home/ec2-user/.ssh

sudo sh -c 'echo "ssh-rsa AAAA... your-key-comment" >> /mnt/recover/home/ec2-user/.ssh/authorized\_keys'

sudo chown -R 1000:1000 /mnt/recover/home/ec2-user/.ssh # adjust UID/GID as needed

sudo chmod 700 /mnt/recover/home/ec2-user/.ssh

sudo chmod 600 /mnt/recover/home/ec2-user/.ssh/authorized\_keys

sudo umount /mnt/recover

1. Detach and reattach the volume back to the original instance as root, start instance, SSH in with the corresponding private key.

4) Create an AMI and launch a replacement instance with a new key

1. Create an AMI (Image) of the existing instance or snapshot the root volume.
2. Launch a new instance from that AMI and choose a new key pair during launch.  
   This preserves the instance image and lets you pick a new .pem.

5) Use EC2 Serial Console (for Nitro-based instances) — useful if SSH broken

Requirements: enable Serial Console access in account and instance OS configured to allow console login.

* Use the EC2 Serial Console from the AWS Console to log in, fix /home/\*/.ssh/authorized\_keys, or reset password.

6) For Windows instances — use EC2Rescue / GetPasswordData alternatives

* If you lost key for a Windows EC2 (where password is encrypted with .pem), you can use Systems Manager to run Reset-Password tools or detach the volume and edit the SAM database via a helper instance to reset Administrator password.

7) If it’s not cloud/AWS (generic Linux server)

* If you have another user account with sudo, ssh with that user.
* If you have physical or KVM access, boot to rescue mode/single-user, add a public key to ~/.ssh/authorized\_keys or enable password auth temporarily.
* If you have console access from the provider, use it to add keys.

Security best-practices & follow-ups

* After regaining access: rotate keys, remove any temporary keys added, enable/confirm SSM for future access, disable password auth and root SSH login, and store your private key safely (or use an SSH agent).
* Prefer SSM and Instance Connect for keyless, auditable access.

**36 \* if you create a transit gateway what exactly you will define in attachments?**

🔹 What is AWS Transit Gateway (TGW)?

AWS Transit Gateway (TGW) acts like a central hub that connects multiple VPCs, on-premises networks, and even VPN connections — all through a single gateway.

It simplifies complex VPC peering setups and provides centralized routing and connectivity.

🔹 Transit Gateway Attachments — Core Concept

When you create a Transit Gateway, it’s just the hub.  
To actually connect resources, you need to create attachments.

Each attachment defines what is being connected to the TGW.

🔹 Types of Attachments

| Attachment Type | Description | Use Case |
| --- | --- | --- |
| VPC Attachment | Connects a VPC (via subnets) to the TGW | To allow communication between VPCs |
| VPN Attachment | Connects an on-premises network via a VPN connection | For hybrid cloud setups |
| Direct Connect (DX) Attachment | Connects an on-premises data center via Direct Connect | For high-speed private link |
| Peering Attachment | Connects Transit Gateways across regions or accounts | For inter-region connectivity |
| Connect Attachment | Used for third-party SD-WAN or appliances | To integrate with SD-WAN routers |

🔹 When You Create an Attachment, You Define:

1. Attachment Type

* e.g. VPC, VPN, DX, or Peering

2. Resource Details

* For VPC: select VPC ID
* For VPN: select Customer Gateway
* For DX: select DX Gateway

3. Subnets (for VPC attachments)

* You must select one subnet per Availability Zone.
* TGW uses these subnets to establish ENIs (Elastic Network Interfaces) inside your VPC.

4. Routing Configuration

* Define whether propagation or association should be enabled for that attachment in the Transit Gateway Route Table.
  + Association → Determines which route table this attachment uses.
  + Propagation → Determines if routes from this attachment are automatically shared with others.

🔹 Example: VPC Attachment

Let’s say you have:

* VPC-A (10.0.0.0/16)
* VPC-B (172.16.0.0/16)
* Both need to communicate via TGW.

You will:

1. Create a Transit Gateway.
2. Create VPC Attachments:
   * Attachment 1 → VPC-A + choose subnets from us-east-1a, us-east-1b
   * Attachment 2 → VPC-B + choose subnets from us-east-1a, us-east-1b
3. Update the Transit Gateway Route Table:
   * Associate and propagate routes between both attachments.

Result → Both VPCs can talk via TGW 🚀

🔹 Summary

| Setting | Defined in Attachment | Example |
| --- | --- | --- |
| Attachment Type | What is connected | VPC / VPN / DX / Peering |
| Resource ID | Which resource | vpc-12345 |
| Subnets | Where TGW ENIs will be created | 1 subnet per AZ |
| Route Table Association | Which TGW route table to use | tgw-rtb-xyz |
| Route Propagation | Whether to auto-share routes | Enabled for dynamic routing |

🧠 In short:

“When you create a Transit Gateway attachment, you define *what* you’re connecting (VPC, VPN, etc.), *where* in the VPC it connects (subnets), and *how* routing is handled (association & propagation).”

**37 \* how your sharing the key for a group of 4 members & these 4 members will be logged in from same username after 4/5 days customer complaining that a file deleted how can you track back who did what?**

Good one — two problems here: (A) how the key was shared, and (B) how to *forensically* find who deleted the file when everyone used the same username/key. Short answer: if people shared a single private key or a single OS account, attribution is hard or impossible unless you already had per-user auditing enabled. Below I’ll give (1) how to investigate now (fast actions and CLI commands), (2) why shared keys/users are bad, and (3) what to put in place so this never happens again.

1) Immediate investigation (find who / what deleted the file)

Do these in order — they are the fastest ways to gather evidence.

A. If the file was in S3

* Check CloudTrail for DeleteObject events (has the principal, source IP, userAgent, time).

# search CloudTrail for DeleteObject events (adjust --start-time/--end-time)

aws cloudtrail lookup-events \

--lookup-attributes AttributeKey=EventName,AttributeValue=DeleteObject \

--start-time "2025-10-25T00:00:00Z" --end-time "2025-10-30T23:59:59Z"

* Check S3 server access logs (if enabled) or S3 object-level logging (CloudTrail data events) — these contain requester IP and requester ID.
* If object versioning was enabled you can restore the deleted version.

B. If the file was on an EC2 instance (Linux/Windows)

1. If Systems Manager (SSM) Session Manager was used
   * Check CloudTrail for StartSession events and Session Manager logs in CloudWatch / S3 (if you enabled session logging). CloudTrail StartSession includes the IAM principal who started the session.

aws cloudtrail lookup-events --lookup-attributes AttributeKey=EventName,AttributeValue=StartSession \

--start-time "..." --end-time "..."

* Check CloudWatch Logs / S3 bucket where session logs are stored.

1. If users SSH’d in with the same key and same user account
   * Check OS auth logs for connection timestamps and source IPs:
     + Debian/Ubuntu:
     + sudo grep 'sshd' /var/log/auth.log | grep 'Accepted'
     + sudo grep 'username' /var/log/auth.log | grep -i 'session'
     + RHEL/CentOS:
     + sudo grep 'sshd' /var/log/secure | grep 'Accepted'
   * Look at ~/.bash\_history but: if the same username is used, bash\_history will not distinguish people and can be edited/deleted.
2. Check auditd / process accounting
   * If auditd was enabled, you can search for unlink / unlinkat or rename syscalls pointing to that filename:
   * sudo ausearch -f /path/to/file -i
   * sudo aureport --file --summary
   * If not enabled, you lose that trace retroactively.
3. If file was deleted at the filesystem level
   * Check for recent EBS snapshots, backups — recover file from snapshot if available.

C. Account-level checks

* Check CloudTrail Management events for console/API actions (e.g., TerminateInstances, DeleteVolume, DeleteObject, DeleteSnapshot) with timestamps and the IAM principal that performed them.

aws cloudtrail lookup-events --lookup-attributes AttributeKey=EventName,AttributeValue=DeleteObject

* List access key last used (for IAM users) to see which keys were active that day:

aws iam list-access-keys --user-name alice

aws iam get-access-key-last-used --access-key-id AKIA...

2) Likely outcome if everyone used the *same username/key*

* If everyone used the same private key to SSH into one OS account, logs will show the same account name only. You can get source IPs and session times — but not the person unless you correlate with e.g. corporate VPN logs, IP-to-user mapping, jump-host logs, or endpoint logs on their machines.
* If people assumed the same IAM role using STS with AssumeRole and passed unique sessionName or sourceIdentity, CloudTrail would show who assumed the role. But if they used a single shared access key or shared OS user, you likely cannot tie the action to an individual.

3) Short-term remediation actions (prevent further damage now)

1. Revoke the shared key immediately (disable the access key) or remove the shared SSH key from authorized\_keys.
2. Restore the deleted file from backups/snapshots/versions (S3 versioning or EBS snapshots).
3. Enable or centralize logging now:
   * Turn on CloudTrail *data events* for S3 and for Lambda/EFS as appropriate.
   * Enable SSM Session Manager with session logging to CloudWatch/S3 and enable CloudTrail StartSession logging.
   * Enable VPC Flow Logs and OS auditd, and configure CloudWatch Logs agent to ship logs centrally.

4) Long-term prevention — best practices (must-haves)

Apply these to achieve accountability and auditable actions:

Identity & Access

* No shared credentials. Never share PEM files or IAM access keys.
* Create individual IAM users or use AWS SSO / Identity Provider (IdP).
* Use IAM Roles for workloads and assume-role for user sessions; require sourceIdentity or sessionName when assuming roles.

Access to instances

* Use SSM Session Manager for SSH-equivalent access — it uses IAM identity and logs sessions (no shared SSH keys).
* If a bastion is required, enforce per-user SSH keys on the bastion, log sessions (ttyrec), or use jump host that records sessions.
* Disable password login and remove shared keys from authorized\_keys.

Auditing & Logging

* CloudTrail (management + data events for S3) — mandatory for attribution of API calls.
* SSM Session Manager session logs to CloudWatch/S3 and CloudTrail StartSession events.
* Enable auditd and forward logs to a central logging system (CloudWatch Logs / ELK).
* S3 Versioning & MFA Delete to recover and protect critical objects.
* Enable AWS Config for config change tracking.

Governance

* Use AWS Organizations + Firewall Manager for policy enforcement across accounts.
* Enforce MFA for critical operations and root account.
* Enforce least privilege and periodic access reviews.

5) Example ideal flow for future auditing (how it should have been done)

1. Each user logs in via corporate SSO and assume-role (CloudTrail records AssumeRole with sessionName).
2. They access EC2 via SSM Session Manager — each session is recorded and logged to CloudWatch/S3 and CloudTrail logs show IAM principal.
3. Any deletes (S3 DeleteObject or EC2 file deletes via SSM) appear in CloudTrail/SSM logs and you can map action → exact user.

Short scripted checklist you can run now

# 1) Find CloudTrail DeleteObject events in a time window

aws cloudtrail lookup-events --lookup-attributes AttributeKey=EventName,AttributeValue=DeleteObject \

--start-time "2025-10-25T00:00:00Z" --end-time "2025-10-30T23:59:59Z"

# 2) Find CloudTrail StartSession events (SSM)

aws cloudtrail lookup-events --lookup-attributes AttributeKey=EventName,AttributeValue=StartSession \

--start-time "..." --end-time "..."

# 3) Check EC2 status/auth logs for IPs/timestamps (on the instance)

sudo grep 'Accepted' /var/log/auth.log | tail -n 50

Final summary (one-liner for interviews)

“If a file was deleted and users shared a single key/account, attribution is difficult — you should check CloudTrail (S3 data & management events), S3 access logs, SSM Session Manager logs, and OS auditd / auth logs to correlate IPs and timestamps. The long-term fix is to stop sharing keys, use per-user IAM or SSO, use SSM Session Manager for instance access, enable CloudTrail data events and S3 versioning/MFA delete — that gives you clear auditability and the ability to recover deleted data.”

**38 \* what instances you are using in your project? why particular those instances?**

Great interview-style question 👏 — this one tests whether you understand EC2 instance types and how to choose the right instance for a project based on workload and cost.

Let’s answer it like you would in a real AWS project discussion 👇

🔹 Sample Answer (General Format)

“In our project, we use different types of EC2 instances based on workload.  
For the application layer, we use t3.medium and m5.large instances, and for the database, we use r5.large.  
The reason we selected these instance types is to balance performance, cost-efficiency, and workload requirements.”

🔹 Detailed Breakdown

🖥️ 1. Frontend / Application Layer

* Instance Type: t3.medium or t3.large
* Reason:
  + These are burstable performance instances → cost-effective for moderate workloads.
  + They automatically scale CPU when load increases.
  + Ideal for web servers, APIs, or microservices that do not need high constant CPU.

✅ Example Use Case:  
Hosting Nginx, Node.js, or Java application servers.

⚙️ 2. Backend / Middleware Layer

* Instance Type: m5.large or m5.xlarge
* Reason:
  + General-purpose compute with a balance of CPU, memory, and networking.
  + Good for application servers, background jobs, or middleware processing.
  + Consistent performance for stable workloads.

✅ Example Use Case:  
Spring Boot microservices, middleware logic servers.

🧠 3. Database Layer

* Instance Type: r5.large or r6g.large
* Reason:
  + “r” = memory-optimized → designed for high RAM workloads.
  + Perfect for RDS, MySQL, PostgreSQL, or MongoDB.
  + Provides better query performance and caching.

✅ Example Use Case:  
RDS MySQL or Aurora databases.

🧮 4. Analytics / Batch Processing

* Instance Type: c5.xlarge or c6g.xlarge
* Reason:
  + “c” = compute-optimized → more vCPUs, less cost per CPU.
  + Ideal for CPU-intensive jobs like data processing, analytics, and CI/CD builds.

✅ Example Use Case:  
ETL processing, data analysis, or machine learning preprocessing.

🎥 5. Caching / In-Memory Layer

* Instance Type: r6g.large (Graviton) or m6g.large
* Reason:
  + ARM-based Graviton processors → lower cost, higher performance.
  + Used with Redis / Memcached (ElastiCache) for low latency and high throughput.

✅ Example Use Case:  
Application caching, session storage.

🔹 Why These Specific Instances

| Criteria | Why chosen |
| --- | --- |
| Performance | Enough CPU/RAM for expected load |
| Cost Efficiency | Burstable (T series) or Graviton (G series) saves up to 40% |
| Scalability | Works well with Auto Scaling groups |
| Workload Fit | Each instance type matches workload (CPU-heavy, memory-heavy, etc.) |
| Region Availability | Commonly available in most AWS regions |

🧠 Short Interview Summary Answer

“We’re using t3.medium for frontend (cost-effective burstable compute),  
m5.large for backend (balanced compute), and r5.large for database (memory-optimized).  
We chose them based on workload needs — balancing cost, performance, and scalability.”

**39 \* customer is asking you to change the instances family to 2 ec2 instances which are part of the autoscalling groups siting on a load balancer & the condition is they are not looking for new instances and none of the instances to be deleted how your going to change the instance family of these 2 instances?**

Excellent question 👏 — this is a real-world AWS scenario about modifying EC2 instance family (for example, from t3.medium to t3.large) inside an Auto Scaling Group (ASG) without replacing or terminating the existing instances.

Let’s break it down step-by-step 👇

🧠 Scenario Summary

* 2 EC2 instances are part of an Auto Scaling Group (ASG).
* They are behind a Load Balancer.
* Customer wants to change the instance family (type).
* Condition: ❌ No new instances to be launched, ❌ No existing instances to be terminated.

⚙️ Key Concept

Normally, when you update an ASG launch template or configuration to a new instance type, AWS will:

Terminate old instances → Launch new ones.

But since termination/new launch is not allowed, we must manually modify the running instances while keeping ASG consistent.

✅ Solution Steps

Step 1: Suspend ASG processes temporarily

This prevents the ASG from terminating or replacing instances while we modify them.

aws autoscaling suspend-processes --auto-scaling-group-name <your-asg-name>

Processes you’re pausing include:  
Launch, Terminate, ReplaceUnhealthy, HealthCheck, AZRebalance, etc.

Step 2: Stop the EC2 instances manually

You can only change the instance type when the instance is stopped (not terminated).

aws ec2 stop-instances --instance-ids i-0123456789abcdef0 i-0abcdef1234567890

Wait until both instances are in stopped state.

Step 3: Change instance type (family)

Modify the instance type for each instance:

aws ec2 modify-instance-attribute \

--instance-id i-0123456789abcdef0 \

--instance-type "{\"Value\": \"t3.large\"}"

Do the same for the second instance.

Step 4: Start the instances again

aws ec2 start-instances --instance-ids i-0123456789abcdef0 i-0abcdef1234567890

Once started, they’ll automatically register back with the Load Balancer (since ASG manages them).

Step 5: Resume ASG processes

Re-enable Auto Scaling operations:

aws autoscaling resume-processes --auto-scaling-group-name <your-asg-name>

✅ Result

* Both instances now have the new instance family (type).
* No new instances were launched.
* No existing instances were terminated.
* Load Balancer health checks continue as before.

🧩 Extra Best Practice

Afterward, update the Launch Template / Launch Configuration in the ASG to reflect the new instance type.  
Otherwise, next time ASG scales up, it’ll create old-type instances again.

Example:

aws ec2 create-launch-template-version \

--launch-template-name MyTemplate \

--source-version 1 \

--launch-template-data '{"InstanceType": "t3.large"}'

aws autoscaling update-auto-scaling-group \

--auto-scaling-group-name MyASG \

--launch-template "LaunchTemplateName=MyTemplate,Version=$Latest"

🔁 Summary

| Step | Action | Purpose |
| --- | --- | --- |
| 1️⃣ | Suspend ASG processes | Prevent automatic termination |
| 2️⃣ | Stop EC2 instances | Required to change type |
| 3️⃣ | Modify instance type | Change instance family |
| 4️⃣ | Start instances | Bring back online |
| 5️⃣ | Resume ASG processes | Restore auto scaling |
| ✅ | Update launch template | Keep ASG future-proof |

**40 \* what kind of individual contribution load that you have played in your experiance on aws? what kind of issues you have resolved? what kind of troubleshooting you have performed?**

You can divide your answer into three parts:

1️⃣ Your Individual Contribution (what you did personally)

“In my AWS experience, I’ve been primarily responsible for designing, configuring, and troubleshooting key AWS infrastructure components.  
I’ve worked hands-on in areas like EC2 setup, VPC design, S3 configuration, IAM policy management, and monitoring using CloudWatch.  
I also participated in automation of deployments using scripts and helped ensure our environment’s security and high availability.”

2️⃣ What kind of issues you have resolved

Here’s a list of real-world AWS issues you can mention (pick 2–3 that match your comfort level):

| Category | Example Issue | Your Resolution |
| --- | --- | --- |
| EC2 / Network | EC2 instance not reachable | Checked Security Groups, NACLs, and Route Tables; found missing IGW route; fixed connectivity. |
| S3 | S3 access denied | Verified bucket policy, IAM policy, and ACL; fixed wrong principal in bucket policy. |
| VPC / Subnets | Instances unable to download packages | Found no NAT Gateway in private subnet; configured NAT Gateway + proper route. |
| Load Balancer | ELB showing unhealthy targets | Verified health check path and port, fixed wrong target group health check path /status. |
| Auto Scaling | Instances not launching | Debugged Launch Template IAM role and security group; fixed missing permission. |
| IAM | User not able to access specific service | Reviewed IAM policies, attached required managed policy with least privilege. |
| CloudWatch | Alarms not triggering | Checked metric filters and SNS topic subscriptions, recreated metric alarms correctly. |
| EBS | Disk full or slow performance | Used df -h, resized EBS volume, and extended partition using growpart and resize2fs. |
| RDS | Database connection timeout | Verified security groups, parameter groups, and subnet group; fixed VPC endpoint config. |

3️⃣ Troubleshooting Examples (detailed)

Give at least one detailed example (STAR method — Situation, Task, Action, Result):

“Once, we had a production EC2 instance that suddenly became unreachable.  
I logged into the console, checked system status — it was failing.  
I stopped and started the instance to move it to a healthy host, verified route tables, and security groups.  
Then, I implemented a CloudWatch alarm and Auto Recovery policy to ensure this doesn’t happen again.  
The service was restored within 15 minutes.”

Another example:

“In one case, S3 uploads from the application were failing. I checked CloudWatch logs and found AccessDenied errors.  
On review, the IAM role attached to the EC2 didn’t have s3:PutObject permission.  
I updated the policy, tested the flow, and uploads started working.  
Later, I documented the IAM permission model for future reference.”

✅ Optional Bonus: Mention Automation / Monitoring Work

“I also automated EC2 backups using Lambda + CloudWatch Events, and implemented CloudTrail + Config for auditing.  
For monitoring, I created custom metrics and alerts in CloudWatch to track CPU, memory, and EBS usage.”

🧠 Short Version (for quick interviews)

“In my AWS projects, I’ve individually handled EC2, S3, IAM, and VPC configuration.  
I’ve troubleshooted connectivity issues (security groups, routes), S3 permission issues, and Auto Scaling failures.  
I’ve also set up CloudWatch alarms and logging for proactive monitoring.  
My focus has been on ensuring security, availability, and automation — for example, using NAT Gateway for private instances to access the internet and configuring IAM roles for least privilege.”

**41 \* pre signed url in s3?**

A Pre-Signed URL is a temporary, secure link that allows a user to upload or download an object from an S3 bucket without requiring AWS credentials.

It’s generated by someone who already has permission to the S3 object (for example, using AWS CLI, SDK, or console).

🔹 Purpose

* To share S3 objects securely with people who do not have direct access to your AWS account.
* The URL is valid only for a limited time (you define the expiration time).

🔹 Use Cases

| Use Case | Description |
| --- | --- |
| Secure file download | Share an S3 file privately (e.g., reports, invoices) without making the bucket public |
| Secure file upload | Allow clients to upload files directly to S3 without giving them AWS credentials |
| Temporary access | Time-limited access for external users, apps, or partners |
| Mobile/web apps | Let users upload images, documents, etc., directly to S3 from browser/app |

🔹 How It Works (Conceptually)

1. An authorized user/app (with AWS credentials) generates a pre-signed URL.
2. The URL contains:
   * The object name
   * The bucket name
   * The permissions (GET / PUT)
   * A timestamp and signature
3. The recipient uses that URL to directly access S3 (for download or upload).
4. After the expiration time, the URL becomes invalid.

🔹 Example: Generate a Pre-Signed URL (Download)

Using AWS CLI:

aws s3 presign s3://my-bucket-name/myfile.txt --expires-in 3600

👉 This generates a link valid for 1 hour (3600 seconds).

You can share the generated URL with anyone —  
they can download the file via browser or curl.

🔹 Example: Generate a Pre-Signed URL (Upload)

Using Python (Boto3):

import boto3

s3 = boto3.client('s3')

url = s3.generate\_presigned\_url(

ClientMethod='put\_object',

Params={'Bucket': 'my-bucket', 'Key': 'upload/test.txt'},

ExpiresIn=600 # valid for 10 minutes

)

print(url)

The user can then upload a file using:

curl --upload-file test.txt "<paste\_pre\_signed\_url\_here>"

🔹 Important Points

* Pre-signed URLs inherit the permissions of the IAM user or role that generated them.
* The URL is valid only for the time period you define.
* No need to make the bucket public — keeps data secure.
* Common expiration time is between 5 minutes to a few hours (default max 7 days for SDKs).

🔹 Example Use Case in a Project

“In our project, we generate pre-signed URLs so users can securely upload profile pictures directly to S3 without exposing AWS credentials or making the bucket public.”

🧠 In short:

“A Pre-Signed URL is a time-limited, secure URL generated by an authorized user that allows someone else to upload or download objects from S3 without needing direct AWS credentials.”

**42 \* different routing policies in aws? which policy you have used for your project?**

🧭 Different Routing Policies in Amazon Route 53

| Routing Policy | Description / Use Case | Example Scenario |
| --- | --- | --- |
| 🎯 1. Simple Routing Policy | Used for a single resource that performs one function (no health checks or failover). | www.example.com → 1 IP (e.g., EC2 or ELB) |
| 🩺 2. Failover Routing Policy | Used to create an active-passive setup. If the primary endpoint fails, traffic goes to the secondary. | Primary site in Mumbai, standby in Singapore. |
| ⚖️ 3. Weighted Routing Policy | Used to distribute traffic across multiple resources based on weights (percentages). | Send 70% of traffic to mumbai.example.com, 30% to singapore.example.com for testing new app version. |
| 🌍 4. Latency-based Routing Policy | Routes users to the region that provides the lowest latency. | A user in India hits Mumbai region, a user in US hits Ohio region automatically. |
| 🏠 5. Geolocation Routing Policy | Routes traffic based on user’s geographic location (country/continent/state). | Users from India → in.example.com, users from US → us.example.com. |
| 📍 6. Geoproximity Routing Policy *(Traffic Flow only)* | Routes traffic based on geographic proximity of resources and allows you to bias traffic toward one region. | 80% traffic to Mumbai region, 20% to Singapore — even though both are near. |
| 🔀 7. Multi-Value Answer Routing Policy | Returns multiple healthy IPs in response to DNS queries (basic load balancing). | Returns 2-3 EC2 IPs, and client picks one at random (with health checks). |

💼 Which Policy I Used in My Project

✅ Latency-based Routing Policy

🔹 Why I used it:

* We had multi-region deployment — India (ap-south-1) and Singapore (ap-southeast-1).
* Needed users to connect to the nearest AWS region for low latency and better performance.
* Combined it with health checks to automatically remove unhealthy endpoints.

🔹 Example:

example.com (Hosted Zone)

├── Latency record → ALB in ap-south-1 (Mumbai)

└── Latency record → ALB in ap-southeast-1 (Singapore)

When a user from India queries DNS → Route 53 returns Mumbai ALB.  
When a user from Malaysia → gets Singapore ALB.

🧠 Summary Table

| Policy | Best For | Supports Health Check |
| --- | --- | --- |
| Simple | Single endpoint | ❌ |
| Failover | DR setup (Active-Passive) | ✅ |
| Weighted | Traffic testing / Gradual rollout | ✅ |
| Latency | Multi-region performance | ✅ |
| Geolocation | Regional content | ✅ |
| Geoproximity | Advanced regional traffic control | ✅ (Traffic Flow only) |
| Multi-Value | Simple load balancing | ✅ |

**43 \* diff b/w cname and alias?**

Here’s the difference between CNAME and Alias records (especially in the context of AWS Route 53 👇):

| Feature | CNAME Record | Alias Record |
| --- | --- | --- |
| Definition | Maps one domain name to another domain name. | AWS-specific record that maps a domain name to an AWS resource. |
| Works With | Any DNS provider (standard DNS record). | Only in Amazon Route 53. |
| Root Domain (Zone Apex) | ❌ Cannot be used at the root domain (e.g., example.com). | ✅ Can be used at the root domain. |
| Targets | Another domain name only (e.g., www.example.com → example.com). | AWS resources like CloudFront, Elastic Load Balancer, S3 website endpoint, or another record in the same hosted zone. |
| DNS Query Cost | Normal DNS query; doesn’t optimize AWS lookups. | No extra DNS query — Route 53 returns the target’s IP directly. |
| Example | www.example.com CNAME example.com | example.com ALIAS d1234.cloudfront.net |

🟢 Summary:

* Use CNAME → when pointing to another domain name (not AWS-specific).
* Use Alias → when pointing to AWS resources and you need it to work at the root domain level (e.g., example.com → CloudFront).

**Top of Form**

**Bottom of Form**

**44 \* what record used for dns to ip?**

he record that maps a domain name to an IP address is called an A record (Address Record) or AAAA record (for IPv6).

🧩 1️⃣ A Record (Address Record)

* Purpose: Maps a domain name → IPv4 address
* Example:
* example.com. IN A 192.168.1.10

👉 When a user types example.com, DNS resolves it to the IP 192.168.1.10.

🧩 2️⃣ AAAA Record (Quad-A Record)

* Purpose: Maps a domain name → IPv6 address
* Example:
* example.com. IN AAAA 2001:0db8:85a3::8a2e:0370:7334

🧩 3️⃣ Summary Table

| Record Type | Maps To | Protocol Version | Example IP |
| --- | --- | --- | --- |
| A Record | Domain → IPv4 | IPv4 | 192.168.1.10 |
| AAAA Record | Domain → IPv6 | IPv6 | 2405:204:1234::abcd |

💬 In Short Answer (for interviews):

“The A record in DNS is used to map a domain name to an IPv4 address, and the AAAA record is used to map it to an IPv6 address.”

**45 \* any experiance with lambda? any experiance inwriting lambda functions?**

Perfect 👍 — this is a common AWS interview question to test your hands-on experience with AWS Lambda (serverless computing).

Here’s how to answer it confidently 👇

🔹 Sample Answer (Interview Style)

“Yes, I have hands-on experience working with AWS Lambda functions.  
I’ve written and deployed Lambda functions in my project for automating backend tasks like S3 event processing, sending notifications, and integrating APIs.  
I usually write them in Python (sometimes Node.js), and trigger them using S3 events, CloudWatch schedules, or API Gateway.”

🔹 What is AWS Lambda?

AWS Lambda is a serverless compute service that lets you run your code without managing servers.  
You just upload your function code, and AWS takes care of:

* Provisioning servers
* Scaling automatically
* High availability

You only pay for the compute time your code actually runs.

🔹 Example: Lambda Use Cases You Can Mention

| Use Case | Trigger | Description |
| --- | --- | --- |
| S3 File Processing | S3 Event | Triggered when a file is uploaded — e.g., resize images, validate CSVs, etc. |
| Scheduled Tasks | CloudWatch Event / EventBridge | Run cleanup jobs, reports, or backups on schedule |
| API Backend | API Gateway | Build serverless REST APIs |
| SNS / SQS Consumer | SNS/SQS Event | Process incoming messages asynchronously |
| CloudWatch Alert Handler | CloudWatch Alarm | Trigger notifications or recovery actions |

🔹 Example: Lambda Function (Python)

Trigger → S3 bucket upload event

This function sends a log message whenever a file is uploaded.

import json

def lambda\_handler(event, context):

# Get bucket and object details from event

bucket = event['Records'][0]['s3']['bucket']['name']

key = event['Records'][0]['s3']['object']['key']

print(f"File uploaded: {key} in bucket: {bucket}")

# Example: custom logic (e.g., validation, SNS trigger, etc.)

return {

'statusCode': 200,

'body': json.dumps(f"Processed file: {key}")

}

✅ Trigger: S3 → ObjectCreated event  
✅ Permission: Lambda IAM role with s3:GetObject access

🔹 Example: Scheduled Lambda (every day at midnight)

* Trigger: CloudWatch Event rule (cron schedule)
* Purpose: Backup DynamoDB → S3 or clear logs

🔹 Languages I’ve Used

* Python → for automation, data validation, AWS SDK (boto3)
* Node.js → for lightweight API handlers or JSON manipulation

🔹 Integrations I’ve Worked With

* API Gateway → expose Lambda as REST API
* S3 → file processing on upload
* DynamoDB → CRUD operations via Lambda
* SNS/SQS → messaging and notification systems
* CloudWatch → monitoring, logs, and scheduled triggers

🔹 Best Practices I Follow

* Use environment variables for configuration
* Use IAM roles with least privileges
* Write modular, stateless functions
* Enable CloudWatch Logs for monitoring
* Package dependencies with Lambda Layers (for Python/Node.js)

🧠 Short Interview Summary Answer

“Yes, I’ve written and deployed multiple Lambda functions — mostly in Python — for S3 automation, CloudWatch scheduled jobs, and API Gateway integrations.  
I follow AWS best practices like using IAM roles, environment variables, and logging with CloudWatch.”

**46 \* what is API gateway why we need it ?**

🌐 What is an API Gateway?

An API Gateway is a managed service that acts as a front door for your backend applications (like AWS Lambda, EC2, or microservices).  
It’s the entry point where all API requests from clients (web apps, mobile apps, etc.) come in.

In AWS, this service is called Amazon API Gateway.

🧩 Simple Example

Imagine you have several backend services:

/users → Lambda 1

/orders → Lambda 2

/payments → EC2 backend

Instead of exposing each backend directly, you create one API Gateway endpoint:

https://api.myapp.com/

API Gateway receives the request → routes it to the correct backend → returns the response.

🔒 Why We Need API Gateway

| Reason | Description |
| --- | --- |
| 🌐 Single Entry Point | All your APIs are accessible through one public endpoint, simplifying client access. |
| 🔐 Security & Authorization | Integrates with AWS IAM, Cognito, or custom authorizers (JWT, OAuth) to control access. |
| 📊 Throttling & Rate Limiting | Protects your backend from too many requests or DDoS-like spikes. |
| 💰 Cost Efficiency | You pay only for requests served (serverless model). |
| 🔄 Request Transformation | You can modify headers, parameters, and payloads before sending to backend. |
| 📈 Monitoring & Logging | Integrates with CloudWatch for request logs, latency, and metrics. |
| 🧱 Integration with AWS Services | Directly connect to Lambda, Step Functions, DynamoDB, or HTTP endpoints without managing servers. |
| 🧩 Versioning & Staging | Supports multiple API versions (e.g., /v1, /v2) and stages (dev, test, prod). |

⚙️ How It Works (Flow)

Client (Mobile/Web)

↓

API Gateway

↓

Lambda / EC2 / Fargate / DynamoDB

↓

Response sent back to Client

💼 Example Use Case (Project Example)

✅ In my project, we used AWS API Gateway + Lambda + DynamoDB to create a serverless REST API for user management.

* API Gateway exposed endpoints like /register, /login, /profile.
* Each route triggered a Lambda function.
* Lambda interacted with DynamoDB to store user data.
* API Gateway handled authentication using Cognito JWT tokens.

Benefits:

* No servers to manage
* Highly scalable
* Secure (Cognito + IAM)
* Pay per request

🧠 In Short

| Feature | API Gateway Provides |
| --- | --- |
| Entry point | For all client requests |
| Security | Authentication + Authorization |
| Scalability | Auto scales globally |
| Cost | Pay per request |
| Monitoring | CloudWatch integration |

**47 \* How you bind role to account ( script explain)**

🧩 1. Concept

* IAM Role: Defines a set of permissions.
* Binding (Attaching): Means associating that role with an AWS resource or entity (like EC2, Lambda, or a user).

🐍 Example: Bind (Attach) an IAM Role to an EC2 Instance Using AWS CLI Script

Step 1: Create an IAM Role

aws iam create-role \

--role-name MyEC2Role \

--assume-role-policy-document file://trust-policy.json

Example trust-policy.json:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": { "Service": "ec2.amazonaws.com" },

"Action": "sts:AssumeRole"

}

]

}

This allows EC2 to assume the role.

Step 2: Attach a Policy to the Role

aws iam attach-role-policy \

--role-name MyEC2Role \

--policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess

✅ This gives the role read-only access to S3.

Step 3: Create an Instance Profile (acts as a bridge)

aws iam create-instance-profile \

--instance-profile-name MyEC2InstanceProfile

Then add the role to it:

aws iam add-role-to-instance-profile \

--instance-profile-name MyEC2InstanceProfile \

--role-name MyEC2Role

Step 4: Attach Instance Profile to EC2 Instance

aws ec2 associate-iam-instance-profile \

--instance-id i-0123456789abcdef0 \

--iam-instance-profile Name="MyEC2InstanceProfile"

✅ Now the EC2 instance has the IAM Role permissions bound to it.

🧠 Summary:

| Step | Action | Command |
| --- | --- | --- |
| 1 | Create Role | aws iam create-role |
| 2 | Attach Policy | aws iam attach-role-policy |
| 3 | Create Instance Profile | aws iam create-instance-profile |
| 4 | Add Role to Profile | aws iam add-role-to-instance-profile |
| 5 | Attach Profile to EC2 | aws ec2 associate-iam-instance-profile |

**48 \* how you configure autoscaling and cloud load balancer**

🧩 1️⃣ Objective

To automatically adjust the number of EC2 instances based on traffic/load and distribute incoming traffic evenly across all healthy instances.

🏗️ 2️⃣ Steps to Configure Auto Scaling with Load Balancer

Step 1 — Create a Launch Template or Launch Configuration

* Go to EC2 → Launch Templates
* Define:
  + AMI (OS image)
  + Instance type (e.g. t3.medium)
  + Key pair
  + Security group (allow required ports like 22, 80, 443)
  + IAM role (if needed for S3, CloudWatch, etc.)
  + User data script (for auto-installing web server/app)

✅ Example user data:

#!/bin/bash

yum update -y

yum install -y httpd

systemctl enable httpd

systemctl start httpd

echo "Hello from $(hostname)" > /var/www/html/index.html

Step 2 — Create an Auto Scaling Group (ASG)

* Navigate to EC2 → Auto Scaling Groups
* Choose the launch template/configuration you just made
* Select your VPC and subnets (usually across 2 AZs for HA)
* Attach to an existing Load Balancer Target Group (next step)
* Define Group Size:
  + Minimum = 2
  + Desired = 2
  + Maximum = 5
* Set Health Checks:
  + Select ELB + EC2 health checks
* Configure Scaling Policies:
  + Example: scale out if average CPU > 70% for 5 mins
  + scale in if average CPU < 30% for 10 mins

Step 3 — Create a Load Balancer (ELB)

* Go to EC2 → Load Balancers → Create Load Balancer
* Choose type:
  + Application Load Balancer (ALB) – for HTTP/HTTPS (Layer 7)
  + Network Load Balancer (NLB) – for TCP/UDP (Layer 4)
  + Gateway Load Balancer (GWLB) – for network appliances

Example (ALB):

* Scheme: Internet-facing
* Listener: Port 80 (HTTP)
* Select at least 2 subnets (different AZs) for high availability
* Create Target Group and register it (ASG will auto-register instances)

Step 4 — Attach Load Balancer to Auto Scaling Group

* In ASG, under “Load balancing”, attach the Target Group from your ALB.
* ASG automatically registers/deregisters instances with the target group.

Step 5 — Test

* Access ALB DNS name (e.g. my-alb-123456.elb.amazonaws.com)
* You’ll see requests distributed across EC2s
* Try stress test → ASG should scale out
* After traffic drops → ASG should scale in

🧩 3️⃣ Monitoring & Logging

* CloudWatch Metrics:
  + CPUUtilization, GroupInServiceInstances, GroupDesiredCapacity
* CloudTrail: tracks scaling events
* Notifications: SNS alerts on scale events

🧩 4️⃣ Example Architecture

Internet

|

↓

[Application Load Balancer]

|

↓

[Auto Scaling Group] → Multiple EC2 instances across AZs

|

↓

[VPC with Public/Private Subnets]

💬 Interview Answer (Concise Version):

“I configure a Launch Template with EC2 details and a user data script, then create an Auto Scaling Group with desired/min/max capacity.  
I attach the group to an Application Load Balancer that distributes traffic evenly across all healthy instances.  
Scaling policies are configured based on CloudWatch metrics like CPU utilization to automatically add or remove instances.  
This setup ensures high availability, fault tolerance, and automatic scaling based on demand.”

**49 \* Where your application is running EC2 or EKS**

💬 Sample Direct Answer (Scenario-Based):

“Our application is currently running on EKS (Elastic Kubernetes Service) because it is containerized using Docker and requires scalability, self-healing, and easy deployment management.  
However, in earlier stages, we deployed on EC2 instances before migrating to EKS for better orchestration and automation.”

🧩 If Your Application Is Running on EC2:

You can say:

“Our application runs on EC2 instances behind an Application Load Balancer (ALB).  
We manage scaling using Auto Scaling Groups (ASG) and configuration management through tools like Ansible or AWS Systems Manager.  
We chose EC2 because the application is monolithic and not yet containerized.”

🧠 When EC2 is preferred:

* Monolithic apps (single large application)
* No containerization
* Simple setup
* Full control over OS and configurations
* Easier for traditional workloads (Java, .NET, etc.)

🧩 If Your Application Is Running on EKS:

You can say:

“We run our application on EKS (Elastic Kubernetes Service) using Docker containers.  
We use Node Groups (EC2 or Fargate) as worker nodes, ALB ingress controllers for routing traffic, and CloudWatch + Prometheus for monitoring.  
This gives us high scalability, portability, and efficient resource utilization.”

🧠 When EKS is preferred:

* Microservices architecture
* CI/CD pipelines using Jenkins, GitHub Actions, etc.
* Need for container orchestration (auto scaling, rolling updates)
* Faster deployments and rollback
* Easier to manage large-scale distributed systems

🔁 Example Migration Scenario:

“Initially, we hosted the app on EC2 with Auto Scaling and ALB, but as traffic grew and the application was containerized, we moved to EKS.  
It allowed us to manage multiple microservices efficiently using Kubernetes deployments, services, and ingress controllers.”

🧩 Short Interview Summary:

| Deployment Type | Used When | Tools Used |
| --- | --- | --- |
| EC2 | Monolithic apps, simple scaling | ALB, ASG, CloudWatch |
| EKS | Containerized/microservice apps | Docker, Kubernetes, Ingress, Helm, CloudWatch, Prometheus |

🗣️ Concise 15-sec Interview Version:

“Our application currently runs on EKS, as we use Docker containers and need Kubernetes orchestration for auto-scaling and high availability.  
Previously, we deployed on EC2 using ALB and ASG before migrating to a containerized environment.”

**50 \* Why your team went manual instead of using EKS**

Excellent 💡 — this is a strategic interview question about why your team didn’t choose Kubernetes (EKS) and instead went with a manual or simpler deployment approach.

Let’s break down how to answer it professionally and logically 👇

🔹 Sample Interview Answer

“Our team initially evaluated EKS, but we decided to go with a manual or simpler deployment approach using EC2 Auto Scaling and Load Balancers instead.  
The main reasons were cost, complexity, and our application’s nature, which didn’t require full container orchestration at that stage.”

🔹 Detailed Justification (Structured Answer)

🧩 1. Project Complexity Didn’t Justify EKS

* Our application was monolithic or lightly microservice-based, so we didn’t need advanced orchestration features like service mesh, sidecar injection, or rolling updates across multiple services.
* EKS adds management overhead — cluster setup, node groups, IAM roles, networking policies — which didn’t add much value for a small to mid-sized app.

*Example:* “We had only 3–4 microservices — managing them with EC2 + ALB + Auto Scaling was simpler.”

💰 2. Cost Optimization

* EKS adds extra control plane cost (~$70/month per cluster) plus compute costs.
* We were focused on cost-efficiency and could achieve scaling with Auto Scaling Groups and Load Balancers without additional expenses.

*Example:* “For our moderate traffic, EC2 Auto Scaling handled peak loads well without EKS overhead.”

⚙️ 3. Team Skill Set

* At that time, not everyone on the team was proficient with Kubernetes concepts (pods, deployments, ingress, Helm, etc.).
* Using EC2 or Elastic Beanstalk allowed faster onboarding and simpler CI/CD pipelines via CodeDeploy.

*Example:* “We prioritized delivery speed and maintainability over advanced orchestration.”

🧠 4. Operational Simplicity

* With EC2 + ALB + CloudWatch:
  + Easier monitoring and debugging
  + No cluster maintenance
  + No YAML or Helm configuration overhead
* Simplified rollback and version management with AMIs or CodeDeploy.

*Example:* “A simple blue-green deployment strategy on EC2 met our availability goals.”

🔐 5. Security & Networking Simplicity

* EKS networking (CNI, service discovery, ingress controllers) can be complex to secure.
* A manual setup allowed easier control of:
  + Security groups
  + IAM roles
  + VPC subnet placement

📈 6. Future Plan

“We’re not against EKS — we plan to migrate when we have more microservices or need container-level scaling and portability.”

✅ Summary Answer (Short Version for Interview)

“We evaluated EKS but chose a manual setup with EC2 Auto Scaling and Load Balancer because our workload was not complex enough to justify Kubernetes overhead.  
It reduced cost, simplified management, and matched our team’s skill set.  
We plan to adopt EKS in the future when our architecture scales to multiple microservices.”

**51 \* Security group ,nacl**

🔐 Security Group (SG)

Definition:  
A virtual firewall that controls inbound and outbound traffic for EC2 instances (or ENIs).

✅ Key Points

| Feature | Description |
| --- | --- |
| Level | Operates at instance level (ENI level). |
| Stateful | ✅ Yes — if you allow inbound traffic, response is automatically allowed outbound. |
| Rules | You define allow rules only (no deny). |
| Applies To | EC2, ENI, RDS, Lambda (VPC), Load Balancers, etc. |
| Default Behavior | By default, all inbound is denied, outbound is allowed. |
| Example | Allow inbound on port 22 (SSH) and 80 (HTTP). |

🧱 Example:

Inbound:

Port 22 → Allow from 203.0.113.0/24

Port 80 → Allow from 0.0.0.0/0

Outbound:

All traffic → Allow

🌐 Network ACL (NACL)

Definition:  
A stateless firewall that controls traffic at the subnet level in a VPC.

✅ Key Points

| Feature | Description |
| --- | --- |
| Level | Operates at subnet level. |
| Stateful | ❌ No — responses must be explicitly allowed in both directions. |
| Rules | Supports both allow and deny rules. |
| Applies To | All instances within a subnet. |
| Default Behavior | Default NACL allows all inbound/outbound. Custom NACL denies all by default. |
| Rule Order | Rules are processed in ascending order (lowest number first). |
| Example | Allow inbound 80, deny all others. |

🧱 Example:

Inbound Rules:

Rule 100 → Allow TCP 80 (0.0.0.0/0)

Rule 110 → Deny All

Outbound Rules:

Rule 100 → Allow TCP 80 (0.0.0.0/0)

Rule 110 → Deny All

⚙️ SG vs NACL — Quick Comparison

| Feature | Security Group (SG) | Network ACL (NACL) |
| --- | --- | --- |
| Scope | Instance (ENI) level | Subnet level |
| State | Stateful | Stateless |
| Allow / Deny | Only Allow | Allow and Deny |
| Default Inbound | Deny all | Allow all (default NACL) |
| Default Outbound | Allow all | Allow all (default NACL) |
| Applies To | Specific instance/ENI | Entire subnet |
| Evaluation | All rules evaluated | Rules evaluated in order |
| Use Case | Control traffic to/from EC2 | Additional subnet-level protection |

💼 In My Project:

✅ We used Security Groups as the main layer of security  
and NACLs for an additional layer of subnet-level control.

Example:

* Public Subnet:
  + NACL allows inbound HTTP(80), HTTPS(443), SSH(22)
* Private Subnet:
  + NACL denies all inbound from internet, only allows traffic from ALB.
* Security Groups:
  + Web SG: Allows 80/443 from internet
  + App SG: Allows 8080 from Web SG
  + DB SG: Allows 3306 from App SG

This creates a tiered security architecture.

🧠 In Short

| Concept | Think of it as… |
| --- | --- |
| Security Group | Door lock on each room (instance-level) |
| NACL | Gate at the community entrance (subnet-level) |

**52 \* Ecs**

🐳 What is Amazon ECS?

ECS (Elastic Container Service) is a fully managed container orchestration service by AWS.  
It lets you run, stop, and manage Docker containers on a cluster of EC2 instances or using AWS Fargate (serverless).

⚙️ Core Components of ECS

| Component | Description |
| --- | --- |
| Cluster | A logical group of EC2 instances or Fargate tasks where your containers run. |
| Task Definition | A JSON template that defines how your container should run — image, CPU, memory, ports, environment variables, etc. |
| Task | A running instance of a task definition (like one containerized app running). |
| Service | Ensures that a specified number of tasks are always running and can connect them to a Load Balancer. |
| Container | The actual Docker container running inside the task. |
| Launch Type | Determines how containers are deployed:  • EC2 → runs on your own EC2 instances.  • Fargate → runs serverless (no EC2 management). |

🚀 Two Ways to Run ECS

1. ECS on EC2 Launch Type
   * You manage the EC2 instances yourself.
   * ECS schedules and runs containers on your cluster.
2. ECS with Fargate Launch Type
   * No need to manage EC2 servers.
   * You just define CPU, memory, and container image — AWS runs it for you.

🧱 ECS Architecture Overview

ECS Cluster

├── Service

│ ├── Task Definition

│ │ └── Container (Docker image)

│ └── Load Balancer (Optional)

└── EC2 Instances or Fargate

🛠️ Common Use Cases

* Running microservices.
* Deploying containerized applications.
* Running batch processing jobs.
* Hosting API services behind a load balancer.

💡 Example Command (AWS CLI)

Create a cluster:

aws ecs create-cluster --cluster-name MyAppCluster

Register a task definition:

aws ecs register-task-definition \

--family my-web-task \

--container-definitions file://container-def.json

Run the task:

aws ecs run-task \

--cluster MyAppCluster \

--launch-type FARGATE \

--task-definition my-web-task \

--network-configuration "awsvpcConfiguration={subnets=[subnet-xxxx],securityGroups=[sg-xxxx],assignPublicIp=ENABLED}"

🧠 Key Differences vs Other Services

| ECS | EKS | Docker on EC2 |
| --- | --- | --- |
| AWS managed container orchestration | Kubernetes service on AWS | Manual container deployment |
| Easier setup | More flexible but complex | No orchestration features |

**53 \* Ec2 s3 iam roles**

🧩 1️⃣ What is an IAM Role?

An IAM Role is a set of permissions that AWS services or users can assume temporarily to perform specific actions.  
Unlike IAM Users, roles don’t have passwords or access keys — they are assigned dynamically.

🧩 2️⃣ Why EC2 Needs an IAM Role

An EC2 instance may need to:

* Access S3 buckets
* Read/write to DynamoDB
* Pull images from ECR
* Send logs to CloudWatch
* Call other AWS APIs

Instead of storing AWS access keys inside the instance (which is insecure),  
you attach an IAM Role to EC2 — and that role gives it the necessary permissions safely.

🧩 3️⃣ Example Scenario — EC2 to Access S3

✅ Step-by-step Configuration:

Step 1: Create IAM Role

* Go to IAM → Roles → Create role
* Select Trusted Entity: “AWS service”
* Choose Use Case: “EC2”
* Attach a policy like:
  + AmazonS3ReadOnlyAccess (for read access)
  + or custom policy (for specific S3 actions)

Example Policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": ["s3:GetObject", "s3:ListBucket"],

"Resource": ["arn:aws:s3:::mybucket", "arn:aws:s3:::mybucket/\*"]

}

]

}

Step 2: Attach Role to EC2

* While launching the instance, under IAM Role, choose the role created.
* If instance is already running → Actions → Security → Modify IAM Role → Attach Role

Step 3: Test Access from EC2

Once inside EC2:

aws s3 ls s3://mybucket

aws s3 cp /tmp/test.txt s3://mybucket/

These commands will work without configuring access keys (thanks to IAM Role).

🧩 4️⃣ How It Works Internally

* When EC2 boots, it gets temporary credentials via the Instance Metadata Service (IMDS).
* These credentials are automatically rotated and used by the AWS CLI/SDK.
* So, EC2 can securely call AWS APIs (like S3) without manual keys.

🧩 5️⃣ Benefits

✅ No hardcoded credentials  
✅ Automatically rotated credentials  
✅ Least privilege principle (assign only needed permissions)  
✅ Secure communication between AWS services

🧠 In Interview: Short Answer Example

“We use IAM roles to allow our EC2 instances to securely access AWS resources like S3 without using access keys.  
The role is attached to the instance and grants only required permissions (for example, AmazonS3ReadOnlyAccess).  
This ensures security and follows AWS best practices.”

🔁 Real-World Example

“In one of our projects, our web application running on EC2 needed to upload logs and images to S3.  
We created an IAM Role with PutObject and GetObject permissions for that S3 bucket and attached it to the EC2.  
The application used the AWS SDK to perform uploads automatically without storing any credentials.”

**54 \* Lambda**

🧩 1️⃣ What is AWS Lambda?

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers.  
You just upload your code, and AWS Lambda runs it automatically when triggered, scaling up or down as needed.

⚙️ 2️⃣ How It Works

1. You write your code (in Python, Node.js, Java, Go, etc.)
2. Define a trigger (e.g., S3 upload, API Gateway call, CloudWatch event)
3. Lambda automatically:
   * Runs your code only when needed
   * Allocates resources dynamically
   * Charges you only for compute time used (per ms)

💡 3️⃣ Example Scenario

Use Case: When a file is uploaded to an S3 bucket → automatically resize the image.

Flow:

1. S3 event (file upload) → triggers Lambda function
2. Lambda function processes the image (resizes it)
3. Lambda saves output to another S3 bucket

🧩 4️⃣ Common Use Cases

| Use Case | Example |
| --- | --- |
| S3 Events | Trigger when file uploaded (image processing, log parsing) |
| API Gateway | Run backend code for serverless APIs |
| CloudWatch Events | Scheduled tasks (cron jobs, cleanup scripts) |
| DynamoDB Streams | Process data changes in DynamoDB |
| SNS / SQS | Respond to messages or notifications |
| EventBridge | Serverless automation workflows |

🧠 5️⃣ Key Features

* Serverless — no EC2 management
* Auto-scaling — runs multiple executions parallelly
* Pay per use — billed by execution time (ms)
* Integrated with 200+ AWS services
* Supports many runtimes — Python, Node.js, Java, Go, .NET, etc.

⚙️ 6️⃣ Lambda Configuration Components

| Component | Description |
| --- | --- |
| Function Code | Your business logic |
| Trigger | Event source (S3, API Gateway, etc.) |
| Execution Role (IAM Role) | Grants permission to access other AWS services |
| Environment Variables | For config values |
| Timeout | Max runtime (default 3s, max 15 mins) |
| Memory | 128 MB – 10 GB |
| Concurrency | No. of parallel executions |

🧩 7️⃣ Example (Simple Python Lambda Function)

import json

def lambda\_handler(event, context):

name = event.get('name', 'Guest')

return {

'statusCode': 200,

'body': json.dumps(f'Hello, {name} from Lambda!')

}

🧠 8️⃣ Example Interview Answer (Concise Version)

“AWS Lambda is a serverless compute service that runs code in response to events without provisioning servers.  
We used it for automating tasks like resizing images when uploaded to S3 and scheduling jobs with CloudWatch.  
It scales automatically, is cost-effective, and integrates easily with other AWS services.”

⚙️ 9️⃣ Example Architecture

S3 → Lambda → DynamoDB → SNS  
(When new data is uploaded → processed → stored → notification sent)

**55 \* aws instances**

An EC2 instance is a virtual machine that runs on AWS infrastructure.  
You can launch it with your choice of:

* Operating system (Linux/Windows),
* Instance type (CPU, RAM, storage),
* Network settings, and
* Storage options (EBS, Instance Store).

You pay only for the time it runs.

🔹 Types of AWS Instances (Families)

AWS categorizes instances into families based on their use cases 👇

| Instance Family | Key Feature | Use Case Example |
| --- | --- | --- |
| General Purpose (t, m series) | Balanced CPU, memory | Web servers, dev/test environments |
| Compute Optimized (c series) | High CPU power | High-performance compute, analytics, CI/CD builds |
| Memory Optimized (r, x, z series) | High RAM | Databases, caching, in-memory processing |
| Storage Optimized (i, d, h series) | High storage throughput | Big data, NoSQL, data warehousing |
| Accelerated Computing (p, g, inf, trn series) | GPUs / ML accelerators | Machine learning, deep learning, video rendering |

🔹 Commonly Used Instance Types (Examples)

| Instance Type | vCPUs | Memory (GB) | Use Case |
| --- | --- | --- | --- |
| t3.micro | 2 | 1 | Free tier, small web app |
| t3.medium | 2 | 4 | Web/API server, low traffic |
| m5.large | 2 | 8 | App servers, backend services |
| c5.xlarge | 4 | 8 | Compute-heavy workloads |
| r5.large | 2 | 16 | Databases, caching |
| p3.2xlarge | 8 | 61 | AI/ML model training (GPU) |

🔹 Instance Lifecycle

1. Launch instance → choose AMI, type, key pair, and network.
2. Configure storage, security group, IAM role.
3. Connect via SSH (Linux) or RDP (Windows).
4. Monitor with CloudWatch.
5. Stop/Start/Terminate when done.

🔹 Pricing Models

| Model | Description | Example Use |
| --- | --- | --- |
| On-Demand | Pay hourly/second — no commitment | Short-term workloads |
| Reserved Instances | 1 or 3-year commitment → up to 75% discount | Steady-state workloads |
| Spot Instances | Bid on unused capacity → up to 90% cheaper | Batch jobs, non-critical workloads |
| Savings Plans | Flexible compute savings over time | Predictable usage |
| Dedicated Hosts | Physical isolation for compliance | Regulated industries |

🔹 Storage Options

* EBS (Elastic Block Store) → Persistent storage (survives instance stop/start)
* Instance Store → Temporary storage (deleted when instance stops)

🔹 Example Project Setup

“In our project, we use t3.medium instances for frontend servers and r5.large for database servers.  
Auto Scaling and Load Balancing handle traffic spikes, and we use CloudWatch for monitoring performance.”

🧠 In short:

“AWS EC2 instances are virtual servers in the cloud, available in different types (general, compute, memory, storage optimized) — you choose based on your application’s performance and cost needs.”

**56 \* difference between nat instance and nat gateway**

Excellent 👏 — this is another very common AWS networking interview question.

Let’s break down the difference between a NAT Instance and a NAT Gateway in simple, clear terms 👇

🌐 Purpose of NAT (Network Address Translation)

NAT allows private subnet instances (without public IPs) to access the Internet (for software updates, API calls, etc.)  
➡️ BUT it prevents the Internet from initiating connections back into those instances.

So — outbound only, no inbound.

🧱 Two Ways AWS Provides NAT Functionality

| Type | Description |
| --- | --- |
| 🧩 NAT Instance | A custom EC2 instance configured to perform NAT. |
| ⚡ NAT Gateway | A managed AWS service that automatically handles NAT — no need to manage servers. |

⚙️ Detailed Comparison: NAT Instance vs NAT Gateway

| Feature | NAT Instance | NAT Gateway |
| --- | --- | --- |
| Type | EC2 instance (you manage it) | AWS-managed service |
| Scalability | Manual — you must choose instance size and scale yourself | Automatic — scales up to 45 Gbps |
| Availability | Single instance → SPOF unless you configure HA manually | Highly available in one AZ; can create multiple for multi-AZ HA |
| Performance | Depends on instance type (t2.micro, m5.large, etc.) | Much faster — AWS handles scaling |
| Maintenance | You must patch, update, monitor instance | Fully managed by AWS |
| Security Groups | Can attach and control with Security Groups | Uses NACLs (no Security Groups) |
| Cost | Cheaper for very small workloads | Higher hourly cost but lower ops overhead |
| Elastic IP | Must assign manually | AWS assigns automatically (you can attach your own EIP) |
| Logging / Monitoring | Manual setup (CloudWatch agent) | Integrated with CloudWatch metrics |
| Bandwidth Limit | Limited by EC2 instance type | Up to 45 Gbps per NAT Gateway |
| Redundancy | Must manually configure in multiple AZs | Create one per AZ for fault tolerance |

🧩 When to Use Which

| Use Case | Recommended Option |
| --- | --- |
| Small test/lab environment | ✅ NAT Instance (low cost, flexible) |
| Production / high traffic | ✅ NAT Gateway (scalable, managed, reliable) |
| Need to control with custom firewall rules | NAT Instance |
| Want no maintenance, auto scaling | NAT Gateway |

🧠 Example Architecture

VPC

├── Public Subnet (has Internet Gateway)

│ └── NAT Gateway (Elastic IP)

└── Private Subnet (no IGW)

└── EC2 instance → Internet via NAT Gateway

➡ Private instances can access Internet for updates,  
but the Internet cannot access them directly.

💼 In My Project:

✅ We used NAT Gateway for production workloads  
because it’s fully managed, scalable, and highly available.

We placed one NAT Gateway per AZ, associated it with private route tables,  
and routed internet-bound traffic (0.0.0.0/0) through it.

🧠 In Short

| Feature | NAT Instance | NAT Gateway |
| --- | --- | --- |
| Managed By | You | AWS |
| Scalable | Manual | Auto |
| Cost | Low | Moderate |
| Availability | Manual setup | Built-in |
| Best For | Small/test | Production workloads |

**57 \* loadBalencer**

A Load Balancer is a service that automatically distributes incoming traffic (like HTTP requests) across multiple servers (EC2 instances, containers, etc.) to ensure:

✅ High availability – if one instance fails, traffic goes to others.  
✅ Scalability – handle more users by adding more backend instances.  
✅ Fault tolerance – keeps your app running smoothly.  
✅ Better performance – balances workloads evenly.

🏗️ Types of Load Balancers in AWS (Elastic Load Balancing – ELB)

| Type | Description | Use Case |
| --- | --- | --- |
| 1️⃣ Application Load Balancer (ALB) | Works at Layer 7 (HTTP/HTTPS) — routes based on URL path, headers, or host. | Web apps, microservices, APIs |
| 2️⃣ Network Load Balancer (NLB) | Works at Layer 4 (TCP/UDP) — ultra-fast, handles millions of requests per second. | Gaming, IoT, real-time apps |
| 3️⃣ Gateway Load Balancer (GWLB) | Works at Layer 3 (Network layer) — used for routing traffic to security appliances. | Firewalls, inspection systems |
| 4️⃣ Classic Load Balancer (CLB) *(Old)* | Works at Layer 4 and 7, limited features. | Legacy apps only |

🧩 How It Works

Internet Users

↓

Load Balancer

/ | \

EC2-1 EC2-2 EC2-3

* The Load Balancer listens on ports (like 80 or 443).
* It forwards traffic to healthy targets (in a target group).
* Uses health checks to remove unhealthy instances automatically.

⚙️ Core Components

| Component | Description |
| --- | --- |
| Listeners | Check for connection requests on configured ports. |
| Target Groups | Contain EC2 instances, IPs, or ECS containers receiving traffic. |
| Health Checks | Continuously test targets; removes unhealthy ones. |
| Rules | Define routing logic (for ALB — e.g., path-based or host-based). |

🧠 Example Scenario

Your app has 3 EC2 instances running behind an Application Load Balancer (ALB):

* If EC2-1 fails, LB sends traffic to EC2-2 and EC2-3.
* When EC2-1 becomes healthy again, it automatically rejoins.

🔐 Integration with Other AWS Services

* EC2 Auto Scaling → adds/removes instances based on load.
* Route 53 → DNS routing to the load balancer.
* CloudWatch → monitors LB metrics (latency, requests, etc.).
* ACM → SSL/TLS certificates for HTTPS traffic.

💻 Example (CLI – Create an ALB)

aws elbv2 create-load-balancer \

--name my-load-balancer \

--subnets subnet-1 subnet-2 \

--security-groups sg-12345678

🧩 Summary Table

| Feature | ALB | NLB | GWLB |
| --- | --- | --- | --- |
| Layer | 7 | 4 | 3 |
| Protocol | HTTP/HTTPS | TCP/UDP | IP |
| Routing | Path/Host-based | Port-based | Appliance-based |
| Use Case | Web Apps | High Performance | Security Tools |

**58 \* aws VPC components**

A VPC (Virtual Private Cloud) is your private network inside AWS where you can securely launch and manage your resources (like EC2, RDS, etc.).  
It’s similar to a traditional on-premise data center — but fully managed and scalable in the cloud.

🧩 Main Components of a VPC

Let’s go step-by-step 👇

| Component | Purpose / Function | Example / Notes |
| --- | --- | --- |
| 1️⃣ VPC (Virtual Private Cloud) | The main virtual network that you create inside AWS. | Example: 10.0.0.0/16 |
| 2️⃣ Subnets | Divide your VPC into smaller networks. You can create Public and Private subnets. | Example: 10.0.1.0/24 (Public), 10.0.2.0/24 (Private) |
| 3️⃣ Internet Gateway (IGW) | Allows communication between VPC and the Internet. | Attach it to your VPC so public subnets can reach the internet. |
| 4️⃣ NAT Gateway / NAT Instance | Allows private subnets to access the internet (for updates/downloads) without being accessible from outside. | Used when you don’t want direct internet access. |
| 5️⃣ Route Tables | Define routing rules for each subnet (where traffic should go). | Public subnet → IGW; Private subnet → NAT Gateway |
| 6️⃣ Security Groups | Virtual firewalls for EC2 instances (control inbound/outbound traffic at instance level). | Example: Allow SSH (22), HTTP (80) |
| 7️⃣ Network ACLs (NACLs) | Subnet-level firewall, controls inbound and outbound traffic at subnet level. | Stateless: must allow both inbound and outbound explicitly |
| 8️⃣ DHCP Option Set | Provides DNS and IP configuration to instances in VPC. | Default points to AWS-provided DNS |
| 9️⃣ VPC Peering | Connects two VPCs so they can communicate using private IPs. | Used between accounts or regions (if needed). |
| 🔟 Endpoints (Gateway/Interface) | Private connections to AWS services (like S3, DynamoDB) without using the internet. | Example: S3 Gateway Endpoint |
| 11️⃣ Elastic IPs (EIP) | Static public IPs for instances in public subnets. | Useful for fixed public IP needs. |
| 12️⃣ Elastic Load Balancer (ELB) | Distributes incoming traffic across multiple EC2 instances. | Typically placed in public subnet. |

🧠 Simple Architecture Example

VPC (10.0.0.0/16)

│

├── Public Subnet (10.0.1.0/24)

│ ├── Internet Gateway (IGW)

│ ├── EC2 (Web/App)

│ └── Route → IGW

│

├── Private Subnet (10.0.2.0/24)

│ ├── NAT Gateway

│ ├── EC2 (DB/Backend)

│ └── Route → NAT Gateway

│

└── Security Groups + NACLs for access control

🔁 In Interview (Concise Answer)

“A VPC consists of components like subnets, route tables, internet gateways, NAT gateways, security groups, and NACLs.  
Public subnets use the Internet Gateway to connect to the internet, and private subnets use a NAT Gateway for outbound traffic.  
We also use VPC endpoints to connect privately to AWS services and security groups/NACLs to control traffic flow.”

✅ Quick Summary Table

| Layer | Component | Purpose |
| --- | --- | --- |
| Network | VPC, Subnets | IP addressing & segmentation |
| Connectivity | IGW, NAT Gateway, Route Table, Endpoints | Control inbound/outbound traffic |
| Security | Security Groups, NACLs | Traffic filtering |
| Extension | Peering, VPN, Transit Gateway | Cross-VPC or on-prem connectivity |

**59 \* what are the DB you are going to use**

🔹 Sample Answer (Short & Direct)

“In our project, we used both RDS (Relational Database Service) and DynamoDB, depending on the type of data.  
For transactional and structured data, we used RDS MySQL, and for high-speed, unstructured data access, we used DynamoDB.”

🔹 Detailed Explanation (Professional Answer)

🧩 1. Relational Database (RDS)

* Managed service for SQL databases like MySQL, PostgreSQL, MariaDB, Oracle, SQL Server, Aurora.
* Supports multi-AZ replication, automated backups, snapshots, monitoring.
* Best for:
  + Applications with structured data (tables, relations)
  + Transactional consistency (ACID) needed
  + OLTP workloads

Example: “We used Amazon RDS MySQL for storing user information, orders, and transactions because it provides high availability and automatic failover.”

⚡ 2. NoSQL Database (DynamoDB)

* Fully managed key-value and document database.
* Scales automatically to handle millions of requests per second.
* Best for:
  + Real-time apps (chat, gaming, IoT)
  + Fast lookup and flexible schema
  + No joins or complex queries

Example: “We used DynamoDB for session management and caching user preferences.”

🧠 3. Amazon Aurora

* AWS’s high-performance managed database compatible with MySQL and PostgreSQL.
* Up to 5x faster than standard MySQL.
* Auto-scaling storage up to 128 TB.

Example: “For critical production workloads, we prefer Aurora because it combines enterprise performance with full RDS automation.”

☁️ 4. Other Databases (Based on Use Case)

| Service | Type | Use Case |
| --- | --- | --- |
| ElastiCache (Redis/Memcached) | In-memory cache | Caching frequent queries, session storage |
| Redshift | Data warehouse | Analytics and reporting |
| Neptune | Graph database | Social networks, recommendation systems |
| DocumentDB | MongoDB-compatible NoSQL | JSON document storage |

🔹 Example Real-World Answer

“In our AWS project, we used:

* RDS MySQL for structured transactional data,
* DynamoDB for user activity logs and sessions, and
* ElastiCache Redis for caching frequently accessed data.

This mix gave us high performance, scalability, and cost-efficiency.”

✅ Summary (For Interview Quick Recall)

| Database | Type | Used For |
| --- | --- | --- |
| RDS (MySQL/PostgreSQL) | Relational | Structured data, transactions |
| Aurora | Relational | High performance and scalability |
| DynamoDB | NoSQL | Fast key-value access |
| ElastiCache | In-memory | Caching, session management |
| Redshift | Data warehouse | Analytics, reporting |

**61 \* difference between network lb and application lb**

⚙️ Overview

| Type | Layer | Main Purpose |
| --- | --- | --- |
| 🌐 Application Load Balancer (ALB) | Layer 7 (Application Layer) | Used for HTTP / HTTPS traffic — intelligent routing based on content. |
| ⚡ Network Load Balancer (NLB) | Layer 4 (Transport Layer) | Used for TCP / UDP traffic — ultra-high performance and low latency. |

🧩 Detailed Comparison

| Feature | Application Load Balancer (ALB) | Network Load Balancer (NLB) |
| --- | --- | --- |
| OSI Layer | Layer 7 (Application Layer) | Layer 4 (Transport Layer) |
| Protocol Support | HTTP, HTTPS, WebSocket | TCP, UDP, TLS |
| Routing Type | Content-based (Path, Host, Header, Query) | IP-based (Destination Port / IP) |
| Use Case | Web apps, microservices, REST APIs | Gaming, IoT, Database proxy, high-performance apps |
| Target Type | Instance, IP, Lambda | Instance, IP, ALB |
| Performance | Good | Very high (handles millions of requests per second) |
| Static IP | ❌ Not supported | ✅ Supported (one per AZ or EIP) |
| Sticky Sessions | ✅ Supported (via cookies) | ✅ Supported (via source IP) |
| SSL Termination | ✅ Yes (can decrypt HTTPS) | ✅ Yes (TLS listener), or pass-through |
| Health Checks | At HTTP/HTTPS level (path-based) | At TCP level (port check) |
| Logging | Access Logs + CloudWatch metrics | Flow Logs + CloudWatch metrics |
| Pricing | Lower cost | Slightly higher cost |
| Integration with WAF (Web Application Firewall) | ✅ Supported | ❌ Not supported |

💼 Use Case Examples

| Scenario | Best Choice |
| --- | --- |
| Hosting a website or REST API | ✅ ALB |
| Routing traffic based on URL (e.g., /api, /images) | ✅ ALB |
| Need SSL termination and path-based routing | ✅ ALB |
| Handling millions of TCP connections (e.g., chat, gaming) | ✅ NLB |
| Need static IPs or Elastic IPs for clients | ✅ NLB |
| Handling non-HTTP protocols (SMTP, MQTT, etc.) | ✅ NLB |

🧠 Architecture Example

Application Load Balancer (ALB):

Clients

↓

ALB (HTTP/HTTPS)

↓

Target Groups (EC2, Lambda, ECS)

Routes requests based on:

* /api → App servers
* /images → Image servers

Network Load Balancer (NLB):

Clients

↓

NLB (TCP/UDP/TLS)

↓

Target Groups (EC2 or IPs)

Routes purely by port/IP, no inspection of HTTP headers.

💼 In My Project:

✅ We used Application Load Balancer (ALB)  
because we were hosting microservices-based web APIs on EC2 + ECS,  
needed path-based routing and SSL termination.

Example:

/api/users → User Service

/api/orders → Order Service

/api/payments → Payment Service

All behind one ALB, which handled HTTPS termination and forwarded traffic to respective target groups.

🧩 Quick Summary

| Feature | ALB | NLB |
| --- | --- | --- |
| Layer | 7 (HTTP/HTTPS) | 4 (TCP/UDP) |
| Routing | Content-based | IP/Port-based |
| Static IP | No | Yes |
| SSL Termination | Yes | Yes |
| Health Checks | HTTP-based | TCP-based |
| AWS WAF Support | Yes | No |
| Use Case | Web, APIs | High-performance, gaming, database |

**62 \* about Route 53**

🌍 What is Amazon Route 53?

Amazon Route 53 is a highly available and scalable Domain Name System (DNS) web service by AWS.  
It’s used to route users’ requests to applications running in AWS or on-premises — just like a phone book that translates a domain name (www.example.com) into an IP address.

⚙️ Main Functions of Route 53

| Function | Description |
| --- | --- |
| 1️⃣ Domain Registration | You can buy and manage domain names directly (like myapp.com). |
| 2️⃣ DNS Routing | Converts domain names → IP addresses (e.g., myapp.com → 192.0.2.1). |
| 3️⃣ Health Checking | Continuously monitors health of your endpoints and routes traffic only to healthy ones. |
| 4️⃣ Traffic Management (Routing Policies) | Intelligent routing worldwide — latency-based, failover, geolocation, etc. |

🧩 Core Components

| Component | Description |
| --- | --- |
| Hosted Zone | A container for DNS records of your domain (like a folder for example.com). |
| Record Set | Actual DNS records — e.g., A, AAAA, CNAME, MX, TXT, Alias, etc. |
| Health Check | Monitors the health of web servers, load balancers, or other endpoints. |

📘 Common DNS Record Types in Route 53

| Record Type | Purpose | Example |
| --- | --- | --- |
| A Record | Maps domain → IPv4 address | example.com → 192.0.2.1 |
| AAAA Record | Maps domain → IPv6 address | example.com → 2001:db8::1 |
| CNAME | Maps one domain to another | www.example.com → example.com |
| Alias Record | AWS-specific; points domain to AWS resource | example.com → CloudFront / ALB |
| MX | Mail servers | example.com → mail.example.com |
| TXT | Verification, SPF, etc. | "v=spf1 include:amazonses.com" |

🚦 Routing Policies in Route 53

| Policy | Description | Use Case |
| --- | --- | --- |
| Simple Routing | One record → one resource. | Single web server. |
| Weighted Routing | Distributes traffic by percentage. | Canary deployments. |
| Latency-based Routing | Routes users to the region with lowest latency. | Global web apps. |
| Failover Routing | Sends traffic to secondary site when primary fails. | Disaster recovery. |
| Geolocation Routing | Routes based on user’s physical location. | Country-specific content. |
| Geoproximity Routing | Routes based on distance to resources. | Region-aware routing. |
| Multivalue Answer Routing | Returns multiple IPs for load balancing and health checks. | Basic load balancing without ALB. |

🧠 Example Flow

User → Route 53 → Load Balancer → EC2 Instances → Application

* User types www.myapp.com
* Route 53 looks up the domain
* Returns the IP of the healthy AWS Load Balancer
* Load Balancer forwards traffic to backend EC2 instances

🔐 Integration with Other AWS Services

* ELB / ALB → Use Alias record for DNS mapping
* CloudFront → Distribute content globally
* S3 Static Website → Use Route 53 to map domain
* Health Checks → Monitor application endpoints

💻 Example (AWS CLI)

Create a hosted zone:

aws route53 create-hosted-zone \

--name example.com \

--caller-reference "myapp-2025"

Create an A record:

aws route53 change-resource-record-sets \

--hosted-zone-id Z1234567890 \

--change-batch '{

"Changes": [{

"Action": "CREATE",

"ResourceRecordSet": {

"Name": "www.example.com",

"Type": "A",

"AliasTarget": {

"HostedZoneId": "Z2FDTNDATAQYW2",

"DNSName": "dualstack.my-alb-12345.elb.amazonaws.com",

"EvaluateTargetHealth": false

}

}

}]

}'

🧾 Quick Summary

| Feature | Description |
| --- | --- |
| Service Type | Global DNS service |
| Managed by | AWS |
| High Availability | Yes (globally distributed) |
| Use Case | Domain registration, DNS routing, health checks |
| Special Feature | Alias records for AWS integration |

**63 \* about autoscaling**

🧩 1️⃣ What is Auto Scaling?

Auto Scaling in AWS is a feature that automatically adjusts the number of EC2 instances (or other resources) based on the demand or load.

It helps you:

* Maintain application availability
* Scale out (add instances) when demand increases
* Scale in (remove instances) when demand decreases
* Save cost and ensure performance

⚙️ 2️⃣ Key Components of Auto Scaling

| Component | Purpose |
| --- | --- |
| Launch Template / Launch Configuration | Defines how new instances will be launched (AMI, instance type, security groups, key pair, etc.) |
| Auto Scaling Group (ASG) | Logical group of EC2 instances — manages scaling and health checks |
| Scaling Policies | Rules that define when to add/remove instances (based on metrics like CPU utilization) |
| CloudWatch Alarms | Monitors performance metrics and triggers scaling actions |
| Load Balancer (optional) | Distributes traffic evenly among instances; integrates with ASG for health checks |

🧩 3️⃣ How Auto Scaling Works (Simple Flow)

CloudWatch monitors → detects high CPU usage

↓

Triggers Auto Scaling Policy

↓

ASG launches new EC2 instance(s)

↓

Traffic distributed via Load Balancer (ALB)

↓

When load decreases → Auto Scaling terminates extra instances

💡 4️⃣ Types of Scaling Policies

| Type | Description |
| --- | --- |
| Dynamic Scaling | Adds/removes instances automatically based on CloudWatch metrics (e.g., CPU > 80%) |
| Scheduled Scaling | Scale in/out based on a fixed schedule (e.g., weekdays at 9 AM) |
| Predictive Scaling | Uses machine learning to predict future traffic patterns |
| Manual Scaling | You manually set the desired number of instances |

🧠 5️⃣ Example: Dynamic Scaling Policy

* Scale out: When CPU > 70% for 5 minutes → add 1 instance
* Scale in: When CPU < 30% for 10 minutes → remove 1 instance

🧩 6️⃣ Example Use Case

Suppose your web app experiences high traffic during business hours and low traffic at night.  
Auto Scaling automatically increases EC2 instances during peak hours and reduces them later — saving cost and maintaining performance.

🧩 7️⃣ Benefits of Auto Scaling

✅ High Availability — replaces failed instances automatically  
✅ Cost Optimization — runs only what’s needed  
✅ Fault Tolerance — unhealthy instances replaced automatically  
✅ Improved Performance — scales with demand  
✅ Integration — works with ALB, CloudWatch, SNS, etc.

⚙️ 8️⃣ Typical Architecture

Users → ALB → Auto Scaling Group (EC2s across multiple AZs)

↳ CloudWatch monitors metrics

↳ Auto Scaling adds/removes instances

💬 9️⃣ Short Interview Answer

“Auto Scaling is used to automatically adjust the number of EC2 instances based on demand.  
We create a Launch Template, define an Auto Scaling Group, and attach CloudWatch alarms with scaling policies.  
It ensures high availability and cost efficiency by scaling up during high traffic and scaling down during low usage.”

🧩 10️⃣ Pro Tip

When you mention Auto Scaling in interviews, always say:

“We configured our Auto Scaling group to work with an Application Load Balancer across multiple Availability Zones, ensuring both fault tolerance and scalability.”

**64 \* how to use terraform in AWS**

🔹 1. What is Terraform?

Terraform is an Infrastructure as Code (IaC) tool by HashiCorp that allows you to automate AWS resource creation, modification, and management using simple configuration files written in HCL (HashiCorp Configuration Language).

It helps you:

* Automate AWS setup (VPC, EC2, S3, IAM, etc.)
* Maintain version control of infrastructure
* Easily replicate or destroy environments
* Ensure consistency and repeatability

🔹 2. Basic Workflow of Terraform in AWS

| Step | Command | Description |
| --- | --- | --- |
| 1. Write Configuration | — | Define AWS resources in .tf files (e.g., VPC, EC2, S3) |
| 2. Initialize | terraform init | Downloads AWS provider plugins |
| 3. Validate | terraform validate | Checks syntax and correctness |
| 4. Plan | terraform plan | Shows what Terraform will create/modify/destroy |
| 5. Apply | terraform apply | Executes the plan and provisions resources |
| 6. Destroy | terraform destroy | Removes all created resources |

🔹 3. Example: Terraform Configuration for AWS EC2

# Provider Section

provider "aws" {

region = "us-east-1"

}

# Create EC2 Instance

resource "aws\_instance" "my\_ec2" {

ami = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI ID

instance\_type = "t2.micro"

tags = {

Name = "Terraform-EC2"

}

}

Commands to run:

terraform init

terraform plan

terraform apply

✅ This will create an EC2 instance in your AWS account.

🔹 4. AWS Credentials Setup

Terraform needs AWS credentials to connect. You can configure them by:

aws configure

or by setting environment variables:

export AWS\_ACCESS\_KEY\_ID="your\_access\_key"

export AWS\_SECRET\_ACCESS\_KEY="your\_secret\_key"

🔹 5. Terraform State File

* Terraform keeps track of all created resources in a file called terraform.tfstate.
* It helps Terraform know what exists in AWS vs what’s defined in code.
* You can store it remotely in S3 for team collaboration:
* backend "s3" {
* bucket = "terraform-state-bucket"
* key = "prod/terraform.tfstate"
* region = "us-east-1"
* }

🔹 6. Common Use Cases in AWS

You can use Terraform to create:

* VPCs, Subnets, Route Tables
* EC2 instances
* S3 Buckets
* Security Groups, IAM Roles
* Load Balancers
* RDS Databases
* Auto Scaling groups

🔹 7. Advantages of Using Terraform in AWS

✅ Infrastructure as Code — Version-controlled, reusable  
✅ Multi-Cloud — Not limited to AWS  
✅ Automation — No manual console steps  
✅ Plan Before Apply — Safe changes  
✅ Consistency — Same setup across environments (dev, test, prod)

🔹 8. Sample Interview Answer

“We use Terraform in AWS to automate infrastructure provisioning.  
For example, we define our VPC, subnets, EC2 instances, and security groups in .tf files.  
Then we initialize Terraform, review the plan, and apply changes to create resources automatically.

We also store Terraform state files in S3 with DynamoDB for locking, ensuring team consistency.  
This helps us manage multiple environments efficiently and avoid manual setup errors.”

**65 \* how will you restric inbound traffic when outbound traffic in open**

🧩 Understanding the Scenario

In AWS, inbound and outbound traffic are controlled separately through:

* Security Groups (SGs) – instance-level firewall
* Network ACLs (NACLs) – subnet-level firewall

You can allow all outbound traffic (for updates, API calls, etc.)  
but restrict or deny inbound traffic (to prevent external access).

⚙️ How to Restrict Inbound Traffic

Option 1 — Use Security Group Rules

Security groups are *stateful*, meaning:

* If you allow outbound traffic, return traffic for that session is automatically allowed.
* You can block new incoming connections by not adding inbound rules.

Steps:

1. Go to EC2 → Security Groups → Your SG
2. Under Inbound Rules, remove all rules or allow only specific trusted IPs.  
   Example:
3. Inbound:
4. Type: SSH | Protocol: TCP | Port: 22 | Source: 10.0.0.0/24 ✅ (your private network)
5. Keep Outbound Rules open:
6. Outbound:
7. Type: All traffic | Destination: 0.0.0.0/0 ✅

✅ Result:  
Your instance can send data to the internet (outbound open),  
but no one can initiate a new connection *into* your instance (inbound restricted).

Option 2 — Use NACL (Network ACL)

NACLs are *stateless*, so inbound and outbound must be defined separately.

To restrict inbound:

1. In VPC → NACLs, edit the subnet’s NACL.
2. Set:
3. Inbound rules:
4. Rule 100 - Deny All (0.0.0.0/0)
5. Outbound rules:
6. Rule 100 - Allow All (0.0.0.0/0)

✅ Result:  
Outbound communication is allowed, but no new inbound connection can enter that subnet.

Option 3 — Use a NAT Gateway (for private subnets)

If your instances are in private subnets,  
you can route outbound traffic through a NAT Gateway (in a public subnet).

* Instances initiate outbound requests via NAT Gateway.
* NAT Gateway translates and sends responses back.
* No inbound connection is possible directly to the instance.

✅ This design is very common for secure architectures.

💡 Real-World Example

In my project:

* EC2s in private subnets had no inbound rules.
* Outbound traffic allowed to 0.0.0.0/0 via NAT Gateway.
* This let the instances download updates or connect to external APIs,  
  but blocked all incoming connections from the internet.

🧠 Summary Table

| Control Method | Restrict Inbound | Allow Outbound | Stateful? | Best Use Case |
| --- | --- | --- | --- | --- |
| Security Group | ✅ (by removing inbound rules) | ✅ (allow all) | ✅ Yes | Instance-level control |
| NACL | ✅ (deny inbound) | ✅ (allow outbound) | ❌ No | Subnet-level control |
| NAT Gateway | ✅ (blocks inbound automatically) | ✅ (enables outbound) | N/A | Private subnet design |

**66 \* what is vpc and vpc perring**

Amazon VPC (Virtual Private Cloud) is your own private network inside AWS.

It allows you to launch AWS resources (like EC2, RDS, etc.) inside a logically isolated network that you control completely — including IP addresses, subnets, routing, and security.

🧩 Key Components of a VPC

| Component | Description |
| --- | --- |
| CIDR Block | Defines IP address range (e.g., 10.0.0.0/16). |
| Subnets | Divide your VPC into smaller networks (e.g., Public & Private). |
| Route Table | Defines how traffic is routed between subnets or to the internet. |
| Internet Gateway (IGW) | Connects VPC to the Internet (for public subnets). |
| NAT Gateway / NAT Instance | Allows private subnets to access the Internet securely. |
| Security Groups | Instance-level firewall – allows or blocks traffic to EC2. |
| Network ACLs (NACLs) | Subnet-level firewall – additional traffic control. |
| VPC Peering / Transit Gateway | For connecting VPCs together. |

🏗️ Example VPC Structure

VPC (10.0.0.0/16)

├── Public Subnet (10.0.1.0/24)

│ ├── EC2 (Web Server)

│ └── Internet Gateway

└── Private Subnet (10.0.2.0/24)

├── EC2 (Database)

└── NAT Gateway

* Public subnet has Internet access.
* Private subnet can only access Internet via NAT.

🔗 What is VPC Peering?

VPC Peering is a network connection between two VPCs that allows them to communicate privately as if they were part of the same network.

🧠 VPC Peering Key Points

| Feature | Description |
| --- | --- |
| Purpose | Connect two VPCs privately without using Internet. |
| Traffic Path | Uses private AWS backbone network. |
| Connection Type | One-to-one (not transitive). |
| Direction | Peering is bidirectional once accepted. |
| Region | Works across same or different regions (inter-region). |
| Routing | Must update Route Tables in both VPCs to allow traffic. |

🛠️ Example

VPC A: 10.0.0.0/16  
VPC B: 192.168.0.0/16

You create a peering connection so that:

* EC2 in VPC A can communicate with EC2 in VPC B via private IPs.
* No need for Internet Gateway or NAT Gateway.

💻 CLI Example: Create a VPC Peering Connection

aws ec2 create-vpc-peering-connection \

--vpc-id vpc-11111111 \

--peer-vpc-id vpc-22222222 \

--region ap-south-1

Then accept it:

aws ec2 accept-vpc-peering-connection \

--vpc-peering-connection-id pcx-1234567890abcdef

Finally, update route tables in both VPCs to route traffic through the peering connection.

⚠️ Limitations

* Peering is not transitive (VPC-A ↔ VPC-B ↔ VPC-C ❌).
* CIDR ranges cannot overlap.
* Cannot use security groups references across peered VPCs (unless same region).

✅ Use Cases

* Communication between Dev & Prod VPCs.
* Centralized logging or monitoring VPC.
* Multi-account setups under AWS Organizations.

🧾 Quick Summary

| Feature | VPC | VPC Peering |
| --- | --- | --- |
| Definition | Private network in AWS. | Connection between two VPCs. |
| Purpose | Isolate and secure AWS resources. | Enable private communication between VPCs. |
| Internet Required? | No (unless using IGW). | No (uses AWS internal network). |
| Transitive? | N/A | ❌ Not transitive. |
| Overlapping CIDR? | Allowed. | ❌ Not allowed. |

**67 \* How will an instance present in Oregon can access a S3 bucket present in Sydney without any manual intervention?**

🧩 1️⃣ Scenario

* EC2 Region: Oregon (us-west-2)
* S3 Bucket Region: Sydney (ap-southeast-2)
* Goal: EC2 in Oregon should automatically access (read/write) data in S3 Sydney without manual login or key copying.

⚙️ 2️⃣ Best AWS Solution — IAM Role + S3 Bucket Policy

The best practice (and automation-friendly) approach is:

✅ Step-by-step

Step 1: Create an IAM Role for EC2

* Create a role in Oregon (EC2’s region).
* Attach a policy like:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": ["s3:GetObject", "s3:PutObject"],

"Resource": "arn:aws:s3:::my-sydney-bucket/\*"

}

]

}

Step 2: Attach the Role to EC2 Instance

* When EC2 runs, it automatically gets temporary credentials using the instance metadata service (IMDS).
* No need for access keys or manual setup.

Step 3: Allow Access from That Role in the S3 Bucket Policy

In the Sydney bucket’s policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::111122223333:role/Oregon-EC2-Role"

},

"Action": ["s3:GetObject", "s3:PutObject"],

"Resource": "arn:aws:s3:::my-sydney-bucket/\*"

}

]

}

✅ Now the EC2 instance in Oregon can automatically read/write to the S3 bucket in Sydney, without any manual credentials.

🌏 3️⃣ If They’re in Different AWS Accounts

You can:

* Use a cross-account IAM role trust  
  (Allow Oregon account’s EC2 role to access Sydney account’s S3 bucket)
* Enable S3 cross-region replication (CRR) if you want automatic copy between buckets.

⚡ 4️⃣ “Without manual intervention” Explanation

* IAM roles auto-manage credentials via AWS STS (Security Token Service).
* Tokens are rotated automatically every few hours.
* No need to manually copy access keys, scripts, or cron jobs.

🔐 5️⃣ Optional: Use S3 Cross-Region Replication (CRR)

If the requirement means:  
“Data from Oregon S3 should automatically be available in Sydney S3”

Then:

* Enable CRR (Cross-Region Replication) between buckets.
* Configure replication rules in the source bucket (Oregon) pointing to the destination bucket (Sydney).
* Requires versioning enabled on both sides.

✅ Once enabled, replication happens automatically, continuously, and securely.

💬 6️⃣ Interview Short Answer

“We can attach an IAM Role to the EC2 instance in Oregon, granting it access to the S3 bucket in Sydney through a proper bucket policy.  
The IAM role automatically manages temporary credentials using AWS STS, so no manual intervention is needed.  
Alternatively, if data needs to be synced automatically, we can use Cross-Region Replication (CRR) between S3 buckets.”

🧠 7️⃣ Key Takeaways

| Requirement | Solution | Manual Work? |
| --- | --- | --- |
| EC2 → S3 cross-region access | IAM Role + Bucket Policy | ❌ |
| Data sync between S3 buckets | Cross Region Replication | ❌ |
| Private, secure access | Use VPC Endpoint or PrivateLink | ❌ |

**68 \* (only the ec2 shd b able to access the S3 without any user involved in it,)**

✅ Short, Direct Answer

Create an IAM Role with permissions to access the S3 bucket, attach it to the EC2 instance, and ensure the S3 bucket policy allows access only from that IAM role.

This way, no IAM user or access key is needed — only the EC2 instance can access S3.

🔹 Step-by-Step Solution

1. Create an IAM Role

* Go to IAM → Roles → Create role
* Choose AWS service → EC2
* Attach a policy like:
  + AmazonS3ReadOnlyAccess (for read access), or
  + a custom policy for specific bucket access

Example custom policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:GetObject",

"s3:PutObject"

],

"Resource": "arn:aws:s3:::my-bucket-name/\*"

}

]

}

✅ This gives the EC2 permission to access only your S3 bucket.

2. Attach the Role to the EC2 Instance

* While launching EC2, under IAM Role, select the role you just created.
* Or attach it later:
* aws ec2 associate-iam-instance-profile \
* --instance-id i-xxxxxxxxx \
* --iam-instance-profile Name=MyS3AccessRole

Now the EC2 instance has temporary credentials to access S3 securely.

3. Modify the S3 Bucket Policy (Optional but More Secure)

To ensure only that EC2 role can access the bucket, edit your S3 bucket policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::123456789012:role/MyS3AccessRole"

},

"Action": "s3:\*",

"Resource": [

"arn:aws:s3:::my-bucket-name",

"arn:aws:s3:::my-bucket-name/\*"

]

}

]

}

✅ This ensures only that EC2 instance’s IAM role can access the bucket — no users, no keys.

4. Access S3 from EC2 (No Keys Needed)

Once the EC2 has the IAM role:

aws s3 ls s3://my-bucket-name

aws s3 cp /path/to/file s3://my-bucket-name/

* It works without configuring aws configure.
* The temporary credentials are automatically provided by the EC2 instance metadata service (IMDSv2).

🔒 Security Benefits

✅ No hard-coded credentials  
✅ IAM Role rotates temporary credentials automatically  
✅ Bucket access restricted to that EC2 only  
✅ Follows AWS best practices for least privilege

🧠 Interview Answer (Ideal Response)

“To allow only EC2 to access S3 without user credentials, I’ll create an IAM Role with S3 access policy and attach it to the EC2 instance.  
The EC2 uses instance profile credentials to communicate with S3 through the AWS SDK or CLI, so no access keys are required.  
I’ll also restrict the S3 bucket policy to allow access only from that IAM role, ensuring complete isolation.”

**69 \* One ALB with 2 instances A and B and ALB is mapped with 2 website site1 and site2,**

🧩 Scenario

* You have one ALB (Application Load Balancer).
* Two EC2 instances: Instance A and Instance B.
* Two websites: site1.com and site2.com.
* Both should be served through the same ALB.

🎯 Goal

Route traffic to the correct website based on the domain name (site1 or site2) or the path (like /app1, /app2).

⚙️ Solution: Use ALB with Host-Based Routing

✅ Step-by-Step Setup

1. Create Target Groups
   * Target Group 1 → Instance A (for site1.com)
   * Target Group 2 → Instance B (for site2.com)
2. Register Instances
   * Register Instance A under Target Group 1
   * Register Instance B under Target Group 2
3. Set Up ALB Listener Rules
   * Go to your ALB → Listeners → HTTP (80) or HTTPS (443) → “View/Edit rules”
   * Add rules like this:

| Condition | Action |
| --- | --- |
| Host is site1.com | Forward to Target Group 1 |
| Host is site2.com | Forward to Target Group 2 |

1. This is called Host-Based Routing.
2. DNS Configuration
   * In Route 53 (or your DNS provider):
     + Create A/ALIAS record for site1.com → points to ALB DNS name
     + Create A/ALIAS record for site2.com → points to ALB DNS name
3. (Optional) Add HTTPS using SSL Certificates
   * Use AWS Certificate Manager (ACM) to issue a single SSL cert for both domains (site1.com, site2.com).
   * Attach that to the ALB listener (port 443).

🧠 How It Works

Internet

↓

Application Load Balancer

↓

┌─────────────────────────────┐

│ Host-based routing rules: │

│ site1.com → Target Group 1 │

│ site2.com → Target Group 2 │

└─────────────────────────────┘

↓ ↓

Instance A Instance B

(serves site1) (serves site2)

⚡ Alternative: Path-Based Routing

If both websites share the same domain but different URLs (example.com/site1 and example.com/site2):

* Create rules like:
  + /site1/\* → Target Group 1 (Instance A)
  + /site2/\* → Target Group 2 (Instance B)

✅ In Short

| Requirement | Routing Type | Rule Example |
| --- | --- | --- |
| Different domain names (site1.com, site2.com) | Host-based | Host is site1.com |
| Same domain but different URLs | Path-based | /site1/\* |

💼 Example from Real Projects

In my project, we hosted multiple client applications on the same ALB.  
Each app had a different domain, and ALB handled routing using host-based rules, saving cost and simplifying management.

**70 \* Can we configure requests which comes to site1 mapped to instance A n site2 to instance B.**

Excellent question 👍 — yes, you can configure requests so that:

➡️ site1.example.com → Instance A  
➡️ site2.example.com → Instance B

This is a very common real-world scenario, and you can achieve it easily using AWS Route 53 + Load Balancer (or even without one).

Let’s see how you can do it step by step 👇

🧩 Option 1: Using Route 53 DNS Records

🎯 Goal:

You have two subdomains:

* site1.example.com
* site2.example.com

And you want:

* site1 traffic → Instance A (EC2-1)
* site2 traffic → Instance B (EC2-2)

⚙️ Steps:

1. Go to Route 53 → Hosted Zone → example.com
2. Create two A records:

| Record Name | Type | Value |
| --- | --- | --- |
| site1.example.com | A | Public IP of Instance A |
| site2.example.com | A | Public IP of Instance B |

✅ Now requests to each subdomain go directly to their respective instances.

🔒 Make sure security groups allow inbound traffic (e.g., port 80/443).

🧩 Option 2: Using Application Load Balancer (Recommended)

If you want a single entry point (like one Load Balancer) that routes traffic internally to different instances, use Host-based Routing on an Application Load Balancer (ALB).

🧱 Architecture

Client

↓

Application Load Balancer

├── Rule 1: If host = site1.example.com → Target Group 1 (Instance A)

└── Rule 2: If host = site2.example.com → Target Group 2 (Instance B)

⚙️ Steps

1. Create an ALB (Application Load Balancer).
2. Create 2 Target Groups:
   * Target Group 1 → Instance A
   * Target Group 2 → Instance B
3. Go to ALB → Listeners → Rules:
   * Add Host-based rule:
     + IF Host is site1.example.com → Forward to Target Group 1
     + IF Host is site2.example.com → Forward to Target Group 2
4. In Route 53, create DNS records:
   * site1.example.com → ALB DNS name
   * site2.example.com → ALB DNS name

✅ ALB will intelligently route traffic to the correct instance based on the hostname in the request.

💡 Option 3: Using NGINX Reverse Proxy (manual approach)

If you don’t want to use ALB:

* Deploy an NGINX instance in front of your app servers.
* Configure server blocks:

server {

server\_name site1.example.com;

location / {

proxy\_pass http://INSTANCE\_A\_PRIVATE\_IP;

}

}

server {

server\_name site2.example.com;

location / {

proxy\_pass http://INSTANCE\_B\_PRIVATE\_IP;

}

}

✅ Summary

| Method | Description | Recommended For |
| --- | --- | --- |
| Route 53 A Records | Simple DNS mapping to different EC2 IPs | Small/static setup |
| ALB Host-based Routing | Intelligent routing based on domain | Scalable production |
| NGINX Reverse Proxy | Manual control using custom configs | Custom or hybrid setups |

**71 \* Can we achieve this behaviour using ALB?**

⚙️ 1️⃣ First — What Behavior Are We Talking About?

If the previous question refers to:

“EC2 in Oregon accessing S3 bucket in Sydney without manual intervention”

Then → No, ALB cannot be used for this.

🚫 2️⃣ Why ALB Cannot Help Here

ALB (Application Load Balancer) works at Layer 7 (HTTP/HTTPS) and is used to:

* Distribute incoming traffic across EC2s, containers, or Lambda functions within or across Availability Zones in one region.
* It is not used for:
  + S3 access
  + Cross-region data transfer
  + Outbound connections from EC2 to another region or service

🧠 So, ALB cannot route traffic from an EC2 in Oregon to an S3 bucket in Sydney, because:

* S3 is a managed AWS service, not a target you can attach to an ALB.
* ALB targets must be registered instances, IPs, or Lambda functions.

✅ 3️⃣ What You Can Use Instead

If you want secure, automated, cross-region access, use:

| Use Case | AWS Service |
| --- | --- |
| EC2 in one region accessing S3 in another | IAM Role + S3 Bucket Policy |
| Cross-region replication between S3 buckets | S3 CRR |
| Web traffic routing between regions | Route53 + ALB |
| Private inter-region connectivity | VPC Peering or Transit Gateway |

🧩 4️⃣ When ALB *is* Useful

If your goal was to:

* Balance traffic between multiple EC2 instances, or
* Distribute requests to EKS pods, or
* Expose an application endpoint to users

Then ALB is perfect.

Example:

“User → Route53 → ALB (Oregon) → EC2s behind it.”

But not:

“EC2 → ALB → S3 (Sydney)” ❌

🗣️ 5️⃣ Short Interview Answer (20 sec):

“No, ALB cannot be used for that. ALB only distributes incoming HTTP/HTTPS traffic across EC2, EKS, or Lambda targets within the same region.  
For an EC2 instance to access an S3 bucket in another region, we use IAM Roles and S3 bucket policies, not ALB.”

💡 6️⃣ Pro Tip (If Interviewer Wants a Broader Answer):

“However, if the goal is to route application requests across multiple regions, we can use Route53 latency-based routing in combination with ALBs deployed in each region.  
That still won’t let EC2 talk to S3, but it enables cross-region load balancing for user traffic.”

**72 \* I have one S3 bucket in which im going to upload 3 files, one with 2TB, 4.9TB and 5.1TB.**

✅ Short Answer

* ✅ The 2 TB and 4.9 TB files can be uploaded successfully using Multipart Upload.
* ❌ The 5.1 TB file cannot be uploaded — it exceeds the maximum object size limit of 5 TB per object in Amazon S3.

🔹 Detailed Explanation

🧩 1. S3 Object Size Limits

| Type | Minimum | Maximum |
| --- | --- | --- |
| Single PUT upload | 0 bytes | 5 GB |
| Multipart upload | 5 MB | 5 TB (per object) |

👉 So, any file larger than 5 GB must use Multipart Upload, and anything above 5 TB is not allowed.

⚙️ 2. Uploading Large Files (2 TB and 4.9 TB)

Use Multipart Upload API — it splits the file into parts (each part 5 MB–5 GB) and uploads them in parallel.

Example using AWS CLI:

aws s3 cp largefile\_2tb.zip s3://my-bucket/ --storage-class STANDARD

✅ The AWS CLI automatically uses multipart upload for large files.

Benefits of Multipart Upload:

* Faster (parallel uploads)
* Fault-tolerant (resume failed parts)
* Efficient retry mechanism

⚠️ 3. The 5.1 TB File

* ❌ Will fail to upload because it exceeds the 5 TB maximum limit per object.
* You’ll get an error like:
* EntityTooLarge: Your proposed upload exceeds the maximum allowed object size

Possible Solutions:

1. Split the file into smaller chunks (e.g., two 2.55 TB parts) and upload separately.
2. Use Amazon S3 Glacier or AWS Snowball if you’re transferring very large datasets offline.
3. If it’s a data backup or video archive, split it at the application level before upload.

🧠 4. Best Practices

✅ Always use Multipart Upload for files > 100 MB  
✅ Keep each object ≤ 5 TB  
✅ For massive data transfer:

* Use AWS Snowball / Snowmobile (physical data transfer devices)
* Or AWS DataSync for automated large-scale data transfer

💬 Sample Interview Answer

“In Amazon S3, the maximum file size per object is 5 TB.  
So, the 2 TB and 4.9 TB files can be uploaded using Multipart Upload, which divides the file into smaller parts and uploads them in parallel.  
However, the 5.1 TB file will fail because it exceeds the 5 TB limit.  
To handle that, I’d split the file into smaller parts or use AWS Snowball for large offline transfers.”

**73 \* So how many files will get uploaded into S3 bucket?**

🧩 Possibility 1 — Single File Upload

If you upload one file to S3 (for example, via console, CLI, or API):

* ✅ Only one object gets uploaded.
* That file becomes one S3 object, regardless of size.

🧩 Possibility 2 — Multipart Upload (for large files >5GB)

If the file is large (e.g., 10 GB):

* S3 automatically splits the file into multiple parts (say 10 parts of 1GB each).
* Internally uploads multiple parts in parallel, then combines them into a single final object.
* ✅ Result: Still 1 file (1 object) in your bucket.

🧩 Possibility 3 — Cross-Region Replication (CRR) or Same-Region Replication (SRR)

If you have replication enabled between two buckets:

* Each file uploaded in source bucket is automatically copied to destination bucket.
* ✅ Result: For each file uploaded, you’ll have 1 copy in source and 1 in destination → 2 total files.

🧩 Possibility 4 — Upload Script / Batch Upload

If you’re uploading multiple files using a script (e.g., aws s3 sync):

* All files in that local folder will be uploaded to the S3 bucket.
* ✅ Number of files in S3 = Number of files in local folder (unless some are skipped/filtered).

🧠 In Summary

| Scenario | No. of Files in S3 | Explanation |
| --- | --- | --- |
| Normal upload | 1 | Single object created |
| Multipart upload | 1 | Split temporarily but merged into 1 object |
| Cross-region replication | 2 | One in source + one in destination |
| Upload folder of 100 files | 100 | Each file = one S3 object |

**74 \* I have 10 ec2 with ALB, each of the instance have 100GB EBS vol. One instance memory utilisation has crossed 80%**

🧠 Situation Recap

* You have 10 EC2 instances behind an Application Load Balancer (ALB)
* Each instance has a 100GB EBS volume
* One instance’s memory utilization > 80%

🎯 Goal

👉 Identify why memory usage is high  
👉 Fix it or reduce load  
👉 Ensure high availability (no user impact)

🧩 Step-by-Step Troubleshooting & Fix

Step 1️⃣ – Identify Which Instance Is Affected

* Go to EC2 → Monitoring → Memory Utilization (via CloudWatch agent)
* Or run this command on the instance:
* top
* free -m
* vmstat 5
* Identify the process consuming memory.

Step 2️⃣ – Check Application or Process

Run:

ps aux --sort=-%mem | head

✅ This shows the top memory-consuming processes.

Possible causes:

* Application memory leak
* Heavy traffic on that particular instance
* Background jobs or cron consuming memory
* Not enough swap memory

Step 3️⃣ – Verify Load Balancer Traffic Distribution

Check if ALB is sending more traffic to that instance:

* In EC2 → Target Groups → Monitoring, compare RequestCount and TargetResponseTime per instance.
* If one instance gets more load → enable stickiness check or re-balance traffic.

Step 4️⃣ – Take Immediate Actions

| Action | Description |
| --- | --- |
| Option 1: Restart the Service | Restart high-memory process to free memory temporarily. |
| Option 2: Reboot Instance | Temporary fix (not preferred long-term). |
| Option 3: Add Swap Memory | Helps handle temporary memory spikes. |
| Option 4: Scale Out | Increase instance count using Auto Scaling Group (ASG). |
| Option 5: Scale Up | Change instance type (e.g., t3.medium → t3.large). |

Step 5️⃣ – Analyze & Fix the Root Cause

* Check application logs in /var/log/ or /opt/ (depending on app).
* Look for memory leaks in Java, Node.js, or PHP apps.
* Enable CloudWatch alarms for memory usage:
* aws cloudwatch put-metric-alarm \
* --alarm-name "HighMemoryUsage" \
* --metric-name "mem\_used\_percent" \
* --namespace "CWAgent" \
* --statistic Average \
* --period 300 \
* --threshold 80 \
* --comparison-operator GreaterThanThreshold \
* --dimensions Name=InstanceId,Value=i-0123456789abcdef0 \
* --evaluation-periods 2 \
* --alarm-actions arn:aws:sns:region:account-id:alarm-topic

Step 6️⃣ – Long-Term Optimization

✅ Enable Auto Scaling Group with:

* Target tracking policy → based on average CPU/memory utilization
* Example:
  + If average memory > 70% → launch new instance
  + If < 40% → terminate extra instance

✅ Use Elastic Load Balancer (ALB) to distribute evenly among all healthy instances.

✅ Regularly monitor using CloudWatch Dashboards or AWS Compute Optimizer.

🧾 Summary Table

| Step | Action | Tool |
| --- | --- | --- |
| 1 | Identify instance | CloudWatch / top |
| 2 | Check process | ps aux |
| 3 | Check traffic distribution | Target Group metrics |
| 4 | Immediate fix | Restart / scale up / add swap |
| 5 | Root cause analysis | Logs / App profiling |
| 6 | Long-term fix | Auto Scaling / ALB tuning |

**75 \* and manager asked u to increase the vol to 1TB with zero downtime for all instances, how will u do that?**

🧠 Scenario Recap

* You have 10 EC2 instances behind an ALB
* Each instance uses a 100 GB EBS volume
* Manager asks to increase each to 1 TB
* Condition → Zero downtime

✅ Goal

✔ Resize (expand) EBS volumes without stopping instances  
✔ Ensure application remains available via ALB  
✔ Apply changes to all instances safely

🧩 Step-by-Step Process

Step 1️⃣ – Verify Volume Type Supports Expansion

EBS volume types that support resizing without downtime:

* gp2 / gp3 / io1 / io2 / st1 / sc1

❗ You cannot modify ephemeral (instance store) volumes.

Check volume type:

aws ec2 describe-volumes --volume-ids vol-xxxxxx --query "Volumes[\*].{ID:VolumeId,Type:VolumeType,Size:Size}"

Step 2️⃣ – Modify the Volume Size

Use AWS Console or CLI (preferred for all 10 instances).

CLI Example:

aws ec2 modify-volume --volume-id vol-0abcd1234efgh5678 --size 1024

You can run this command for each instance’s volume.

📌 Note:

* --size in GB
* This operation is instant, but the resizing process runs in background.

Step 3️⃣ – Check Volume Modification Progress

aws ec2 describe-volumes-modifications --volume-id vol-0abcd1234efgh5678

Wait until "ModificationState": "completed"

Step 4️⃣ – Expand the Filesystem (No Reboot)

Once the volume is resized, expand the partition and filesystem inside the OS.

For Linux:

Check device name:

lsblk

Run grow command (depending on filesystem type):

✅ For ext4:

sudo growpart /dev/xvda 1

sudo resize2fs /dev/xvda1

✅ For XFS (Amazon Linux 2 default):

sudo growpart /dev/xvda 1

sudo xfs\_growfs -d /

No need to reboot or unmount.  
This expands the partition and filesystem online 🚀

Step 5️⃣ – Validate

df -h

You should now see:

Filesystem Size Used Avail Use% Mounted on

/dev/xvda1 1000G ... ... ...

Step 6️⃣ – Automate for All 10 Instances

You can use an AWS Systems Manager (SSM) Run Command or script to apply the same modification to all instances in parallel.

Example (bash automation snippet):

for VOL in $(aws ec2 describe-volumes --filters "Name=tag:Role,Values=AppServer" --query "Volumes[\*].VolumeId" --output text)

do

aws ec2 modify-volume --volume-id $VOL --size 1024

done

Then use SSM Run Command:

aws ssm send-command \

--targets "Key=tag:Role,Values=AppServer" \

--document-name "AWS-RunShellScript" \

--parameters 'commands=["sudo growpart /dev/xvda 1", "sudo xfs\_growfs -d /"]'

This ensures zero downtime and uniform resize.

Step 7️⃣ – Monitoring

Track progress in:

* EC2 → Volumes → State: “in-use, optimizing”
* CloudWatch → VolumeModification metrics

🧾 Summary Table

| Step | Action | Command / Tool |
| --- | --- | --- |
| 1 | Check volume type | describe-volumes |
| 2 | Modify size | modify-volume |
| 3 | Monitor resize | describe-volumes-modifications |
| 4 | Expand filesystem | growpart + xfs\_growfs / resize2fs |
| 5 | Verify | df -h |
| 6 | Automate | SSM RunCommand or Bash loop |
| 7 | Monitor | CloudWatch metrics |

💡 Key Points

* Zero downtime ✅
* No need to detach or reboot ✅
* Safe, in-place operation ✅
* Works for both root & data volumes ✅

**76 \* In RDS, we have parameter group and an option group, whats the diff?**

🧩 1️⃣ Parameter Group → Controls Database Engine Settings

* Think of a Parameter Group as the “configuration file” for your database engine (like my.cnf for MySQL or postgresql.conf for PostgreSQL).
* It defines how the database engine behaves — memory usage, cache size, timeout, etc.

🧠 Examples:

| DB Engine | Example Parameters | Description |
| --- | --- | --- |
| MySQL | max\_connections, slow\_query\_log, innodb\_buffer\_pool\_size | Tuning and logging |
| PostgreSQL | work\_mem, shared\_buffers | Memory configuration |
| Oracle | open\_cursors, optimizer\_mode | Performance tuning |

✅ You can modify these parameters in a custom Parameter Group and apply it to your RDS instance.

Analogy:  
Parameter Group = “Database configuration settings.”

🧩 2️⃣ Option Group → Enables Additional Database Features

* Option Group is used to enable or manage additional features that are *not part of the default DB engine configuration*.
* These are optional add-ons like advanced security, monitoring, or management tools.

🧠 Examples:

| DB Engine | Example Option | Purpose |
| --- | --- | --- |
| Oracle | TDE, OEM, APEX | Transparent Data Encryption, Oracle Enterprise Manager |
| SQL Server | SQLSERVER\_BACKUP\_RESTORE, TDE | Native backup/restore, encryption |
| MySQL | MARIADB\_AUDIT\_PLUGIN, MEMCACHED | Logging, caching |

✅ You associate an Option Group to your RDS instance to enable these optional features.

Analogy:  
Option Group = “Extra features or plugins.”

🧾 3️⃣ Simple Comparison Table

| Feature | Parameter Group | Option Group |
| --- | --- | --- |
| Purpose | Controls DB engine settings | Enables additional DB features |
| Scope | Affects how DB behaves internally | Adds new capabilities |
| Examples | max\_connections, slow\_query\_log | TDE, OEM, Audit Plugin |
| Applied To | RDS instance (per DB engine type) | RDS instance (optional features) |
| Modify | Immediate or requires reboot | Enable/disable options |
| Supported In | All RDS engines | Selected engines (Oracle, SQL Server, MySQL) |

🧩 4️⃣ Example Scenario

You have a MySQL RDS and you want to:

* Increase the max\_connections → Use Parameter Group
* Enable MariaDB Audit Plugin → Use Option Group

🗣️ 5️⃣ Interview Short Answer (20 seconds)

“Parameter Groups in RDS control the database engine settings like memory, connections, and timeouts, similar to config files.  
Option Groups, on the other hand, enable additional features like encryption, auditing, or native backup options that are not part of the core database engine.  
Parameter Group = configuration; Option Group = extra features.”

**77 \* I have 2 subnets in VPC, how can I know which subnet is public?**

A public subnet is one that has a route to an Internet Gateway (IGW) in its route table.

🔹 Step-by-Step Explanation

🧩 1. Check the Route Table

Each subnet in a VPC is associated with a route table.

* Go to VPC → Route Tables in AWS Console
* Select the route table linked to each subnet
* Look for a route entry like:

Destination: 0.0.0.0/0

Target: igw-xxxxxxxx

✅ If the Target is an Internet Gateway (igw-), that subnet is Public.  
❌ If the Target is a NAT Gateway (nat-) or another subnet, it’s Private.

⚙️ 2. Example

| Subnet | Route Table Target | Subnet Type |
| --- | --- | --- |
| subnet-1 | 0.0.0.0/0 → igw-12345 | Public ✅ |
| subnet-2 | 0.0.0.0/0 → nat-98765 | Private ❌ |

🧠 3. Additional Indicators

✅ Public Subnet

* Route to Internet Gateway (IGW)
* Instances can have public IPs or Elastic IPs
* Can be accessed from the internet (if security groups allow)

❌ Private Subnet

* No direct IGW route
* Route to NAT Gateway or NAT Instance for outbound access
* No public IP assigned to instances

🧩 4. CLI Command Option

You can also verify using AWS CLI:

aws ec2 describe-route-tables \

--filters "Name=vpc-id,Values=vpc-xxxxxxxx" \

--query "RouteTables[].Routes[]"

Check for:

{

"DestinationCidrBlock": "0.0.0.0/0",

"GatewayId": "igw-xxxxxxxx"

}

→ That means the associated subnet is public.

💬 Sample Interview Answer

“To identify a public subnet, I check the subnet’s route table.  
If it has a route 0.0.0.0/0 pointing to an Internet Gateway (igw), it’s a public subnet.  
If it points to a NAT Gateway or has no IGW route, it’s private.  
Additionally, public subnets usually assign public IPs to EC2 instances.”

**78 \* There is a public IP and u have to restrict all type of access for this publicIp, how will u do that?**

Option 1 — Use Network ACL (NACL)

NACLs (Network Access Control Lists) operate at the subnet level and are stateless, which means you can explicitly *deny* traffic from an IP.

Steps:

1. Go to VPC → Network ACLs.
2. Choose the NACL associated with your subnet.
3. Add a rule in Inbound Rules and Outbound Rules:
4. Rule #: 100
5. Type: All traffic
6. Protocol: All
7. Source: 203.0.113.45/32
8. Action: DENY
9. Move this rule above any “Allow” rules (since NACLs are processed in order).

✅ Result:  
All traffic from/to that public IP is blocked at the subnet level.

Option 2 — Use Security Groups

Security Groups are stateful and don’t support explicit deny rules — they only allow what you define.  
So, to block a public IP:

* Do not add any inbound rule allowing that IP.
* Or, if already allowed, remove that rule.

⚠️ Limitation:  
You can’t create a “deny” rule directly in a security group — only “allow”.

Option 3 — Use AWS WAF (Web Application Firewall)

If your public IP is trying to access a website through ALB, CloudFront, or API Gateway, use AWS WAF.

Steps:

1. Create a WAF Web ACL.
2. Add a Rule → IP match condition:
3. Match type: IP set
4. IP set: 203.0.113.45/32
5. Action: BLOCK
6. Associate the Web ACL with your ALB or CloudFront distribution.

✅ Result:  
Any HTTP(S) request from that public IP is dropped before reaching your application.

Option 4 — Use Firewall Manager or AWS Network Firewall (for advanced setups)

If you have a VPC with multiple subnets and firewalls, you can use:

* AWS Network Firewall to define deny rules for a list of IPs.
* Or AWS Firewall Manager for organization-wide IP blocking.

🧠 Quick Summary

| Method | Where it Applies | Can Explicitly Deny? | When to Use |
| --- | --- | --- | --- |
| Security Group | Instance level | ❌ No | Basic EC2 control |
| NACL | Subnet level | ✅ Yes | Block IPs completely |
| WAF | ALB / CloudFront / API Gateway | ✅ Yes | Block malicious HTTP traffic |
| Network Firewall | VPC level | ✅ Yes | Enterprise-wide protection |

✅ Recommended Answer (Simple and Clear for Interview)

I’ll use a Network ACL to deny all inbound and outbound traffic from that public IP address.  
I can also use AWS WAF if the IP is accessing through a web application (via ALB or CloudFront).

**79 \* How does S3 bucket talk to NAT Gateway?**

Actually… the key here is:  
👉 S3 itself is a public AWS service, not inside your VPC — so your EC2 instances in private subnets need a way to reach S3 without direct internet access.

That’s where NAT Gateway (or VPC Endpoint) comes in.

🧩 1️⃣ Scenario

You have:

* Private subnet → EC2 instance
* No direct Internet Gateway access
* EC2 needs to access S3 bucket (for uploading logs, code, etc.)

⚙️ 2️⃣ How the Communication Happens (via NAT Gateway)

🔹 Step 1:

EC2 in private subnet sends a request to:

https://s3.ap-south-1.amazonaws.com

🔹 Step 2:

That traffic goes to the Route Table of the private subnet.

Route:

0.0.0.0/0 → NAT Gateway (in public subnet)

🔹 Step 3:

The NAT Gateway, which has access to the Internet Gateway (IGW),  
translates the private IP → public IP and routes the request to S3 over the internet.

🔹 Step 4:

S3 (public AWS service endpoint) responds back via NAT Gateway → EC2 instance.

✅ EC2 successfully communicates with S3 indirectly via NAT Gateway.

🧠 3️⃣ Key Notes

| Component | Purpose |
| --- | --- |
| EC2 (Private Subnet) | Needs S3 access but no direct internet |
| NAT Gateway (Public Subnet) | Provides outbound internet for private subnet |
| Internet Gateway | Allows NAT to reach S3 public endpoint |
| Route Table | Defines the traffic flow |
| S3 Public Endpoint | S3 service endpoint (reachable via internet) |

⚠️ 4️⃣ Drawback of Using NAT Gateway

* Data to S3 goes over the public internet (even if secure via HTTPS).
* It incurs data transfer charges (through NAT).
* Less secure compared to private connectivity.

✅ 5️⃣ Better (Recommended) Way — Use S3 VPC Endpoint

Instead of NAT Gateway, create an S3 Gateway VPC Endpoint.

Advantages:

* Private connectivity (no internet used)
* No NAT/data transfer cost
* More secure — stays inside AWS network

Route table entry:

Destination: com.amazonaws.ap-south-1.s3

Target: VPC Endpoint

💬 6️⃣ Interview Short Answer

“S3 is a public AWS service.  
When an EC2 instance in a private subnet needs to access S3,  
it sends traffic through the NAT Gateway in the public subnet.  
The NAT Gateway routes traffic to the S3 public endpoint via the Internet Gateway and returns the response.

However, the recommended approach is to use an S3 VPC Endpoint for secure, private connectivity without using NAT or the internet.”

🧩 7️⃣ Summary Diagram (Conceptual)

Private Subnet (EC2)

│

│ Route: 0.0.0.0/0 → NAT GW

↓

Public Subnet (NAT Gateway)

│

↓

Internet Gateway → S3 Public Endpoint

🟩 Alternative (Better):

Private Subnet (EC2)

│

↓

S3 VPC Endpoint → S3 (Private AWS Network)

**80 \* Which AWS service will be best suited for persistent vol for a specific pod in a cluster which wil b catering 10000 requests.**

Excellent — this is a Kubernetes + AWS integration interview question 👏

Let’s restate it:

“Which AWS service is best suited for persistent volume for a specific pod in a Kubernetes cluster that handles 10,000 requests?”

✅ Short, Direct Answer

The best-suited AWS service for persistent volume in this case is Amazon EBS (Elastic Block Store) — specifically gp3 (General Purpose SSD) or io1/io2 (Provisioned IOPS SSD) volumes depending on IOPS requirements.

🔹 Detailed Explanation

🧩 1. Persistent Volume (PV) in Kubernetes

A Persistent Volume is used to store data beyond the pod’s lifecycle — meaning the data remains even if the pod is deleted or recreated.

When running Kubernetes on AWS (like EKS), persistent volumes can be backed by AWS storage services like:

* EBS (Elastic Block Store)
* EFS (Elastic File System)
* FSx for Lustre
* S3 (via CSI plugins for object storage)

⚙️ 2. Best Option: Amazon EBS

Why EBS?

* Block-level storage attached to a specific EC2 node.
* High performance and low latency.
* Perfect for a single pod that needs fast, durable, and consistent storage.

Types of EBS Volumes:

| Type | Performance | Use Case |
| --- | --- | --- |
| gp3 | General purpose SSD (3,000 IOPS baseline) | Default choice for most workloads |
| io1 / io2 | Provisioned IOPS SSD (up to 64,000 IOPS) | High-performance DBs, high-traffic apps |
| st1 / sc1 | Throughput optimized HDD | Large, sequential workloads |

For handling 10,000+ requests, use io1/io2 or highly tuned gp3 volumes with higher IOPS.

⚙️ 3. Alternative Option: Amazon EFS (Elastic File System)

If multiple pods across different nodes need to share the same data, use EFS.

| Feature | EBS | EFS |
| --- | --- | --- |
| Access | Single EC2 node | Multi-node (shared access) |
| Performance | Low latency | Scalable throughput |
| Use Case | Database, single pod storage | Web content, shared files, ML model storage |

⚙️ 4. How It Works in EKS

In your PersistentVolume (PV) YAML:

* Define a StorageClass that uses the EBS CSI driver.
* Kubernetes automatically provisions an EBS volume.
* Attach it to the pod as a persistent volume claim (PVC).

Example:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: app-data-pvc

spec:

accessModes:

- ReadWriteOnce

storageClassName: gp3

resources:

requests:

storage: 50Gi

This will dynamically provision an EBS volume in AWS.

🧠 Final Recommendation

| Requirement | Best AWS Storage |
| --- | --- |
| Single pod persistent volume | Amazon EBS (gp3 or io2) |
| Shared volume across pods | Amazon EFS |
| Object-based storage (not true PV) | Amazon S3 via CSI plugin |

✅ Sample Interview Answer

“For a specific pod that needs persistent storage and serves high traffic (10,000+ requests), I’d use Amazon EBS as the backend for the persistent volume.  
It offers low latency and can be tuned for high IOPS using gp3 or io2 volumes.  
If multiple pods needed shared access, I’d go with Amazon EFS instead.”

**81 \* Data shd b accessible at all point of time? S3 or EBS or EFS drive**

⚙️ Quick Summary

| Storage Type | Access Type | Availability | Multi-instance Access | Use Case |
| --- | --- | --- | --- | --- |
| 🪣 S3 (Simple Storage Service) | Object storage (via HTTP API) | ✅ Highly available (99.99%) | ✅ Yes (global access) | Backup, static data, logs, images, shared files |
| 💽 EBS (Elastic Block Store) | Block storage (attached to one EC2) | ⚠️ Limited to one AZ | ❌ No (attached to one EC2 at a time) | OS disk, database volume |
| 📁 EFS (Elastic File System) | File storage (NFS mount) | ✅ Highly available across multiple AZs | ✅ Yes (multiple EC2s can access) | Shared data for multiple servers |

🔍 Detailed Explanation

1️⃣ S3 (Simple Storage Service)

* Object storage — stores data as objects (files + metadata).
* Designed for durability (99.999999999%) and availability (99.99%).
* Data accessible from anywhere (via URL, SDK, or API).
* Doesn’t depend on EC2 — even if servers go down, data remains available.
* Ideal for:
  + Backups
  + Static websites
  + Application data (logs, reports, images)
  + Cross-region access

✅ Best for long-term, globally available storage.

2️⃣ EBS (Elastic Block Store)

* Block-level storage attached to one EC2 instance at a time.
* Data persists even if the instance stops, but:
  + It’s tied to a single Availability Zone (AZ).
  + Not accessible if the instance or AZ fails.
* Suitable for:
  + Boot volumes
  + Databases (MySQL, PostgreSQL)
  + Application disks

⚠️ Not ideal for “accessible at all times” — single AZ failure can cause downtime.

3️⃣ EFS (Elastic File System)

* Managed NFS file system, accessible from multiple EC2 instances across multiple AZs.
* Auto-scales storage and throughput.
* Data is available continuously (even if one AZ fails).
* Ideal for:
  + Shared application data
  + Web servers needing shared content
  + Home directories

✅ High availability and shared access within a region.

🧠 So — Which One to Choose?

| Requirement | Best Option | Reason |
| --- | --- | --- |
| Data must be accessible anytime, anywhere | 🪣 S3 | Global, highly durable and available |
| Data must be shared between EC2s in same region | 📁 EFS | Multi-AZ, mountable on multiple instances |
| Data is attached to a single instance (like a disk) | 💽 EBS | Persistent block storage, but not multi-AZ |

✅ Final Answer:

If data must be accessible at all points of time (high availability, durable, independent of instance) →  
Use Amazon S3.

If you mean shared storage between EC2s in a VPC, use EFS.

**82 \* How will migrate the app from onprime to AWS Cloud**

☁️ Goal

Move your on-premises application (servers, databases, storage, etc.) to AWS while ensuring:

* Minimal downtime
* Data integrity
* Cost efficiency
* Security and performance

🧩 High-Level Migration Phases

| Phase | Description | Example Tools |
| --- | --- | --- |
| 1️⃣ Assessment | Understand current infrastructure, dependencies, and costs. | AWS Migration Evaluator, Application Discovery Service |
| 2️⃣ Planning | Decide migration strategy, target AWS services, and timeline. | AWS Migration Hub, CloudEndure |
| 3️⃣ Migration | Move data, applications, and databases to AWS. | Server Migration Service (SMS), Database Migration Service (DMS), CloudEndure |
| 4️⃣ Testing | Verify functionality, performance, and security on AWS. | Load testing, pilot environment |
| 5️⃣ Cut-over & Go-Live | Switch production traffic to AWS environment. | Route 53, ELB |
| 6️⃣ Optimization | Monitor, right-size, and automate resources. | CloudWatch, Trusted Advisor |

🔧 Migration Strategies (6 R’s)

| Strategy | Description | Example |
| --- | --- | --- |
| Rehost (Lift & Shift) | Move servers as-is to EC2. | CloudEndure / SMS |
| Replatform (Lift, Tinker & Shift) | Slight modifications (e.g., move DB to RDS). | EC2 + RDS |
| Repurchase | Move to SaaS (e.g., on-prem CRM → Salesforce). | Replace app |
| Refactor / Re-architect | Redesign app for cloud-native (e.g., microservices). | Lambda, ECS, DynamoDB |
| Retire | Decommission unused apps. | – |
| Retain | Keep on-prem temporarily. | – |

🏗️ Typical Migration Architecture Example

On-Premises

├── App Servers

├── DB Server

├── Storage (NFS/SAN)

↓

AWS Cloud

├── EC2 (App Servers)

├── RDS (Database)

├── S3 (Storage)

├── ELB (Load Balancer)

├── Route 53 (DNS)

⚙️ Example Migration Flow

1. Assess On-Prem Environment
   * Inventory servers, DBs, network, storage.
   * Identify dependencies and data volume.
2. Design Target AWS Architecture
   * Create VPC, subnets, security groups, IAM.
   * Plan EC2, RDS, and storage sizing.
3. Migrate Data
   * Use AWS DMS for database migration (supports continuous replication).
   * Use AWS Storage Gateway or S3 Transfer Acceleration for large data.
4. Migrate Servers
   * Use CloudEndure Migration or AWS Application Migration Service to replicate on-prem VMs to AWS EC2.
5. Testing Phase
   * Verify application connectivity and performance.
   * Run functional and UAT tests.
6. Cutover
   * Update Route 53 DNS to point to AWS load balancer.
   * Stop on-prem traffic gradually.
7. Optimization
   * Enable Auto Scaling, CloudWatch monitoring, Cost Explorer for savings.

🧰 AWS Tools Commonly Used

| Purpose | Tool |
| --- | --- |
| Discovery | AWS Application Discovery Service |
| Server Migration | AWS Application Migration Service (CloudEndure) |
| Database Migration | AWS DMS |
| File Transfer | AWS DataSync / S3 Transfer Acceleration |
| Monitoring | CloudWatch, CloudTrail |
| DNS Cutover | Route 53 |

🧠 Example: Real Scenario

* On-prem web app running on Tomcat + MySQL  
  ➡️ Migrate to AWS as:
* Tomcat app → EC2 Auto Scaling group
* MySQL → Amazon RDS MySQL
* Storage → S3
* Load balancing → Application Load Balancer
* DNS → Route 53

✅ Summary

| Step | Action |
| --- | --- |
| 1 | Assess on-prem setup |
| 2 | Design AWS architecture |
| 3 | Choose migration strategy (Rehost/Replatform/etc.) |
| 4 | Use AWS tools (CloudEndure, DMS, DataSync) |
| 5 | Test migrated workloads |
| 6 | Perform final cutover |
| 7 | Optimize & monitor |

**83 \* Suppose I have multiple accounts and need to create AWS infrastructure? How can I do that**

🧩 1️⃣ Problem Context

You have multiple AWS accounts (for example:

* Dev,
* QA,
* Staging,
* Production,
* Security, etc.)

You need to create and manage AWS infrastructure (like VPCs, EC2s, IAM roles, S3, etc.) across all these accounts — efficiently, consistently, and securely.

⚙️ 2️⃣ Solution Overview – How to Manage Multi-Account Infra

You can do it using one or a combination of these AWS tools and services 👇

🧩 Option 1: AWS Organizations + Service Control Policies (SCPs)

Best for: Centralized multi-account management.

* Create a Management (root) account using AWS Organizations.
* Add multiple Member accounts under it.
* Apply Service Control Policies (SCPs) to restrict what services/accounts can do.
* Enable Consolidated Billing and Single Sign-On (SSO) if needed.

✅ Benefits:

* Central governance
* Unified billing
* Security and compliance control
* Can combine with Control Tower

🧩 Option 2: AWS Control Tower

Best for: Setting up and managing secure, compliant multi-account environments automatically.

* Control Tower automates account creation, IAM guardrails, logging, and monitoring.
* It uses Landing Zones — preconfigured environments with best practices.
* Integrates with AWS Organizations, CloudFormation, and Service Catalog.

✅ Benefits:

* Automated setup
* Enforced governance
* Standardized environments across accounts

🧩 Option 3: AWS CloudFormation StackSets

Best for: Deploying the same infrastructure (IaC) across multiple accounts and regions.

* Write a CloudFormation template (YAML/JSON) defining your infrastructure.
* Use StackSets to deploy that stack into multiple AWS accounts and regions simultaneously.

✅ Example:  
You can create one VPC, EC2, or S3 setup and deploy it to 10+ accounts automatically.

✅ Benefits:

* Consistent infra setup
* Easy rollback
* Fully automated

🧩 Option 4: Terraform / IaC Tools

Best for: Cross-account, cross-region automation using code.

* Use Terraform with assume-role for each AWS account.
* Maintain separate state files or workspaces for each account.

✅ Benefits:

* Multi-cloud and multi-account support
* Version control using Git
* Automation via CI/CD pipelines

🧩 Option 5: AWS Single Sign-On (AWS SSO) / IAM Identity Center

Best for: Managing user access across multiple AWS accounts.

* Integrates with AWS Organizations
* Assigns users to multiple accounts with specific roles
* Centralized access control

🧠 3️⃣ Example Real-World Implementation

“In our organization, we manage around 10 AWS accounts — Dev, QA, and Prod.  
We use AWS Organizations and Control Tower to centrally manage all accounts.  
Infrastructure deployment is automated via CloudFormation StackSets and Terraform through a Jenkins CI/CD pipeline.  
IAM roles and access are handled using AWS SSO, ensuring security and least-privilege access.”

💬 4️⃣ Short Interview Answer

“If I have multiple AWS accounts and need to create infrastructure,  
I’ll use AWS Organizations for centralized management,  
Control Tower for governance and account setup,  
and CloudFormation StackSets or Terraform to automate infrastructure deployment across accounts.  
This ensures consistency, compliance, and automation at scale.”

✅ 5️⃣ Summary Table

| Tool / Service | Purpose | Use Case |
| --- | --- | --- |
| AWS Organizations | Multi-account governance | Structure, billing, policy control |
| Control Tower | Automated multi-account setup | Landing zones, compliance |
| CloudFormation StackSets | Multi-account deployment | IaC automation |
| Terraform | Cross-account automation | CI/CD + version control |
| AWS SSO / IAM Identity Center | User access management | Centralized identity control |

**84 \* Which type of instances are using in AWS**

Amazon EC2 (Elastic Compute Cloud) offers different types of instances optimized for various use cases — compute, memory, storage, and GPU-intensive workloads.

🧩 1️⃣ EC2 Instance Families Overview

| Category | Family | Optimized For | Example Use Case |
| --- | --- | --- | --- |
| General Purpose | t, m | Balanced CPU, memory, and networking | Web servers, dev/test environments |
| Compute Optimized | c | High CPU-to-memory ratio | Batch processing, gaming servers, scientific modeling |
| Memory Optimized | r, x, u, z | Large memory footprint | Databases, caching (Redis, SAP HANA) |
| Storage Optimized | i, d, h | High disk throughput and IOPS | Data warehousing, big data, Hadoop |
| Accelerated Computing (GPU/FPGA) | p, g, inf, f, trn | GPU/ML acceleration, high-performance compute | Machine learning, AI, deep learning, 3D rendering |

🧩 2️⃣ Common EC2 Instance Families and Examples

| Type | Full Name | Use Case | Example Instance Types |
| --- | --- | --- | --- |
| T Family (Burstable) | General Purpose | Low-cost, burstable CPU | t3.micro, t4g.small |
| M Family | General Purpose | Balanced compute/memory | m5.large, m6g.medium |
| C Family | Compute Optimized | CPU-heavy workloads | c6i.large, c7g.xlarge |
| R Family | Memory Optimized | High-memory workloads | r6i.xlarge, r7g.2xlarge |
| X Family | Extra Memory Optimized | In-memory databases like SAP HANA | x2idn.32xlarge |
| I Family | Storage Optimized | High IOPS and throughput | i3.large, i4i.4xlarge |
| D Family | Dense Storage | Data-intensive apps, Hadoop | d2.2xlarge, d3.4xlarge |
| P Family | GPU Instances | ML/DL training, AI workloads | p3.8xlarge, p4d.24xlarge |
| G Family | Graphics | Video rendering, streaming | g5.xlarge, g6e.12xlarge |
| Inf Family | Inference Optimized | ML inference workloads | inf1.xlarge |
| Trn Family | Training Optimized | Large-scale ML model training | trn1.32xlarge |
| F Family | FPGA Accelerated | Custom hardware acceleration | f1.2xlarge |
| H Family | High Storage Throughput | Hadoop, distributed file systems | h1.4xlarge |

🧩 3️⃣ Instance Naming Convention

Example:  
m5.large

| Part | Meaning |
| --- | --- |
| m | Family (General Purpose) |
| 5 | Generation (newer = better performance) |
| large | Size (CPU/RAM scale, e.g. micro → xlarge → 16xlarge) |

🧩 4️⃣ Instance Types Based on Architecture

| Architecture | Family Examples | Notes |
| --- | --- | --- |
| x86-based | m6i, c6i, r6i | Intel/AMD processors |
| ARM-based (Graviton) | m6g, c6g, r6g | AWS Graviton2/3 processors — better cost/performance ratio |

🧩 5️⃣ Real-World Example

* For a web server → t3.medium or m5.large (general purpose)
* For database (RDS, Redis) → r6i.large (memory optimized)
* For AI/ML training → p3.2xlarge or p4d.24xlarge (GPU optimized)
* For batch processing → c6i.large (compute optimized)
* For big data / Hadoop → i3.4xlarge (storage optimized)

🧩 6️⃣ Interview Example Answer

“AWS offers multiple instance families based on workload needs.  
For example, I use T or M series for general-purpose applications, C series for compute-heavy workloads, R or X series for memory-intensive databases, and P or G series for GPU-based machine learning or graphics workloads.

I choose instance type based on CPU, memory, storage, and network requirements — for example, a web app server may run on m6i.large, while a deep learning model would use p4d.24xlarge.”

**85 \* There is surge in traffic how you will maintain frontend and backend?**

🔹 Scenario: Surge in Traffic

You have a web application with:

* Frontend: Web servers / Application Load Balancer
* Backend: Database, APIs, or microservices

You need to maintain performance, availability, and stability during a sudden increase in user traffic.

🔹 Frontend Handling (Web/App Layer)

1. Auto Scaling

* Use EC2 Auto Scaling Groups to automatically add more instances when CPU, memory, or network usage goes high.
* Define scaling policies:
  + Target tracking: maintain average CPU < 70%
  + Step scaling: add 2 instances if traffic > threshold
* After the traffic reduces, scale in to save cost.

✅ Example:

Target CPU utilization: 60%

Min instances: 2

Max instances: 10

2. Elastic Load Balancing (ELB / ALB / NLB)

* Place an Application Load Balancer (ALB) in front of your web servers.
* It automatically distributes incoming requests across multiple instances and AZs.
* Enable Cross-Zone Load Balancing to evenly spread traffic.

3. Use CDN (CloudFront)

* Amazon CloudFront caches static content (images, CSS, JS) at edge locations worldwide.
* Reduces load on your servers and latency for users.

4. Enable Caching

* Use Elasticache (Redis or Memcached) to cache frequently requested data or API responses.
* Reduces repeated database hits and improves response times.

5. Use Auto Scaling for Containers

* If using ECS / EKS, enable Service Auto Scaling to add more containers dynamically.

6. Use Route53 with Health Checks

* Route traffic to healthy endpoints or regions using weighted or latency-based routing.
* Helps handle regional surges effectively.

🔹 Backend Handling (Database & Services Layer)

1. Read Replicas

* For RDS, use read replicas to handle increased read traffic.
* The primary DB handles writes, replicas handle reads.

2. Database Scaling

* Use RDS Auto Scaling for storage.
* Move to Aurora Serverless, which automatically adjusts capacity.

3. Caching at Backend

* Use Elasticache between application and DB.
* Example: Cache product data, user sessions, etc.

4. Queue Processing

* Use Amazon SQS (Simple Queue Service) to handle spikes in requests asynchronously.
* Frontend pushes requests into a queue; backend processes them gradually.

5. Microservices Architecture

* Break down monolithic backend into independent services using API Gateway + Lambda / ECS.
* Easier to scale specific services that face heavy load.

🔹 Monitoring & Alerts

* Use CloudWatch for:
  + CPU, memory, request count, latency metrics
* Set up Alarms and Auto Scaling Policies
* Use AWS X-Ray for tracing and performance debugging

🔹 Summary Table

| Layer | AWS Service | Purpose |
| --- | --- | --- |
| Web Frontend | ALB / CloudFront | Distribute & cache traffic |
| App Servers | Auto Scaling Group | Scale instances dynamically |
| Cache | ElastiCache | Reduce backend load |
| Database | RDS Read Replicas / Aurora | Scale reads and storage |
| Messaging | SQS / SNS | Handle async requests |
| Monitoring | CloudWatch / X-Ray | Track metrics & performance |

✅ In short:

“Scale horizontally in frontend with Load Balancer & Auto Scaling, and optimize backend using caching, read replicas, and queues.”

**86 \* why we have to use SSL certificate**

🔐 What is SSL/TLS?

SSL (Secure Sockets Layer) or its newer version TLS (Transport Layer Security) is a security protocol that encrypts communication between a client (like a web browser) and a server (like your website).

💡 Why We Use SSL Certificates

| Purpose | Explanation |
| --- | --- |
| 🔒 1. Encrypts Data | SSL encrypts all data transferred between user and server — like passwords, credit card info, etc. — so even if someone intercepts it, they can’t read it. |
| 🧾 2. Authentication | Confirms that your website really belongs to you — not an attacker pretending to be your site (prevents phishing or “man-in-the-middle” attacks). |
| ✅ 3. Data Integrity | Ensures data isn’t changed or corrupted while transferring between server and browser. |
| 🌍 4. Trust and SEO | Browsers show a padlock icon 🔒 for secure sites. Google also ranks HTTPS sites higher than non-secure ones. |
| 💳 5. Required for Sensitive Data | If your site accepts logins, payments, or any personal data — SSL is mandatory for compliance (like PCI-DSS). |

⚙️ How It Works (Simplified Flow)

1. User visits https://yourwebsite.com
2. The browser checks the website’s SSL certificate.
3. If valid, the browser and server establish an encrypted connection using a public/private key handshake.
4. All data sent after that is encrypted.

🧱 Example

Without SSL:

http://yourwebsite.com/login

➡ Data like username & password travels in plain text.

With SSL:

https://yourwebsite.com/login

➡ Data is encrypted — hackers can’t read it even if they intercept.

🏁 In Short

🔐 SSL = Security + Trust + Encryption + Authentication + Better SEO

**89 \* why we need ELB & how you select which type of loadbalancer suitable for my application.**

🧩 1️⃣ Why We Need ELB

Elastic Load Balancer (ELB) automatically distributes incoming network traffic across multiple targets — such as EC2 instances, containers, IP addresses, or Lambda functions — within one or more Availability Zones (AZs).

It ensures:

* High availability
* Fault tolerance
* Scalability
* Security

🔹 Main Reasons We Need ELB

| Purpose | Explanation |
| --- | --- |
| High Availability | Distributes traffic across multiple AZs — if one instance fails, traffic automatically goes to healthy ones. |
| Scalability | Works with Auto Scaling Groups to add/remove instances dynamically. |
| Health Monitoring | Performs health checks and routes traffic only to healthy instances. |
| Security | Integrates with AWS Certificate Manager (ACM) for SSL/TLS, and WAF/Shield for protection. |
| Seamless Routing | Supports sticky sessions, path-based routing, and weighted target groups. |

🧩 2️⃣ Types of Load Balancers in AWS

| Load Balancer | Protocols Supported | Use Case | Layer |
| --- | --- | --- | --- |
| Application Load Balancer (ALB) | HTTP, HTTPS, WebSocket | For web apps, microservices, container apps | Layer 7 |
| Network Load Balancer (NLB) | TCP, UDP, TLS | For high-performance, low-latency apps, databases, gaming servers | Layer 4 |
| Gateway Load Balancer (GWLB) | IP (GENEVE) | For deploying security appliances like firewalls, IDS, IPS | Layer 3 |
| Classic Load Balancer (CLB) *(Legacy)* | HTTP, HTTPS, TCP, SSL | Older apps, backward compatibility | Layer 4/7 |

🧩 3️⃣ How to Select the Right Load Balancer

Let’s match the application requirement to the correct ELB type 👇

| Application Requirement | Recommended Load Balancer | Why |
| --- | --- | --- |
| Web applications using HTTP/HTTPS | ALB | Supports Layer 7 features like path-based routing (/api, /images) and host-based routing (app1.example.com). |
| Real-time gaming, streaming, or VoIP | NLB | Handles millions of requests per second with ultra-low latency. |
| Microservices (ECS, EKS, API-based) | ALB | Integrates with containers, supports target groups per service. |
| Database replication, internal TCP apps | NLB | Works well for TCP-based apps like MySQL, Redis, etc. |
| Security/inspection appliances (firewalls, IDS) | GWLB | Integrates 3rd-party or custom appliances transparently. |
| Legacy apps (still using EC2-Classic) | CLB | For backward compatibility only (not recommended for new use). |

🧩 4️⃣ Real-World Example

Scenario 1 – Web Application

You have a web app running on EC2, serving HTTP/HTTPS traffic to users.  
✅ Use Application Load Balancer (ALB)  
Reason: You can route requests like

* /api → API servers
* /images → image servers
* Supports SSL termination and WebSockets.

Scenario 2 – Banking/Financial Application

TCP-based app requiring very low latency and static IP.  
✅ Use Network Load Balancer (NLB)  
Reason: Operates at Layer 4, supports TLS offloading, and can handle millions of requests per second.

Scenario 3 – Deploying Firewalls or Security Devices

You want to insert third-party firewalls inline to inspect traffic.  
✅ Use Gateway Load Balancer (GWLB)  
Reason: It distributes traffic using the GENEVE protocol to appliance instances.

🧩 5️⃣ Additional ELB Features You Can Mention

* Cross-Zone Load Balancing → balances traffic across all AZs
* SSL Termination → offloads encryption from EC2
* Sticky Sessions → keeps users connected to same backend
* Integration with Auto Scaling → auto adds/removes instances
* Target Groups → separate backend sets for microservices
* Logging → stores access logs in S3 for auditing

💬 Example Interview Answer

“We use an Elastic Load Balancer to distribute incoming traffic across multiple instances for high availability, fault tolerance, and scalability.

It automatically performs health checks and routes traffic only to healthy targets.

To choose the right load balancer, I look at the type of traffic:

* For web and HTTP/HTTPS applications, I use an Application Load Balancer (ALB) because it supports Layer 7 routing and path-based rules.
* For low-latency TCP/UDP applications, I use a Network Load Balancer (NLB).
* For security appliances, I use a Gateway Load Balancer (GWLB).

These configurations, combined with Auto Scaling and multi-AZ deployment, ensure high availability and performance for the application.”

**90 \* crosszone Loadbalancer**

🔹 Cross-Zone Load Balancing (AWS Load Balancer)

Cross-Zone Load Balancing is a feature in AWS Load Balancers that distributes incoming traffic evenly across all targets (like EC2 instances) in all Availability Zones (AZs) — not just within the zone where the load balancer node is located.

🧠 Why It Matters

Without cross-zone load balancing, traffic from a load balancer node in one Availability Zone is routed only to targets in the same zone, which can cause uneven load distribution if the number of targets differs between zones.

With cross-zone load balancing enabled, the load balancer can route requests across all healthy targets in all zones, improving performance and availability.

🔹 Example:

Without Cross-Zone Load Balancing

* You have:
  + 2 AZs: us-east-1a and us-east-1b
  + 2 EC2 instances in 1a, 4 EC2 instances in 1b
* Each Load Balancer node only routes to targets in its own zone.
* Result:
  + us-east-1a targets get more traffic per instance (heavier load).

With Cross-Zone Load Balancing

* Each Load Balancer node routes traffic to all 6 EC2 instances (across both AZs).
* Result:
  + Even traffic distribution among all instances.

🔹 Supported Load Balancers

| Load Balancer Type | Cross-Zone Support | Default Setting | Notes |
| --- | --- | --- | --- |
| Application Load Balancer (ALB) | ✅ Yes | Enabled by default, cannot disable in Console (can via API/CLI) | No data transfer charges between AZs |
| Network Load Balancer (NLB) | ✅ Yes | Disabled by default | When enabled, there are cross-AZ data transfer costs |
| Gateway Load Balancer (GWLB) | ✅ Yes | Enabled by default | Similar to ALB behavior |

🔹 How to Enable (Console Steps):

1. Go to EC2 → Load Balancers
2. Select your Load Balancer
3. Choose the Attributes tab
4. Under Cross-Zone Load Balancing, click Edit
5. Turn ON the toggle → Save changes ✅

🔹 Benefits

* Better load distribution across targets
* Improved fault tolerance
* Increased overall availability

⚠️ Considerations

* For Network Load Balancer, enabling this incurs inter-AZ data transfer charges.
* For Application Load Balancer, it’s free and usually best to keep enabled.

🧩 Example Use Case

You have an ALB with targets in 3 AZs:

* AZ1: 1 EC2
* AZ2: 2 EC2s
* AZ3: 3 EC2s  
  With cross-zone load balancing ON → all 6 EC2s share the load evenly.  
  With it OFF → AZ1’s instance could get overloaded.

**92 \* Suppose you have ur dns in one server example xyz.com and one more asdf.com in another computer server how your server xyz.com knows asdf.com also your how you do that**

🌍 Step-by-Step Explanation of How It Works

1. Each domain is registered with a Top-Level Domain (TLD) server
   * When you register xyz.com, it’s stored in the “.com” TLD DNS servers with information about which Name Servers (NS) are responsible for it.  
     Example:
   * xyz.com → ns1.xyz.com, ns2.xyz.com
   * Similarly:
   * asdf.com → ns1.asdf.com, ns2.asdf.com
2. When someone from xyz.com wants to reach asdf.com
   * The request goes to the local DNS resolver (like your ISP or corporate DNS).
   * The resolver doesn’t know asdf.com directly, so it follows the DNS resolution path:
   * Root DNS Server → .com TLD Server → asdf.com’s Authoritative Name Server
   * Then it retrieves the IP address for asdf.com.
3. xyz.com doesn’t need to “know” asdf.com directly
   * Because DNS is hierarchical and distributed, no single DNS server stores all domain records.
   * Each DNS server just knows how to refer queries to the next level (Root → TLD → Domain → Subdomain).
4. If you want xyz.com to directly know about asdf.com  
   You have a few configuration options:
   * 🔹 DNS forwarding / conditional forwarding:  
     Configure the DNS server for xyz.com to forward requests for asdf.com to the DNS server managing asdf.com.  
     Example (in BIND or Windows DNS):
   * Forward zone: asdf.com → 192.168.2.10 (IP of asdf.com DNS)
   * 🔹 Add a record manually:  
     In xyz.com’s DNS, add a record like:
   * asdf.com. IN A 192.168.2.10

(This works if both domains are in your private network.)

* + 🔹 Use public DNS:  
    If both domains are public (on the Internet), DNS resolution happens automatically via root and TLD servers — no manual link needed.

⚙️ Summary Table

| Case | How xyz.com Knows asdf.com |
| --- | --- |
| Both domains public | DNS hierarchy handles it automatically |
| Both internal/private | Use conditional forwarding or host entries |
| Hybrid (one public, one private) | Use split DNS or forwarder configuration |

**94 \* Public subnet private subnet**

🌐 Public Subnet

Definition:  
A subnet that has direct access to the Internet through an Internet Gateway (IGW).

Key Characteristics:

* Route Table: Has a route to the Internet Gateway (IGW).  
  Example:
* Destination: 0.0.0.0/0 → Target: igw-xxxxxxxx
* Use Cases:
  + Web servers that need to be accessed publicly
  + Bastion hosts (jump servers)
  + Load balancers (like Application Load Balancer)

Example:  
If you deploy an EC2 instance in a public subnet, you can assign a public IP or Elastic IP to access it directly from the internet.

🔒 Private Subnet

Definition:  
A subnet that does not have direct Internet access. It connects to the Internet indirectly via a NAT Gateway (for outbound) or through internal routing.

Key Characteristics:

* Route Table: No direct route to the Internet Gateway. Instead, it routes to a NAT Gateway for outbound traffic.  
  Example:
* Destination: 0.0.0.0/0 → Target: nat-xxxxxxxx
* Use Cases:
  + Application servers, databases, backend services
  + Instances that should not be accessed from the Internet
  + Internal processing workloads

Example:  
If you deploy a database in a private subnet, it won’t have a public IP and cannot be reached directly from the Internet. Only other instances within the VPC or via a bastion host can connect to it.

🧩 Quick Comparison Table

| Feature | Public Subnet | Private Subnet |
| --- | --- | --- |
| Internet Access | Yes (via Internet Gateway) | No (only via NAT Gateway) |
| Public IP Allowed | Yes | No |
| Route Table | Points to IGW | Points to NAT Gateway or none |
| Typical Use | Web servers, Bastion host | Databases, Application servers |
| Security | Lower (public exposure) | Higher (internal only) |

**95 \* Cloudtrail**

AWS CloudTrail is a service that enables governance, compliance, operational auditing, and risk auditing of your AWS account.

It records API calls and user activities made in your AWS account, so you can see who did what, when, and from where.

🔹 Key Features of CloudTrail:

1. Tracks all API calls
   * Logs all API requests made through the AWS Management Console, AWS SDKs, CLI, or other AWS services.
2. Captures detailed event information
   * Records:
     + The user identity (who made the request)
     + Time of the request
     + Source IP address
     + Services and actions performed
     + Request and response parameters
3. Stores logs in S3
   * Events are stored as log files in an Amazon S3 bucket (you specify which one).
   * Optionally integrate with CloudWatch Logs for real-time monitoring.
4. Event history
   * By default, AWS CloudTrail provides the last 90 days of management events in the console for free.
5. Multi-region tracking
   * You can configure CloudTrail to log events across all regions automatically.
6. Supports organizational trails
   * For AWS Organizations, you can enable a single trail for all member accounts.

🔹 Types of Events Logged:

1. Management Events
   * Operations that change or view AWS resource configurations (e.g., CreateUser, DeleteBucket).
2. Data Events
   * Operations performed on resources (e.g., S3 object-level activity like GetObject, PutObject).
3. Insight Events
   * Detect unusual activity, like sudden spikes in API calls.

🔹 Use Cases:

* Security Auditing → Track unauthorized access or suspicious actions.
* Compliance → Meet requirements (HIPAA, PCI-DSS, etc.).
* Troubleshooting → Investigate API errors or failed operations.
* Monitoring Changes → Know who modified your infrastructure or configurations.

🔹 Example:

Suppose a user deleted an EC2 instance accidentally.  
You can go to CloudTrail → Event History and check:

* The username or IAM role that performed the delete action.
* The timestamp and region of the event.
* The source IP address used.
* The API action (TerminateInstances).

🔹 Integration Options:

* CloudWatch Logs → For alerts and real-time monitoring.
* AWS Athena → To query CloudTrail logs in S3 using SQL.
* AWS GuardDuty → Uses CloudTrail events for threat detection.

**96 \* If we have like 100 servers in that 1-2 server failing how do you find out which one is failing**

The goal here is to quickly identify which EC2 instances (or servers) are unhealthy — and ideally automate that detection using AWS tools.

🧩 1️⃣ Use Amazon CloudWatch (Health & Metrics)

* CloudWatch automatically collects metrics from all EC2 instances.

✅ Steps:

1. Go to CloudWatch → Metrics → EC2 → Per-Instance Metrics
2. Check:
   * CPUUtilization
   * StatusCheckFailed\_Instance
   * StatusCheckFailed\_System

🚨 If you see:

* StatusCheckFailed\_Instance = 1 → OS-level issue
* StatusCheckFailed\_System = 1 → Host hardware or hypervisor issue

You’ll instantly see which instance IDs are failing.

💡 Tip:

You can create a CloudWatch Alarm:

Trigger: EC2 StatusCheckFailed > 0 for 2 consecutive periods

Action: Send notification via SNS or trigger Lambda

🧩 2️⃣ Check EC2 Console → Instance Status Checks

* Go to EC2 Dashboard → Instances → Status Checks tab.
* AWS automatically runs two checks:
  1. System Status Check
  2. Instance Status Check

🔹Healthy instance: Both “2/2 checks passed”

🔹Unhealthy instance: “1/2 checks passed” or “0/2 checks passed”

👉 You can filter in the console:  
Status Check: Failed → instantly lists failing instances.

🧩 3️⃣ Use AWS CLI (for quick command-line detection)

If you have 100 instances, checking manually isn’t practical.  
Use the CLI to find failing ones:

aws ec2 describe-instance-status --region ap-south-1 --filters Name=instance-status.status,Values=impaired --query 'InstanceStatuses[\*].InstanceId'

✅ This will output the Instance IDs that are failing (status = impaired).

You can also check system check failures:

aws ec2 describe-instance-status --filters Name=system-status.status,Values=impaired

🧩 4️⃣ Use CloudWatch Dashboard / Alarms

* Create a CloudWatch Dashboard to visualize all 100 instances.
* Add widgets for:
  + CPU Utilization
  + Network In/Out
  + Status Check Failed

This helps spot failing instances at a glance.

🧩 5️⃣ Integrate with AWS Health Dashboard (Account Level)

* The AWS Health Dashboard shows issues affecting your account’s EC2 hosts or AZs.
* Helps identify if multiple servers failed due to an AWS-side outage.

🧩 6️⃣ Use Auto Scaling Group (ASG) for Auto-Recovery

If your 100 servers are in an ASG:

* The ASG will automatically detect failed instances (via health checks) and replace them.

✅ You can see replaced instances in Auto Scaling → Activity History.

🧩 7️⃣ Enable EC2 Auto Recovery (for standalone instances)

You can enable CloudWatch alarms to automatically recover an impaired EC2:

aws cloudwatch put-metric-alarm \

--alarm-name "Recover-EC2-Instance" \

--metric-name StatusCheckFailed\_System \

--namespace AWS/EC2 \

--statistic Maximum \

--period 60 \

--threshold 1 \

--comparison-operator GreaterThanOrEqualToThreshold \

--dimensions Name=InstanceId,Value=i-1234567890abcdef0 \

--evaluation-periods 2 \

--alarm-actions arn:aws:automate:ap-south-1:ec2:recover

🧩 8️⃣ Optional: Use AWS Systems Manager (SSM)

* If all your servers have SSM Agent, you can run a Fleet Manager view.
* It shows system health, patching status, and connectivity for all instances.
* You can quickly identify failed or unreachable instances.

💬 Example Interview Answer

“If I have 100 servers and 1–2 are failing, I first check CloudWatch metrics and EC2 status checks to identify which instances are impaired.

I also set up CloudWatch alarms that trigger an SNS notification whenever a status check fails.

For automation, I use the AWS CLI to filter all instances with a failed status, and if the servers are part of an Auto Scaling Group, AWS automatically replaces them.

For standalone instances, I configure CloudWatch Auto Recovery.

Additionally, I use Systems Manager Fleet Manager for centralized visibility and patch status across all 100 servers.”

**97 \* Best practices for securing Ur aws**

AWS follows a shared responsibility model:

* AWS secures the infrastructure (data centers, network, hardware).
* You (the customer) secure your configurations, data, and access.

Below are the key layers of AWS security best practices 👇

🧩 1️⃣ Identity and Access Management (IAM)

🔸 Follow the Principle of Least Privilege

* Give only the permissions needed for a specific task.
* Use IAM roles instead of access keys.

🔸 Use IAM Roles over Users

* For EC2, Lambda, ECS → assign roles for temporary credentials.
* Avoid hardcoding credentials in code.

🔸 Enable Multi-Factor Authentication (MFA)

* For root and IAM users.
* Use virtual or hardware MFA devices.

🔸 Use IAM Policies and Groups

* Create groups (e.g., Admin, Developer, ReadOnly).
* Attach policies to groups instead of individual users.

🔸 Rotate Access Keys Regularly

* Disable unused access keys immediately.

🧩 2️⃣ Network Security

🔸 Use VPC Isolation

* Keep public-facing and private workloads in separate subnets.
* Use NAT Gateway for private instances to access the internet securely.

🔸 Configure Security Groups (SGs) and Network ACLs (NACLs)

* Security Groups = stateful (allow only required inbound/outbound).
* NACLs = stateless (layer of subnet-level control).

🔸 Restrict SSH/RDP access

* Use bastion hosts or Session Manager (from Systems Manager) instead of direct SSH.
* Limit SSH/RDP to specific IPs (e.g., office CIDR).

🔸 Enable VPC Flow Logs

* Capture all traffic flow metadata for auditing and analysis.

🧩 3️⃣ Data Protection

🔸 Encrypt Data at Rest

* Use KMS (Key Management Service) for managing encryption keys.
* Enable encryption for:
  + EBS volumes
  + S3 buckets
  + RDS / DynamoDB tables
  + EFS

🔸 Encrypt Data in Transit

* Use HTTPS (TLS/SSL) for all communications.
* Enforce S3 Bucket Policies to allow only HTTPS access.

🔸 Manage Secrets Securely

* Use AWS Secrets Manager or SSM Parameter Store.
* Never store passwords, keys, or tokens in code or EC2 user data.

🧩 4️⃣ Monitoring and Logging

🔸 Enable AWS CloudTrail

* Track every API call in your account.
* Store logs in an S3 bucket with encryption.

🔸 Enable AWS Config

* Continuously audit and record configuration changes.

🔸 Use Amazon CloudWatch

* Set up alarms for unusual activity (e.g., CPU spikes, failed logins).

🔸 Use GuardDuty

* Intelligent threat detection service for continuous security monitoring.

🔸 Use AWS Security Hub

* Centralized dashboard for compliance and security posture.

🧩 5️⃣ S3 Security Best Practices

* Enable Block Public Access at account and bucket level.
* Use Bucket Policies + IAM policies for fine-grained access.
* Enable versioning and MFA Delete.
* Enable Server Access Logging.

🧩 6️⃣ Application Layer Security

* Use AWS WAF (Web Application Firewall) to block SQL injection, XSS, etc.
* Use Shield / Shield Advanced for DDoS protection.
* Use API Gateway authorization for APIs.
* Scan AMIs for vulnerabilities with Inspector or ECR image scanning.

🧩 7️⃣ Account & Governance

* Use AWS Organizations to manage multiple accounts.
* Apply Service Control Policies (SCPs) for restrictions.
* Use Control Tower for account baseline security setup.
* Enable Consolidated Billing + Budgets + Cost Anomaly Detection.

🧩 8️⃣ Backup and Disaster Recovery

* Enable automatic EBS and RDS backups.
* Use S3 versioning + lifecycle policies.
* Test restore procedures periodically.

🧠 Quick Summary Table

| Area | Best Practices | AWS Services |
| --- | --- | --- |
| Identity | MFA, Roles, Least Privilege | IAM, Organizations |
| Network | Private subnets, restricted SGs | VPC, NACL, NAT |
| Data | Encrypt at rest & in transit | KMS, S3, EBS |
| Monitoring | Logging & alerts | CloudTrail, GuardDuty, CloudWatch |
| Application | DDoS & WAF | WAF, Shield |
| Governance | Policies & compliance | Control Tower, Config |
| Backup | DR and recovery | Backup, S3, RDS snapshots |

💬 Example Interview Answer

“To secure my AWS environment, I follow the principle of least privilege using IAM roles and MFA.  
I isolate public and private subnets within a VPC, control inbound/outbound rules using Security Groups and NACLs, and enable VPC Flow Logs.

All data is encrypted at rest using KMS and in transit via HTTPS.  
I enable CloudTrail, Config, and GuardDuty for continuous monitoring and auditing.

I also use S3 block public access, WAF, and Shield for additional protection at the application layer.

For governance, I manage multiple accounts using AWS Organizations and enforce Service Control Policies (SCPs).

These practices ensure a secure, auditable, and compliant AWS setup.”

**98 \* how do you configure ec2 for high availability**

When an interviewer asks this, they want to know:  
➡️ How do you design, deploy, and automate EC2 to stay online even if something fails (like AZ or instance failure).

🧩 1️⃣ Use Multiple Availability Zones (Multi-AZ)

* While launching EC2 manually, you can choose one Availability Zone (AZ).
* But for high availability, you shouldn’t rely on a single AZ.
* So, you use an Auto Scaling Group (ASG) that deploys EC2 instances across at least two AZs within the same region.

✅ Command/Setup:

* In Launch Template / Configuration, specify:
  + subnet-AZ1 and subnet-AZ2
  + desired capacity: e.g., 2 instances (1 per AZ)
* Select Enable load balancing across multiple AZs.

📘 Example:  
1 EC2 in ap-south-1a  
1 EC2 in ap-south-1b

If one AZ fails → traffic shifts to the healthy AZ automatically.

🧩 2️⃣ Attach an Elastic Load Balancer (ELB)

* Add an Application Load Balancer (ALB) or Network Load Balancer (NLB) in front of EC2s.
* It distributes incoming traffic evenly and automatically routes requests only to healthy instances.

✅ Steps:

1. Create an ALB
2. Add Target Group with your EC2 instances or ASG
3. Enable cross-zone load balancing
4. Set health checks (HTTP/HTTPS ping path)

🧩 3️⃣ Configure an Auto Scaling Group (ASG)

* Auto Scaling monitors instance health and automatically replaces unhealthy instances.
* It can also scale out (add more instances) during high load and scale in during low traffic.

✅ Steps:

1. Create a Launch Template (with AMI, instance type, SGs, etc.)
2. Create an ASG using that template.
3. Set:
   * Minimum = 2
   * Desired = 2
   * Maximum = as per load
4. Choose multiple AZs (e.g., 1a, 1b)

🧩 4️⃣ Use Elastic IP (for standalone EC2 setups)

If you only have one EC2 (not using ASG yet):

* Allocate an Elastic IP (EIP) and associate it with the instance.
* If that EC2 fails, reassociate the EIP with a new instance — your app stays accessible using the same IP.

🧩 5️⃣ Store Data in Durable Storage (EBS, S3, RDS)

* Use EBS volumes for persistence (they survive reboots).
* Take EBS snapshots regularly for backup.
* Avoid storing app data directly on the instance store.

🧩 6️⃣ Use Route 53 for Multi-Region or Failover

If you need disaster recovery (DR) across regions:

* Set up a secondary environment in another region.
* Configure Route 53 failover routing with health checks.
  + If primary region goes down → traffic automatically routes to DR region.

🧩 7️⃣ Monitoring & Automation

* Enable CloudWatch alarms on:
  + CPU utilization
  + Status checks
  + Instance termination
* Add alarms to trigger ASG scaling policies or SNS notifications.
* Enable CloudTrail for auditing configuration changes.

🧩 8️⃣ (Optional) Use Placement Groups for Hardware-Level HA

* Use Spread Placement Groups → EC2s are placed on different underlying hardware racks within AZs → prevents single rack failure.

💡 Example Architecture:

High Availability EC2 Setup:

Route 53 (DNS Failover)

↓

Application Load Balancer (Cross-AZ)

↓

Auto Scaling Group

↙ ↘

EC2 in AZ1 EC2 in AZ2

↘ ↙

Amazon EBS / S3 for Data

💬 Example Interview Answer:

“To configure EC2 for high availability, I deploy instances across multiple Availability Zones using an Auto Scaling Group.

The ASG monitors instance health and automatically replaces any failed instance.

I attach an Application Load Balancer in front to distribute traffic evenly and enable cross-zone load balancing.

All logs and data are stored in EBS or S3 for persistence.

For region-level redundancy, I set up Route 53 health checks and failover routing to a secondary region.

Finally, I monitor instance health and scaling metrics using CloudWatch.

This ensures the EC2 environment is self-healing, fault-tolerant, and highly available.”

**99 \* what are options aws provide for highly availiable ec2 if get trouble in one environment**

AWS gives multiple strategies and services to make EC2 highly available — so that even if one instance, Availability Zone (AZ), or Region fails, your application continues to run smoothly.

🧩 1️⃣ Multi–Availability Zone (Multi-AZ) Deployment

Concept:

* Deploy EC2 instances in multiple Availability Zones within the same region.
* If one AZ goes down, traffic is routed to the healthy instance(s) in the other AZ.

Implementation:

* Create an Auto Scaling Group (ASG) that spans 2 or more AZs.
* Attach a Load Balancer (ALB/NLB) to distribute traffic across AZs.

✅ Benefits:

* Automatic failover between AZs
* No single point of failure
* Managed by AWS networking backbone

🧩 2️⃣ Auto Scaling Group (ASG)

Concept:  
Auto Scaling automatically replaces unhealthy instances or adds new ones when traffic increases.

Example:

* If an EC2 instance in one AZ fails, ASG detects it (via health checks) and launches a new instance in a healthy AZ.

✅ Benefits:

* Self-healing
* Load-based scaling
* Cost-optimized

🧩 3️⃣ Elastic Load Balancer (ELB)

Concept:  
Distributes traffic across multiple EC2 instances (and AZs).

Types:

* Application Load Balancer (ALB) → HTTP/HTTPS
* Network Load Balancer (NLB) → TCP/UDP
* Gateway Load Balancer (GWLB) → for appliances like firewalls

✅ Benefits:

* Automatically routes to healthy targets
* Supports cross-zone load balancing
* Integrates with ASG for seamless scaling

🧩 4️⃣ Elastic IP Address (EIP)

Concept:  
A static public IP that can be remapped to another EC2 instance if one fails.

Use Case:

* Quick manual or scripted failover.

✅ Benefit:

* Keeps same public IP even after instance replacement.

🧩 5️⃣ Amazon Machine Image (AMI) Backups

Concept:  
Create AMI snapshots of EC2 periodically.  
In case of failure → Launch new instance from the latest AMI.

✅ Benefit:

* Quick recovery
* Versioned instance templates

🧩 6️⃣ Multi–Region Deployment

Concept:  
If the entire region fails, deploy application in another AWS region.  
Use:

* Route 53 (DNS-based failover)
* Global Accelerator for low latency

✅ Benefit:

* Region-level disaster recovery (DR)
* Zero or minimal downtime

🧩 7️⃣ Route 53 Health Checks and Failover Routing

Concept:  
Monitors endpoint health and routes traffic to healthy regions.

Example:

* Primary Region (ap-south-1)
* Secondary Region (us-east-1)
* If primary fails → Route 53 sends traffic to secondary.

✅ Benefit:

* DNS-based automatic failover
* Global resilience

🧩 8️⃣ Placement Groups (for Specific HA Needs)

* Spread Placement Group:  
  Spreads instances across different racks in multiple AZs → protects from hardware failure.

✅ Use Case:  
High availability & fault isolation.

🧩 9️⃣ Elastic Block Store (EBS) Snapshots & Multi-Attach

* Regular EBS snapshots for recovery.
* EBS Multi-Attach (for io2 volumes) → multiple EC2s can attach same volume (for clustered apps).

✅ Benefit:

* Persistent storage even if instance fails

🧩 🔟 AWS Backup / Disaster Recovery Tools

* AWS Backup for centralized backup.
* AWS Elastic Disaster Recovery (DRS) for failover automation.
* CloudFormation templates to recreate infrastructure quickly.

🧠 Summary Table

| Strategy | Purpose | Key Service |
| --- | --- | --- |
| Multi-AZ | Survive AZ failure | EC2 + ELB + ASG |
| Auto Scaling | Replace failed instances | ASG |
| Load Balancing | Distribute traffic | ALB / NLB |
| Elastic IP | Reassign static IP | EIP |
| AMI Snapshots | Recreate instance quickly | EC2 |
| Multi-Region | Regional disaster recovery | Route 53 + Global Accelerator |
| Health Checks | Automatic failover | Route 53 |
| Spread Placement Group | Hardware-level fault tolerance | EC2 |
| EBS Snapshots | Data protection | EBS |
| AWS Backup / DRS | Full DR solution | AWS Backup / Elastic DR |

💬 Example Interview Answer

“For EC2 high availability, AWS provides multiple mechanisms.

At the infrastructure level, I deploy instances in multiple AZs and attach an Elastic Load Balancer.

I use an Auto Scaling Group so failed instances are automatically replaced.

For regional disasters, I set up Route 53 failover routing to a standby environment in another region.

Additionally, I maintain AMI and EBS snapshots for fast recovery and Elastic IPs for quick remapping.

Together, these ensure fault tolerance, self-healing, and minimal downtime.”

**100\* what are different configuration used for do autoscaling**

Auto Scaling in AWS automatically adjusts the number of EC2 instances (or other resources) based on demand and policies.

There are core components/configurations that define how it works 👇

🧩 1️⃣ Launch Configuration (Older Method)

* Defines how to launch EC2 instances in the Auto Scaling group.
* Specifies:
  + AMI (Amazon Machine Image)
  + Instance type
  + Key pair
  + Security groups
  + Storage (EBS)
  + IAM role
  + User data (startup script)

❌ Note: Launch Configurations are deprecated — replaced by Launch Templates (recommended).

🧩 2️⃣ Launch Template (Modern & Recommended)

* A newer, more flexible replacement for Launch Configurations.
* Can have multiple versions (e.g., v1, v2, etc.).
* Supports:
  + Spot instances + On-demand mix
  + T2/T3 Unlimited options
  + Network interfaces
  + Detailed instance metadata options

✅ Recommended for new deployments.  
✅ Allows parameter overrides in Auto Scaling policies.

🧩 3️⃣ Auto Scaling Group (ASG)

* The main component that controls scaling.
* Defines how many instances to run and where.

Key Settings:

* Min Size → Minimum number of instances
* Desired Capacity → Target number of instances
* Max Size → Maximum number of instances
* VPC/Subnet → Where instances will launch
* Load Balancer / Target Group → For distributing traffic
* Health Checks → EC2 or ELB based

✅ Automatically launches or terminates instances based on scaling policies.

🧩 4️⃣ Scaling Policies

Define when and how Auto Scaling should add or remove instances.

Main Types:

| Policy Type | Description |
| --- | --- |
| Target Tracking | Keeps a metric (e.g., CPUUtilization) at a target value. Most common & easiest. |
| Step Scaling | Adds/removes instances in steps based on metric thresholds. |
| Simple Scaling | Single scale-out or scale-in action after threshold breach (older method). |
| Scheduled Scaling | Scale at specific times (e.g., weekdays at 9AM scale-out). |
| Predictive Scaling | Uses ML to forecast future traffic & scale in advance. |

🧩 5️⃣ CloudWatch Alarms

* Monitor metrics like CPUUtilization, RequestCount, or Memory.
* Trigger scaling actions when thresholds are breached.

Example:

When CPUUtilization > 80% → Scale out by +2 instances  
When CPUUtilization < 30% → Scale in by -1 instance

🧩 6️⃣ Load Balancer Integration

* Usually integrates ALB / NLB / ELB with Auto Scaling Group.
* Ensures:
  + New instances are automatically registered.
  + Unhealthy instances are deregistered.
  + Traffic is evenly distributed.

🧩 7️⃣ Health Checks

Two types:

1. EC2 Health Check → Checks instance status.
2. ELB Health Check → Checks instance response via Load Balancer.

If instance fails → Auto Scaling terminates and replaces it.

🧩 8️⃣ Lifecycle Hooks (Optional but Advanced)

* Let you run custom actions during scaling events.

Example:

Run a script to install dependencies before instance becomes active.  
Archive logs before termination.

🧩 9️⃣ Warm Pools (Optional)

* Pre-initialized EC2 instances kept ready to launch quickly.
* Reduces startup time for scaling events.

✅ Useful for predictable traffic spikes.

🧩 10️⃣ Termination Policies

* Control which instances are terminated first when scaling in.

Examples:

* Oldest instance
* Newest instance
* Closest to next billing hour
* Random

🧠 Summary Table

| Component | Purpose |
| --- | --- |
| Launch Template / Launch Configuration | Defines how to launch EC2s |
| Auto Scaling Group (ASG) | Manages scaling of instances |
| Scaling Policies | Define when/how scaling happens |
| CloudWatch Alarms | Trigger scaling actions |
| Load Balancer | Distributes traffic to instances |
| Health Checks | Replace unhealthy instances |
| Lifecycle Hooks | Custom steps during scale events |
| Warm Pools | Pre-warmed instances for faster scale |
| Termination Policies | Control which instance to remove |

💬 Example Interview Answer

“In Auto Scaling, we use several configurations.

The key ones are the Launch Template (which defines how instances are created) and the Auto Scaling Group, which manages instance count and placement.

We attach scaling policies — like Target Tracking, Step, or Scheduled — which are triggered by CloudWatch alarms.

Integration with a Load Balancer ensures traffic is routed to healthy instances, and Health Checks automatically replace failed ones.

Optionally, we can use Lifecycle Hooks and Warm Pools to customize instance setup and reduce launch time.”

**101\* how do you ensure that multi AZ configuration**

**🧠 What is Multi-AZ (Availability Zone)?**

* **AZ (Availability Zone) = a physically separate data center within an AWS Region.**
* **Multi-AZ configuration means you deploy your infrastructure (like EC2, RDS, ALB, etc.) across multiple AZs to achieve high availability (HA) and fault tolerance.**

**If one AZ goes down (power/network failure), your app still runs from another AZ.**

**✅ Goal**

**Ensure your application and data remain available even if one entire Availability Zone fails.**

**🔹 How to Ensure Multi-AZ Configuration (Service-wise)**

**1️⃣ EC2 (Virtual Machines)**

* **Launch instances in different AZs under the same VPC.**
* **Example:**
  + **Subnet A → us-east-1a**
  + **Subnet B → us-east-1b**
* **Use an Application Load Balancer (ALB) to distribute traffic across both instances.**
* **Attach Auto Scaling Group (ASG) spanning multiple AZs.**

**✅ Ensures:  
Traffic automatically reroutes if one AZ’s instance fails.**

**Diagram:**

**ALB**

**├── EC2 in us-east-1a**

**└── EC2 in us-east-1b**

**2️⃣ RDS (Database Layer)**

* **Choose Multi-AZ deployment while creating the RDS instance.**
* **AWS automatically creates a standby replica in a different AZ and keeps it synchronously replicated.**
* **If the primary DB fails → automatic failover to standby (no data loss).**

**✅ Ensures:  
Database stays available during maintenance or AZ outage.**

**3️⃣ Load Balancer (ALB / NLB)**

* **ALBs and NLBs are by default multi-AZ.**
* **You just need to attach subnets from at least two AZs.**
* **Load balancer routes traffic only to healthy targets across those AZs.**

**✅ Ensures:  
Incoming traffic is balanced and fault-tolerant.**

**4️⃣ Auto Scaling Group (ASG)**

* **Define subnets in two or more AZs in ASG configuration.**
* **If one AZ instance fails, ASG launches new one in another AZ.**

**✅ Ensures:  
Compute layer scales and recovers automatically.**

**5️⃣ S3, DynamoDB, and CloudFront**

* **These services are inherently multi-AZ (and multi-region in some cases).**
* **No extra configuration required.**
* **Data automatically replicated across multiple AZs.**

**✅ Ensures:  
Durability and availability of stored data.**

**6️⃣ EFS (Elastic File System)**

* **EFS automatically stores data across multiple AZs in a region.**
* **Mount it on EC2 instances in different AZs for shared file access.**

**✅ Ensures:  
Redundant, fault-tolerant file storage.**

**7️⃣ VPC Subnets**

* **When designing your VPC:**
  + **Create public and private subnets in at least two AZs.**
  + **Example:**
    - **public-subnet-1 in us-east-1a**
    - **public-subnet-2 in us-east-1b**
    - **private-subnet-1 in us-east-1a**
    - **private-subnet-2 in us-east-1b**

**✅ Ensures:  
Each AZ has both public and private resources for high availability.**

**🔹 Checklist to Ensure Multi-AZ Setup**

| **Layer** | **What to Check** | **Tool/Setting** |
| --- | --- | --- |
| **VPC** | **Subnets created in ≥2 AZs** | **VPC Console** |
| **EC2** | **Instances distributed across AZs** | **EC2 Dashboard** |
| **Load Balancer** | **Subnets from multiple AZs attached** | **ALB Settings** |
| **ASG** | **Multiple AZs in group config** | **Auto Scaling Group** |
| **RDS** | **Multi-AZ deployment enabled** | **RDS Settings** |
| **EFS/S3** | **Multi-AZ replication (automatic)** | **Built-in** |
| **Route 53** | **Health checks and failover routing** | **DNS Level** |

**✅ In short**

**You ensure Multi-AZ configuration by deploying your resources (compute, database, load balancer, and subnets) across at least two Availability Zones and using AWS-managed replication or failover features.**

**102\*** **how can we connect s3 through vpc privetly**

**To connect Amazon S3 privately (without going through the internet), we use an S3 VPC Endpoint — specifically an Interface Endpoint or Gateway Endpoint.**

**🧱 1. Use a VPC Endpoint for S3**

**AWS provides VPC Endpoints that let your EC2, Lambda, or ECS resources access S3 privately — traffic stays within the AWS network and never goes through the public internet.**

**💡 Types of VPC Endpoints**

| **Type** | **Used For** | **Example** | **Notes** |
| --- | --- | --- | --- |
| **Gateway Endpoint** | **For AWS services like S3 and DynamoDB** | **com.amazonaws.<region>.s3** | **Free, simple, preferred for S3** |
| **Interface Endpoint (PrivateLink)** | **For most other AWS services** | **com.amazonaws.<region>.s3** | **Uses private IPs (ENIs) in subnets, costs per hour/data** |

**✅ For S3, we generally use a Gateway Endpoint.**

**⚙️ 2. Steps to Create a Gateway Endpoint for S3**

**1️⃣ Go to VPC Console → Endpoints → Create endpoint  
2️⃣ Choose service:**

* **Service category: AWS services**
* **Service name: com.amazonaws.<region>.s3  
  3️⃣ Select your VPC  
  4️⃣ Select Route Tables (associate with subnets that need S3 access)  
  5️⃣ Add Policy (example below)  
  6️⃣ Click Create endpoint**

**🪪 3. Example Policy (Restrict to Specific Bucket)**

**{**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Principal": "\*",**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-private-bucket",**

**"arn:aws:s3:::my-private-bucket/\*"**

**]**

**}**

**]**

**}**

**This ensures only your bucket is accessible through the endpoint.**

**🔐 4. Modify S3 Bucket Policy (for Endpoint access only)**

**Attach this to your S3 bucket policy:**

**{**

**"Effect": "Deny",**

**"Principal": "\*",**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-private-bucket",**

**"arn:aws:s3:::my-private-bucket/\*"**

**],**

**"Condition": {**

**"StringNotEquals": {**

**"aws:sourceVpce": "vpce-0a1b2c3d4e5f6g7h8"**

**}**

**}**

**}**

**➡️ This means:  
Access to the bucket is only allowed if traffic comes through the VPC endpoint.**

**🧠 5. Verification**

**From an EC2 inside that VPC (without internet access), try:**

**aws s3 ls s3://my-private-bucket**

**✅ You’ll be able to access it privately — no public internet route is used.**

**🗣️ Short Interview Summary:**

**To connect to S3 privately from a VPC, I create a VPC Gateway Endpoint for S3, associate it with my route table, and update the S3 bucket policy to allow access only through that endpoint.  
This keeps all S3 traffic inside AWS’s private network — no internet exposure.**

**103\* how can we execute round robin waited routing of route53**

**Route 53 Weighted Routing Policy lets you distribute traffic across multiple resources (like EC2 instances, load balancers, or IPs) based on assigned weights.**

**It’s similar to a round-robin load balancing, but with control over traffic percentage to each target.**

**🔹 How It Works**

* **You create multiple DNS records with the same name (e.g., app.example.com) but different IPs or endpoints.**
* **Assign a weight value to each record (e.g., 70, 30).**
* **When a user queries DNS, Route 53 randomly returns IPs proportionally to their weight values.**

**✅ Example:**

| **Record** | **IP Address** | **Weight** | **Traffic %** |
| --- | --- | --- | --- |
| **A record** | **10.0.0.1** | **70** | **70%** |
| **A record** | **10.0.0.2** | **30** | **30%** |

**So, roughly 70% of DNS queries go to 10.0.0.1 and 30% go to 10.0.0.2.**

**🔹 Steps to Configure Weighted Routing (Round Robin)**

**Step 1: Open Route 53 Console**

**Go to Route 53 → Hosted Zones → your domain (example.com).**

**Step 2: Create the First Record**

* **Record name: app.example.com**
* **Record type: A / AAAA / CNAME**
* **Value: IP or endpoint (e.g., ALB or EC2 public IP)**
* **Routing policy: Weighted**
* **Weight: e.g., 70**
* **Set ID: Primary-Server**

**Step 3: Create the Second Record**

* **Same record name: app.example.com**
* **Type: Same (A or CNAME)**
* **Value: Another endpoint (e.g., 10.0.0.2)**
* **Routing policy: Weighted**
* **Weight: e.g., 30**
* **Set ID: Secondary-Server**

**Step 4: Optional – Health Check**

**Attach a Route 53 health check to each record so that:**

* **If one endpoint fails → Route 53 stops routing traffic to it.**

**This makes the setup both weighted and highly available.**

**Step 5: Save the Records**

**Now, when users resolve app.example.com, Route 53 distributes responses based on weight (like round robin, but controlled).**

**🔹 Round Robin vs Weighted Routing**

| **Feature** | **Round Robin** | **Weighted Routing** |
| --- | --- | --- |
| **Equal traffic** | **Yes** | **Not necessarily** |
| **Control over %** | **No** | **Yes** |
| **Use case** | **Basic load balancing** | **Gradual traffic shift / A/B testing / Blue-Green deployment** |

**🔹 Use Cases**

**✅ A/B testing → Send 80% users to v1, 20% to v2  
✅ Blue-Green Deployment → Gradually move traffic to new version  
✅ Multi-region traffic → Control which region gets more traffic**

**✅ Example Interview Answer**

**“We can achieve round-robin weighted routing in Route 53 by creating multiple records with the same domain name and assigning different weights to each.  
For example, if I have two EC2 instances, I can assign weights 70 and 30 so that roughly 70% of traffic goes to one and 30% to the other.  
Route 53 will randomly serve IPs according to these weights, effectively achieving controlled round-robin routing.  
We can also add health checks to make it fault-tolerant.”**

**104\* how can you search for particular error in K8s cloudwatch**

**To search for a specific error (e.g., "error", "timeout", "failed"), you can use CloudWatch Logs Insights — it allows you to query and filter logs efficiently.**

**1️⃣ Prerequisites**

* **Your EKS cluster must be sending logs to CloudWatch via:**
  + **CloudWatch agent, or**
  + **Fluent Bit / Fluentd / CloudWatch Container Insights.**

**Logs are grouped under:**

**/aws/containerinsights/<cluster-name>/application**

**/aws/containerinsights/<cluster-name>/dataplane**

**/aws/containerinsights/<cluster-name>/host**

**2️⃣ Steps to Search for Errors**

**🔹 Step 1: Go to CloudWatch Console**

* **Open AWS Console → CloudWatch → Logs Insights.**

**🔹 Step 2: Select Log Group**

* **Example: /aws/containerinsights/my-cluster/application**

**🔹 Step 3: Run a Query**

**Example query to search for “error” in application logs:**

**fields @timestamp, @message**

**| filter @message like /error/i**

**| sort @timestamp desc**

**| limit 20**

**👉 Explanation:**

* **@message like /error/i → searches for “error” (case-insensitive).**
* **limit 20 → shows 20 most recent error logs.**

**3️⃣ Other Useful Queries**

**🔹 Search for Pod name and error**

**fields @timestamp, kubernetes.pod\_name, @message**

**| filter @message like /Exception|Error|Failed/**

**| sort @timestamp desc**

**| limit 50**

**🔹 Search for specific namespace**

**fields @timestamp, kubernetes.namespace\_name, kubernetes.pod\_name, @message**

**| filter kubernetes.namespace\_name = "production"**

**| filter @message like /timeout/**

**| sort @timestamp desc**

**4️⃣ Best Practices**

* **Use Fluent Bit with EKS to ship logs automatically to CloudWatch.**
* **Create CloudWatch Alarms to alert on recurring errors.**
* **Use CloudWatch dashboards or Grafana for visual monitoring.**
* **Tag logs with namespace, pod, and container for easy filtering.**

**5️⃣ Short Interview Answer Example**

**“In EKS, application logs are sent to CloudWatch via Fluent Bit or Container Insights.  
I open CloudWatch Logs Insights, select the log group for my cluster, and run a query like filter @message like /error/i to find specific errors. I can also filter by pod or namespace to narrow down the issue.”**

**105\* Apln load balancer :**

**Application Load Balancer (ALB) is a Layer 7 (Application Layer) load balancer provided by AWS Elastic Load Balancing (ELB) service.**

**It distributes incoming HTTP and HTTPS traffic across multiple targets like:**

* **EC2 instances**
* **Containers (ECS / EKS)**
* **Lambda functions**
* **IP addresses**

**⚙️ Key Features of ALB**

| **Feature** | **Description** |
| --- | --- |
| **Layer 7 (Application Layer)** | **Works at HTTP/HTTPS level — understands URL, headers, cookies, etc.** |
| **Path-based Routing** | **Route traffic based on URL path (e.g., /api → one target group, /images → another).** |
| **Host-based Routing** | **Route based on domain name (e.g., app.example.com vs api.example.com).** |
| **SSL Termination (HTTPS)** | **ALB can handle SSL/TLS certificates (using AWS Certificate Manager).** |
| **WebSocket & HTTP/2 Support** | **Supports modern web protocols.** |
| **Health Checks** | **Monitors target health — routes traffic only to healthy targets.** |
| **Integration with Auto Scaling** | **Works with EC2 Auto Scaling groups for dynamic capacity.** |
| **Redirects and Fixed Responses** | **Can redirect (e.g., HTTP → HTTPS) or return custom responses.** |
| **WAF Integration** | **Integrates with AWS WAF for security (block malicious traffic).** |

**🧩 ALB Architecture Example**

**Internet**

**│**

**┌───────▼────────┐**

**│ Application │**

**│ Load Balancer │**

**└───────▲────────┘**

**│**

**┌─────────┴─────────┐**

**│ │**

**EC2 Instance A EC2 Instance B**

**(Target Group 1) (Target Group 2)**

* **ALB receives client requests.**
* **Checks routing rules (path/host).**
* **Forwards requests to appropriate Target Group (which contains instances, ECS tasks, or Lambdas).**
* **Performs health checks continuously.**

**🔐 Listeners and Target Groups**

* **Listener: Listens for connection requests (e.g., port 80 for HTTP or 443 for HTTPS).**
* **Target Group: A group of targets (instances, containers, or IPs) to which ALB routes traffic.**

**Example:**

* **Listener on port 443 (HTTPS)**
* **Path /api/\* → Target Group 1 (ECS)**
* **Path /images/\* → Target Group 2 (S3 or EC2)**

**📈 Advantages**

**✅ High availability (multi-AZ)  
✅ Scales automatically  
✅ Smart routing (Layer 7)  
✅ Secure with HTTPS + WAF  
✅ Works well with containers and microservices**

**⚡ Difference between ALB and NLB**

| **Feature** | **Application LB (ALB)** | **Network LB (NLB)** |
| --- | --- | --- |
| **Layer** | **7 (Application)** | **4 (Transport - TCP/UDP)** |
| **Protocols** | **HTTP, HTTPS** | **TCP, UDP, TLS** |
| **Routing** | **Based on URL/Host** | **Based on IP and Port** |
| **Use Case** | **Web apps, microservices** | **High-performance or low-latency apps** |
| **Example** | **Website traffic** | **Gaming, IoT, databases** |

**✅ Example Use Case**

**You host an e-commerce app:**

* **/ → web frontend (EC2)**
* **/api → backend (ECS service)**
* **/images → static files (S3 or CloudFront)**

**All managed via one ALB.**

**106\* User data ,meta data**

**🔧 Definition:**

**User Data is a script or configuration that you provide to an EC2 instance at launch time.  
It runs automatically when the instance first boots.**

**Used to automate instance setup — like installing packages, updating software, or configuring apps.**

**🧠 Example:**

**You can add a shell script when launching an EC2 instance (under “Advanced Details → User data”):**

**#!/bin/bash**

**yum update -y**

**yum install -y httpd**

**systemctl start httpd**

**systemctl enable httpd**

**echo "<h1>Hello from EC2 User Data</h1>" > /var/www/html/index.html**

**💡 This script will automatically:**

* **Update the OS**
* **Install Apache**
* **Start the web server**
* **Create a sample web page**

**✅ Purpose: Automate server provisioning (no need to manually SSH).**

**🧩 2️⃣ EC2 Metadata**

**🔧 Definition:**

**Metadata is information about the running EC2 instance.  
It provides details like instance ID, IP address, region, AMI ID, etc.**

**You can access it from inside the EC2 instance using a special URL:**

**curl http://169.254.169.254/latest/meta-data/**

**Example output:**

**ami-id**

**instance-id**

**local-ipv4**

**public-ipv4**

**security-groups**

**To view a specific value:**

**curl http://169.254.169.254/latest/meta-data/instance-id**

**👉 Returns something like:**

**i-0abc1234def56789**

**🧾 Bonus: Instance Identity Document (IID)**

**You can also access instance identity metadata for verification:**

**curl http://169.254.169.254/latest/dynamic/instance-identity/document**

**It gives JSON output with:**

* **instanceId**
* **region**
* **accountId**
* **availabilityZone**

**✅ Summary Table**

| **Feature** | **Description** | **Access Method** | **Example Use** |
| --- | --- | --- | --- |
| **User Data** | **Script runs on first boot** | **Added at launch** | **Install software, configure app** |
| **Metadata** | **Info about the instance** | **curl 169.254.169.254** | **Get instance ID, IP, AZ** |
| **Runs automatically?** | **✅ Yes (on first boot)** | **❌ No (you query manually)** | **—** |

**107\* Listener target**

**🔹 Listener and Target in AWS ELB**

**1️⃣ Listener**

**A Listener is a process that checks for connection requests (like HTTP, HTTPS, or TCP) from clients using the configured protocol and port.**

* **It acts as the entry point for the Load Balancer.**
* **Each listener has:**
  + **Protocol (e.g., HTTP, HTTPS, TCP)**
  + **Port (e.g., 80, 443)**
  + **Rules that define where to forward traffic (to target groups)**

**🧩 Example:**

| **Component** | **Example** |
| --- | --- |
| **Protocol** | **HTTP** |
| **Port** | **80** |
| **Default Action** | **Forward to target group “web-app-tg”** |

**So when traffic hits the load balancer on port 80, the listener forwards it to the defined target group.**

**2️⃣ Target (or Target Group)**

**A Target is the actual destination resource (like EC2 instances, Lambda functions, IPs, or containers) where the load balancer sends requests.**

* **Targets are grouped into Target Groups.**
* **Each target group has:**
  + **Registered targets (EC2 instances, ECS tasks, IPs)**
  + **Health check configuration**
* **The listener forwards requests to the target group based on rules.**

**🧩 Example:**

| **Target Group** | **Protocol** | **Port** | **Registered Targets** | **Health Check Path** |
| --- | --- | --- | --- | --- |
| **web-app-tg** | **HTTP** | **80** | **i-0abc123, i-0xyz456** | **/health** |

**3️⃣ How They Work Together**

**Flow:**

1. **Client → Load Balancer (Listener on port 80)**
2. **Listener rule → Forward to Target Group (e.g., web-app-tg)**
3. **Target Group → Routes request to healthy instance (Target)**
4. **Instance responds → Load Balancer → Client**

**4️⃣ Example with Application Load Balancer (ALB)**

* **Listener:**
  + **HTTP:80**
  + **HTTPS:443**
* **Rules:**
  + **If URL path is /api/\* → Forward to api-target-group**
  + **Else → Forward to web-target-group**

**✅ In summary:**

| **Concept** | **Definition** | **Example** |
| --- | --- | --- |
| **Listener** | **Listens for incoming traffic on a specific protocol/port** | **HTTP:80** |
| **Target Group** | **Group of resources where traffic is sent** | **web-app-tg** |
| **Target** | **The actual instance/IP/Lambda receiving the traffic** | **EC2 instance i-0abc123** |

**109\* Application load balancer , classic network; disable ,pricing**

**🔹 1️⃣ Types of AWS Load Balancers**

| **Load Balancer** | **Layer** | **Use Case** | **Protocols Supported** |
| --- | --- | --- | --- |
| **Application Load Balancer (ALB)** | **Layer 7 (Application Layer)** | **HTTP/HTTPS traffic, web apps, microservices** | **HTTP, HTTPS, WebSocket** |
| **Network Load Balancer (NLB)** | **Layer 4 (Transport Layer)** | **High-performance, TCP/UDP traffic, gaming, IoT, or VPNs** | **TCP, UDP, TLS** |
| **Classic Load Balancer (CLB)** | **Layer 4 & 7 (Legacy)** | **Older apps, EC2-Classic (deprecated)** | **HTTP, HTTPS, TCP, SSL** |

**🔸 2️⃣ Application Load Balancer (ALB)**

**✅ Features**

* **Operates at Layer 7 (Application).**
* **Supports path-based and host-based routing.**
* **Supports target groups (EC2, ECS, Lambda, IP targets).**
* **Supports sticky sessions, WebSocket, and redirects.**
* **Integrated with AWS WAF, CloudWatch, and Auto Scaling.**
* **Can route traffic to multiple target groups (great for microservices).**

**⚙️ When to use**

* **Web applications, APIs, microservices.**
* **If you need routing based on URL path or hostname.  
  Example:**
  + **/api/\* → backend API target group**
  + **/app/\* → frontend target group**

**🔸 3️⃣ Network Load Balancer (NLB)**

**✅ Features**

* **Operates at Layer 4 (Transport).**
* **Handles millions of requests per second with very low latency.**
* **Best for TCP/UDP traffic (e.g., gaming, real-time, IoT).**
* **Can assign static IPs and Elastic IPs.**
* **Supports TLS termination and health checks.**
* **Works well with PrivateLink and VPC endpoints.**

**⚙️ When to use**

* **High-performance or low-latency apps.**
* **Applications using non-HTTP protocols.**
* **Need static IPs for clients.**

**🔸 4️⃣ Classic Load Balancer (CLB)**

**⚠️ Status: Legacy — being replaced by ALB/NLB.**

**✅ Features**

* **Operates at both Layer 4 & 7.**
* **Basic load balancing for HTTP, HTTPS, TCP.**
* **No support for modern routing (like ALB’s path-based).**
* **Uses single target group (no multiple groups).**

**⚙️ When to use**

* **Only for legacy apps or EC2-Classic networks (no longer recommended).**

**🔹 5️⃣ How to Disable / Delete Load Balancer**

**If you no longer need a load balancer:**

1. **Go to EC2 Console → Load Balancers.**
2. **Select your ALB / NLB / CLB.**
3. **Choose Actions → Delete.**
4. **Confirm deletion — resources (listeners, target groups) will be released.**

**⚠️ Note: Deleting the load balancer does not delete your target instances.**

**🔹 6️⃣ Pricing (Approximate)**

| **Type** | **Fixed Hourly Cost** | **Data Processing Cost** | **Example** |
| --- | --- | --- | --- |
| **ALB** | **~$0.025 per hour** | **~$0.008 per LCU-hour** | **Pay for hours + LCUs (new connections, active connections, rule evaluations)** |
| **NLB** | **~$0.0225 per hour** | **~$0.006 per LCU-hour** | **Pay for hours + LCUs (data processed, active flows)** |
| **CLB** | **~$0.025 per hour** | **~$0.008 per GB processed** | **Older, less cost-efficient** |

**💡 LCU = Load Balancer Capacity Unit (based on active connections, new connections, processed bytes, and rules).**

**🔹 7️⃣ Summary Comparison**

| **Feature** | **ALB** | **NLB** | **CLB** |
| --- | --- | --- | --- |
| **OSI Layer** | **Layer 7** | **Layer 4** | **Layer 4 & 7** |
| **Protocols** | **HTTP, HTTPS** | **TCP, UDP, TLS** | **HTTP, HTTPS, TCP** |
| **Routing** | **Path/Host-based** | **IP/Port-based** | **Basic** |
| **Performance** | **High** | **Extreme** | **Moderate** |
| **Static IP** | **❌** | **✅** | **❌** |
| **WAF Support** | **✅** | **❌** | **❌** |
| **Lambda Target** | **✅** | **✅** | **❌** |
| **Status** | **Modern** | **Modern** | **Legacy** |
| **Best For** | **Web apps / APIs** | **Low-latency / high traffic** | **Legacy EC2-Classic apps** |

**✅ In short**

* **ALB: For HTTP/HTTPS apps — intelligent routing at Layer 7.**
* **NLB: For TCP/UDP — ultra-fast performance at Layer 4.**
* **CLB: Legacy — not recommended for new deployments.**
* **To disable: Delete from EC2 Console → Load Balancers.**
* **Pricing: Charged per hour + per LCU (usage-based).**

**110\* Bucket policy , access control**

**✅ 1. Bucket Policy (Resource-based Policy)**

**Definition:**

**A Bucket Policy is a JSON-based resource policy that you attach directly to an S3 bucket to control who can access the bucket and what actions they can perform.**

**💡 Key Points:**

* **Applied at the bucket level**
* **Supports fine-grained permissions (like IAM policies)**
* **Written in JSON**
* **Can allow or deny access to:**
  + **Specific AWS accounts, IAM users, or roles**
  + **Specific IP ranges or VPC endpoints**

**🧱 Example: Public Read Bucket Policy**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Sid": "AllowPublicRead",**

**"Effect": "Allow",**

**"Principal": "\*",**

**"Action": "s3:GetObject",**

**"Resource": "arn:aws:s3:::my-bucket-name/\*"**

**}**

**]**

**}**

**✅ Anyone can read objects from the bucket (not recommended for private data).**

**🧱 Example: Allow Access Only from a Specific IAM Role**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Principal": {**

**"AWS": "arn:aws:iam::123456789012:role/AppServerRole"**

**},**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-private-bucket",**

**"arn:aws:s3:::my-private-bucket/\*"**

**]**

**}**

**]**

**}**

**🧱 Example: Allow Access Only via VPC Endpoint**

**{**

**"Effect": "Deny",**

**"Principal": "\*",**

**"Action": "s3:\*",**

**"Resource": "arn:aws:s3:::my-private-bucket/\*",**

**"Condition": {**

**"StringNotEquals": {**

**"aws:sourceVpce": "vpce-0a1b2c3d4e5f6g7h8"**

**}**

**}**

**}**

**✅ 2. Access Control List (ACL)**

**Definition:**

**An Access Control List (ACL) is a legacy permission mechanism in S3 that grants read/write permissions at the bucket or object level to other AWS accounts or predefined groups.**

**💡 Key Points:**

* **Works at object level and bucket level**
* **Grants basic permissions like:**
  + **READ, WRITE, READ\_ACP, WRITE\_ACP**
* **Less flexible and granular than bucket policies**
* **AWS recommends using bucket policies or IAM policies instead of ACLs**

**🧱 Example: Grant Read Access to Another AWS Account via ACL**

**aws s3api put-bucket-acl \**

**--bucket my-bucket \**

**--grant-read "id=CanonicalUserID\_of\_AnotherAccount"**

**⚖️ Bucket Policy vs Access Control List**

| **Feature** | **Bucket Policy** | **ACL** |
| --- | --- | --- |
| **Type** | **Resource-based JSON policy** | **Legacy permission system** |
| **Granularity** | **Very fine-grained** | **Limited (read/write)** |
| **Scope** | **Bucket + objects** | **Bucket + objects** |
| **Cross-account support** | **Yes** | **Yes (basic)** |
| **Recommended** | **✅ Yes** | **🚫 Legacy (use only if required)** |

**🗣️ Short Interview Answer:**

**Bucket Policy is a JSON-based resource policy attached to an S3 bucket to control access (who can do what).  
Access Control List (ACL) is an older method that gives simple read/write permissions to other AWS accounts.  
Nowadays, AWS recommends Bucket Policies and IAM policies over ACLs for better security and flexibility.**

**111\* Autoscaling ,cloud watch alarm ,;predictive**

**Auto Scaling in AWS automatically adds or removes EC2 instances based on demand to maintain performance and reduce cost.**

**It ensures:**

* **You always have the right number of instances**
* **You don’t overpay for idle resources**

**✅ How Auto Scaling Works**

**Auto Scaling uses three key components:**

| **Component** | **Purpose** |
| --- | --- |
| **Launch Template / Launch Configuration** | **Defines how new instances are created (AMI, type, security groups, etc.)** |
| **Auto Scaling Group (ASG)** | **Defines min, max, and desired number of instances** |
| **Scaling Policies** | **Define *when* to scale out (add) or scale in (remove) instances** |

**🔹 2️⃣ What is a CloudWatch Alarm in Auto Scaling?**

**CloudWatch Alarms are used to trigger scaling actions based on metrics.**

**For example:**

* **If CPU utilization > 70% for 5 minutes → scale out (add instance)**
* **If CPU utilization < 30% for 10 minutes → scale in (remove instance)**

**✅ Steps to Configure**

1. **Go to CloudWatch → Alarms → Create Alarm**
2. **Choose a metric like EC2 > Per-Instance Metrics > CPUUtilization**
3. **Set threshold (e.g., >70%)**
4. **In the Actions section, select Auto Scaling Policy**
5. **Choose your ASG and attach a scaling policy (scale out/in)**

**Now CloudWatch continuously monitors your metric and triggers scaling events automatically.**

**🔹 3️⃣ What is Predictive Auto Scaling?**

**Predictive Scaling uses machine learning to forecast traffic patterns and scale proactively — *before* a traffic spike happens.**

**Instead of waiting for CPU to rise, AWS analyzes historical data (like daily or weekly usage patterns) and adds instances ahead of time.**

**✅ Example:  
If your website traffic spikes every morning at 9 AM,  
Predictive Scaling will automatically launch instances at 8:55 AM, so your app stays fast and stable.**

**✅ How to Enable Predictive Scaling**

1. **Open EC2 → Auto Scaling Groups**
2. **Select your ASG → Choose Automatic Scaling → Add Policy**
3. **Choose Predictive scaling policy**
4. **Define:**
   * **Metric type (CPU, network, custom)**
   * **Forecasting period (1 day, 1 week, etc.)**
5. **Save — AWS will analyze historical data (needs ~24 hours to start forecasting).**

**🔹 4️⃣ Types of Scaling Policies**

| **Type** | **Description** |
| --- | --- |
| **Target Tracking** | **Keeps a specific metric (like CPU at 50%) automatically balanced** |
| **Step Scaling** | **Scales by steps (e.g., add 1 instance if >70%, add 2 if >90%)** |
| **Simple Scaling** | **Add/remove fixed number of instances when threshold is crossed** |
| **Predictive Scaling** | **Uses ML to forecast and scale in advance** |

**🔹 5️⃣ Example Interview Answer**

**“Auto Scaling automatically adjusts the number of EC2 instances based on demand.  
We configure CloudWatch Alarms on metrics like CPU utilization to trigger scaling policies — for example, scaling out when CPU exceeds 70%.**

**We also use Predictive Auto Scaling, which uses historical trends to forecast traffic and pre-launch instances before peak times.  
This ensures optimal performance and avoids latency during high-load periods while keeping costs efficient.”**

**112\* Cool down period (waits)**

**The cooldown period is the time Auto Scaling waits after launching or terminating an instance before starting another scaling activity.**

**This helps prevent rapid, repeated scaling (flapping).**

**Example:**

**Let’s say:**

* **You set a cooldown period of 300 seconds (5 minutes).**
* **A scaling policy triggers and launches 2 new EC2 instances.**
* **The Auto Scaling group will wait 5 minutes before evaluating further scaling triggers.**

**This gives the new instances time to start, stabilize, and handle load.**

**Key Points**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | **Prevents multiple scaling events from happening too close together** |
| **Default cooldown** | **300 seconds (5 minutes)** |
| **Type** | **There are default and per-policy cooldowns** |
| **Scope** | **Applies to entire Auto Scaling group or individual scaling policies** |
| **Effect** | **During cooldown, CloudWatch alarms are ignored for that Auto Scaling group** |

**Per-Policy Cooldown**

**You can define a different cooldown for each scaling policy if you need more granular control.  
Example: CPU-based scale-out may wait 180 seconds, while scale-in waits 600 seconds.**

**Real-World Example**

**If CPU goes above 70%, scale-out adds 2 instances.  
You don’t want it to keep adding more instantly, so you give 5 minutes for CPU load to balance across the new instances.  
That’s where cooldown helps.**

**Interview Short Answer:**

**“A cooldown period is the waiting time after an Auto Scaling activity during which no new scaling actions occur. It allows the new instances to stabilize before another scaling event is triggered, preventing unnecessary scaling.”**

**113\* Default encryption algo is base 64**

* **Base64 is an *encoding* method, not encryption.**
* **It is used to convert binary data into text (readable ASCII format).**
* **Commonly used for:**
  + **Sending binary files (images, certificates) via text protocols (like email, JSON)**
  + **Storing credentials in configuration files (e.g., Basic Auth headers)**
  + **Encoding data safely for transmission — not for security**

**🧩 Example:  
If you encode Hello in Base64 → it becomes SGVsbG8=  
Anyone can decode it back easily — so it’s not secure.**

**🔐 Encryption Algorithms (Actual Security)**

**Encryption algorithms protect data by making it unreadable without a key.  
Examples of encryption, not encoding:**

| **Algorithm** | **Type** | **Used for** |
| --- | --- | --- |
| **AES (Advanced Encryption Standard)** | **Symmetric** | **S3, EBS, RDS encryption** |
| **RSA** | **Asymmetric** | **Key exchange, SSL/TLS** |
| **SHA (Secure Hash Algorithm)** | **Hashing** | **Integrity check (not reversible)** |
| **KMS (Key Management Service)** | **AWS-managed encryption** | **Encrypts AWS resources using AES-256** |

**🧠 In AWS**

**The default encryption algorithm used by AWS services like S3, EBS, RDS is:**

**AES-256 (Advanced Encryption Standard, 256-bit key)**

**So:**

**🔸 Base64 = Encoding (not secure)  
🔸 AES-256 = Encryption (secure)**

**✅ In short:**

| **Concept** | **Purpose** | **Secure?** |
| --- | --- | --- |
| **Base64** | **Encoding for data transfer** | **❌ No** |
| **AES-256** | **Encryption for data protection** | **✅ Yes** |

**114\* Ingress ,loadbalancer**

**A Load Balancer is a service or device that distributes incoming network traffic across multiple targets (servers, containers, or pods).  
It ensures high availability, scalability, and fault tolerance.**

**🔧 In AWS:**

**You have Elastic Load Balancing (ELB) types:**

* **Application Load Balancer (ALB) – Layer 7 (HTTP/HTTPS)**
* **Network Load Balancer (NLB) – Layer 4 (TCP/UDP)**
* **Gateway Load Balancer (GLB) – Layer 3**
* **Classic Load Balancer (legacy)**

**These AWS load balancers sit outside your Kubernetes cluster or EC2 servers.**

**✅ Example:**

**You have 3 EC2 instances running a web app.  
An ALB receives requests and distributes them evenly:**

**User → ALB → EC2 instance 1, 2, 3**

**🔹 2️⃣ What is Ingress (in Kubernetes)?**

**An Ingress is a Kubernetes resource that manages external access to services inside a cluster — typically HTTP and HTTPS traffic.**

**Ingress acts like a smart Layer 7 router.**

**🔧 In simple terms:**

**Ingress = a set of routing rules inside Kubernetes that define how external traffic reaches internal services.**

**✅ Example YAML:**

**apiVersion: networking.k8s.io/v1**

**kind: Ingress**

**metadata:**

**name: myapp-ingress**

**spec:**

**rules:**

**- host: myapp.example.com**

**http:**

**paths:**

**- path: /**

**pathType: Prefix**

**backend:**

**service:**

**name: myapp-service**

**port:**

**number: 80**

**This means:**

**Traffic → Ingress Controller → myapp-service → pods**

**🔹 3️⃣ Relationship between Ingress and Load Balancer**

**In AWS (or any cloud):**

* **The Ingress Controller (e.g., AWS ALB Ingress Controller or NGINX Ingress Controller) usually uses a Load Balancer behind the scenes.**
* **The Load Balancer handles the external network traffic.**
* **The Ingress defines the routing rules inside the cluster.**

**✅ Example Architecture:**

**Internet**

**↓**

**AWS ALB (Load Balancer)**

**↓**

**Ingress Controller (inside Kubernetes)**

**↓**

**Kubernetes Services**

**↓**

**Pods (Application)**

**✅ Summary Table**

| **Feature** | **Ingress** | **Load Balancer** |
| --- | --- | --- |
| **Scope** | **Kubernetes (cluster level)** | **AWS / network level** |
| **Layer** | **Layer 7 (HTTP/HTTPS)** | **L4 or L7** |
| **Purpose** | **Route traffic to services inside cluster** | **Distribute traffic across servers** |
| **Managed by** | **Kubernetes API + Ingress Controller** | **AWS ELB service** |
| **Example Tool** | **AWS ALB Ingress Controller, NGINX Ingress** | **AWS ALB, NLB, CLB** |

**In short:**

**Ingress = Rules inside Kubernetes for routing  
Load Balancer = AWS service that distributes traffic to servers/pods**

**115\* Stateful sets ,static ip**

**A StatefulSet is a Kubernetes workload controller used to manage stateful applications, like databases or systems that need persistent identity and stable storage.**

**Unlike Deployments (used for stateless apps), StatefulSets maintain:**

* **Stable network identity**
* **Stable persistent storage**
* **Ordered, graceful deployment and scaling**

**🧩 Example use cases:**

* **Databases → MySQL, PostgreSQL, Cassandra**
* **Message queues → Kafka, RabbitMQ**
* **Systems needing fixed names or network identities**

**🔹 2️⃣ Key Features of StatefulSets**

| **Feature** | **Description** |
| --- | --- |
| **Stable Pod Name** | **Each pod gets a unique, predictable name like mysql-0, mysql-1, mysql-2.** |
| **Stable Storage** | **Each pod gets its own persistent volume claim (PVC) that stays even if the pod is rescheduled.** |
| **Ordered Deployment & Termination** | **Pods are created and deleted in order (mysql-0 before mysql-1).** |
| **Stable Network Identity** | **Each pod gets a stable DNS name that does not change.** |

**🔹 3️⃣ Static (Stable) IP in StatefulSets**

**StatefulSets achieve static networking using a Headless Service.**

**⚙️ Headless Service**

**When you create a Headless Service (by setting clusterIP: None), Kubernetes assigns DNS records for each pod.**

**Example:**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: mysql**

**spec:**

**clusterIP: None**

**selector:**

**app: mysql**

**ports:**

**- port: 3306**

**This gives each pod a stable DNS name like:**

**mysql-0.mysql.default.svc.cluster.local**

**mysql-1.mysql.default.svc.cluster.local**

**mysql-2.mysql.default.svc.cluster.local**

**Each pod gets a static network identity — even if rescheduled on another node.**

**🔹 4️⃣ Example StatefulSet YAML**

**apiVersion: apps/v1**

**kind: StatefulSet**

**metadata:**

**name: mysql**

**spec:**

**serviceName: "mysql"**

**replicas: 3**

**selector:**

**matchLabels:**

**app: mysql**

**template:**

**metadata:**

**labels:**

**app: mysql**

**spec:**

**containers:**

**- name: mysql**

**image: mysql:8**

**ports:**

**- containerPort: 3306**

**volumeMounts:**

**- name: data**

**mountPath: /var/lib/mysql**

**volumeClaimTemplates:**

**- metadata:**

**name: data**

**spec:**

**accessModes: [ "ReadWriteOnce" ]**

**resources:**

**requests:**

**storage: 10Gi**

**Here:**

* **Each pod (mysql-0, mysql-1, mysql-2) gets:**
  + **Its own PVC (data-mysql-0, data-mysql-1, etc.)**
  + **Its own stable hostname (mysql-0.mysql...)**

**🔹 5️⃣ Static IP vs. Static DNS**

* **Static DNS → Provided by Headless Service (like mysql-0.mysql)**
* **Static IP → Kubernetes doesn’t guarantee fixed pod IPs (they can change if recreated)**
  + **But with StatefulSet + Headless Service, the DNS name remains constant, which is better for cluster communication.**

**If you truly need static IPs, you can:**

* **Use a Service of type ClusterIP or LoadBalancer (with a fixed IP).**
* **Or use Network Policies / CNI plugins (like Calico) to assign static IPs at pod level (advanced setup).**

**✅ In summary:**

| **Concept** | **Description** |
| --- | --- |
| **StatefulSet** | **Manages stateful apps needing stable identity and storage** |
| **Headless Service** | **Gives stable DNS names (not IPs)** |
| **Static IP** | **Not guaranteed for pods, but stable DNS ensures reliable access** |
| **Use Case** | **Databases, Kafka, etc.** |

**116\* Have you used INSTANA?**

**117\* agroCD**

**118\* AWS glue?**

**119\* Servicemesh?**

**120\* SQL vs No SQL**

**121\* In aws, i have an instance which is critical machine that cant go down and its reaching 100% cpu utilisation.**

**122\* what are the steps that u r going to take to save this machine from going down?**

**123\* I have a ec2 instance, I want to change the private ip of that instance, can I do that?**

**124\* Can I change the private ip of running ec2 instance?**

**125\* I have a S3 bucket which my client want to track the access for internal audit.**

**126\* How can I track the access on my S3 bucket?**

**127\* For what purpose cloudtrail is used?**

**128\* I have an instance n db connected to this instance is on prem. how can I connect my application to on prem database.**

**129\* How will u connect aws with on prem database?**

**130\* nacl & security group**

**131\* nacl will be having inbound & outbound.. is it any configuration have to do manually**

**132\* traffic flow from NAT gateway to IGW**

**133\* traffic flow how cross NACL, SECURITY GROUP, NAT gateway**

**134\* Why NAT gateway why we cant NAT instance?**

**135\* In aws, i have an instance which is critical machine that cant go down and its reaching 100% cpu utilisation.**

**136\* what are the steps that u r going to take to save this machine from going down?**

**137\* where u involved in the production release of ur application? are u deploying any application**

**138\* what is the deployment strategy that is used in ur company and other stratgies that u know?**

**139\* How you will spin up VPC**

**140\* How you were using cloudwatch**

**141\* Services used in AWS ?**

**144\* S3 transfer accereration ?**

**146\* Suppose I have 1 master and 2 replicas, master crashes and what will happen to replication ?**

**147\* In case there is only one master and fails, application runs or failure ?**

**148\* Suppose you create Auto scaling group and in launch conf you specify as 3 instances, then the all the 3 instances are running in full capacity ? will the applciation fails or it will be running ? Do you think AWS will provide new instance ?**

**150\* If credentials icon is not available? What you will do?**

**🔸 1️⃣ Understand What “Credentials” Refers To**

**The Credentials icon/tab usually appears in one of these contexts:**

| **Context** | **Meaning** |
| --- | --- |
| **IAM Console** | **Access keys, passwords, signing certificates** |
| **EC2 Instance** | **“Get Windows Password” option** |
| **AWS CLI / SDK** | **Stored credentials for authentication** |

**🔹 Possible Scenarios & Solutions**

**🟢 Scenario 1: IAM User can’t see “Security Credentials”**

**Cause:  
You are using a federated login, role-based access, or root restrictions where the credentials tab is hidden.**

**Solution:**

1. **Ensure you are logged in as an IAM user, not an assumed role.**
2. **Check if your IAM permissions include:**
3. **"iam:CreateAccessKey",**
4. **"iam:ListAccessKeys",**
5. **"iam:UpdateLoginProfile"**
6. **If missing → ask an admin to update your IAM policy or enable the credentials option.**

**✅ Path:  
AWS Console → IAM → Users → *Select User* → Security credentials tab  
If the tab is missing → permission issue.**

**🟢 Scenario 2: “Get Windows Password” button missing (for EC2 Windows instance)**

**Cause:**

* **The instance is not using a key pair.**
* **You did not select the correct key pair during launch.**
* **The instance was launched using SSM Session Manager, not a key pair.**

**Solution:**

* **Verify EC2 instance has a key pair associated.**
* **If not, you cannot retrieve the password (AWS cannot decrypt it).**
* **Use SSM Session Manager to connect instead.**

**✅ Alternate method:**

1. **Go to Systems Manager → Session Manager**
2. **Connect directly to the instance (no password needed).**

**🟢 Scenario 3: “Credentials” tab missing in AWS CLI or SDK**

**Cause:**

* **CLI not configured properly.**
* **Missing ~/.aws/credentials file.**

**Solution:  
Run:**

**aws configure**

**Then enter:**

**AWS Access Key ID [None]: <YourAccessKey>**

**AWS Secret Access Key [None]: <YourSecretKey>**

**Default region name [None]: ap-south-1**

**Default output format [None]: json**

**✅ This creates:**

**~/.aws/credentials**

**~/.aws/config**

**🟢 Scenario 4: Root Account Access**

**Cause:  
Root account does not show “credentials” like IAM users (because it doesn’t use access keys by default).**

**Solution:**

* **Go to My Security Credentials (from the top-right dropdown under your username).**
* **Expand “Access keys” section → create one if needed.**

**✅ In Short**

| **Situation** | **Cause** | **Solution** |
| --- | --- | --- |
| **IAM “Credentials” tab missing** | **Missing permissions or federated login** | **Add IAM permissions or log in as IAM user** |
| **EC2 “Get Password” missing** | **No key pair selected** | **Use key pair or Session Manager** |
| **CLI credentials missing** | **Not configured** | **Run aws configure** |
| **Root account missing credentials tab** | **Root account behavior** | **Use My Security Credentials under your name** |

**💡 Quick Tip**

**If you don’t see “Security Credentials” in the console:**

* **Click on your username (top-right corner).**
* **Choose “My Security Credentials” → you’ll find keys, passwords, and MFA settings there.**

**151\* You are working on EC2 and that goes down and how do you resolve this ?**

**✅ Step-by-Step Troubleshooting Approach**

**1️⃣ Check EC2 Instance Status**

* **Go to EC2 Console → Instances → Status Checks**
* **Check:**
  + **System Status Check: Problem with AWS infrastructure**
  + **Instance Status Check: Problem inside your OS or configuration**

**If system check fails → issue on AWS side (host failure).  
If instance check fails → issue inside your instance (software/network/configuration).**

**2️⃣ If It’s a System Issue (AWS Hardware Failure):**

* **Stop the instance → Start it again → AWS migrates it to healthy hardware.**
* **If it’s in an Auto Scaling Group, a new instance will automatically launch.**
* **If using EBS-backed instance, your data will still persist.**

**3️⃣ If It’s an Instance Issue (OS or App Level):**

* **Try to connect via SSH:**
* **ssh -i mykey.pem ec2-user@<public-ip>**

**If not connecting → check Security Group, NACL, or key permissions.**

* **Use EC2 Serial Console (for Linux/Windows) if SSH fails — to view boot logs and fix login issues.**

**4️⃣ Check Logs**

* **View System Logs from EC2 console:**
  + **Right-click instance → *Monitor and troubleshoot → Get system log***
* **Check for:**
  + **Kernel panic**
  + **Disk full errors**
  + **Application crash**

**5️⃣ Check Networking**

* **Verify Security Group rules — ensure inbound/outbound ports are open.**
* **Check Route Table and NACLs (e.g., ensure 0.0.0.0/0 → IGW for public subnet).**
* **Ensure Elastic IP is associated correctly if you re-launch the instance.**

**6️⃣ Check Disk / Storage**

**If EBS volume is corrupted or full:**

* **Detach the EBS volume from the failed instance**
* **Attach it to another healthy EC2**
* **Mount and troubleshoot (remove large logs, fix fstab errors, etc.)**

**7️⃣ Recovery Option**

**If the instance is not recoverable:**

* **Launch a new EC2 instance from the same AMI or snapshot**
* **Attach the old EBS volume to restore data**

**8️⃣ Preventive Measures**

* **Use Auto Scaling Group (ASG) for automatic recovery**
* **Enable CloudWatch alarms to detect instance health failures**
* **Take regular AMI snapshots or EBS backups**
* **Enable Termination Protection**

**🗣️ Short Interview Answer:**

**If my EC2 instance goes down, I first check the instance and system status checks in the EC2 console.  
If it’s a system issue, I stop and start the instance to move it to healthy hardware.  
If it’s an OS issue, I check logs, security groups, or attach the volume to another instance for recovery.  
For critical workloads, I use Auto Scaling and CloudWatch alarms to ensure high availability and automatic recovery.**

**152\* You want to give a access to a EC2 and you have only pvt key and how do you enable him to access the EC2 ?**

**Since the private key allows *only you* to SSH, you should not share that .pem file (for security reasons).  
Instead, you can securely add the new user’s SSH public key to the EC2 instance.**

**🔸 Option 1: (Best Practice) Add their public key to the EC2 instance**

1. **Ask the user to generate their own key pair on their system:**
2. **ssh-keygen -t rsa**

**This creates two files:**

* + **id\_rsa (private key — they keep it safe)**
  + **id\_rsa.pub (public key — they share with you)**

1. **You SSH into EC2 using your existing private key:**
2. **ssh -i mykey.pem ec2-user@<public-ip>**
3. **Create a new user for them (optional but recommended):**
4. **sudo adduser devuser**
5. **Create SSH directory and authorized\_keys file:**
6. **sudo mkdir /home/devuser/.ssh**
7. **sudo nano /home/devuser/.ssh/authorized\_keys**
8. **Paste their public key (id\_rsa.pub) into authorized\_keys.**
9. **Set proper permissions:**
10. **sudo chmod 700 /home/devuser/.ssh**
11. **sudo chmod 600 /home/devuser/.ssh/authorized\_keys**
12. **sudo chown -R devuser:devuser /home/devuser/.ssh**
13. **Now they can SSH using:**
14. **ssh -i id\_rsa devuser@<public-ip>**

**✅ Result:  
They can now log in securely without needing your private key.**

**🔸 Option 2: (If You Must Reuse Existing Key Pair)**

**If you want to temporarily share access and don’t have a new key:**

* **You can copy your authorized\_keys entry to their user profile.**
* **sudo cp ~/.ssh/authorized\_keys /home/devuser/.ssh/**

**But ⚠️ this is not secure, since both users now share the same key pair.**

**🔸 Option 3: Use AWS Systems Manager (SSM Session Manager)**

**If SSH access is restricted or keys are lost:**

* **Enable SSM Agent on the EC2 instance.**
* **Give the IAM user an IAM role with AmazonSSMManagedInstanceCore policy.**
* **They can connect via the AWS Console → Session Manager, without SSH keys.**

**✅ This is the most secure and recommended AWS-native method.**

**💡 Best Interview Answer Summary**

**“I won’t share my private key.  
Instead, I’ll ask the user to generate their own SSH key pair and share their public key.  
Then, I’ll add that public key to the EC2 instance under /home/<username>/.ssh/authorized\_keys with proper permissions.**

**Alternatively, if the EC2 has SSM Agent enabled, I’ll grant them access via AWS Systems Manager Session Manager, which doesn’t require key pairs at all.”**

**153\* steps to spin up EC2 instance on linux server**

**1️⃣ Log in to AWS Management Console**

* **Go to EC2 Dashboard under Compute Services.**

**2️⃣ Click “Launch Instance”**

* **Click the Launch Instance button to start the wizard.**

**3️⃣ Configure Basic Details**

* **Name: Give a name to your instance (e.g., web-server).**
* **Tags: Add tags like Environment=Dev or Project=Ecommerce.**

**4️⃣ Choose an Amazon Machine Image (AMI)**

* **Select a Linux-based AMI such as:**
  + **Amazon Linux 2**
  + **Ubuntu Server**
  + **Red Hat Enterprise Linux**
  + **CentOS**

**📘 *AMI defines the OS and base configuration of your instance.***

**5️⃣ Choose Instance Type**

* **Example types:**
  + **t2.micro or t3.micro → Free-tier eligible (for small workloads)**
  + **m5.large → Medium workloads**
* **This determines CPU, memory, and performance.**

**6️⃣ Configure Key Pair (for SSH access)**

* **Choose an existing key pair or create a new one.**
* **Download the .pem file — it’s used to SSH into your instance securely.**
* **Set proper permissions:**
* **chmod 400 mykey.pem**

**7️⃣ Configure Network Settings**

* **Choose VPC and Subnet (public subnet for internet access).**
* **Assign a public IP if you want to connect from the internet.**
* **Select or create a Security Group to control inbound/outbound traffic:**
  + **Allow SSH (port 22) for admin access.**
  + **Optionally allow HTTP (port 80) or HTTPS (port 443) for web access.**

**8️⃣ Configure Storage (EBS Volume)**

* **Default volume: 8 GB gp3 (general purpose SSD)**
* **Increase if needed (e.g., 30 GB)**
* **Optionally enable encryption.**

**9️⃣ Review and Launch**

* **Review all settings.**
* **Click Launch Instance.**
* **Once status = running, it’s ready to use.**

**10️⃣ Connect to the Instance**

**Use SSH from your terminal:**

**ssh -i mykey.pem ec2-user@<public-ip-address>**

**For Ubuntu AMI:**

**ssh -i mykey.pem ubuntu@<public-ip-address>**

**✅ After Connection**

**You can now:**

* **Install software (sudo yum install or sudo apt install)**
* **Deploy your application**
* **Configure monitoring or Auto Scaling**

**🧠 Interview Short Answer**

**“To spin up an EC2 instance on Linux, I log in to the AWS console, select a Linux AMI, choose an instance type, configure key pair and network settings, allow SSH in the security group, allocate storage, and then launch it.  
I connect via SSH using the PEM key to access the Linux server.”**

**154\* why Loadbalancer required**

**A Load Balancer (LB) is a service that automatically distributes incoming network traffic across multiple backend servers (like EC2 instances, containers, or IPs).**

**It ensures that no single server is overloaded, improving performance, reliability, and availability.**

**💡 Why Load Balancer is Required (Main Reasons)**

**1️⃣ Distribute Traffic Evenly**

* **Without a load balancer, all users connect to one instance → it becomes overloaded.**
* **LB spreads incoming requests across multiple instances → avoids bottlenecks.**

**🧩 Example:  
If 3 EC2 instances host your website:  
CloudFront / DNS → Load Balancer → EC2-1, EC2-2, EC2-3**

**2️⃣ High Availability & Fault Tolerance**

* **If one EC2 instance fails, LB automatically routes traffic to healthy instances.**
* **Ensures no downtime for users.**

**✅ Example:  
During instance patching or failure, traffic continues via remaining healthy targets.**

**3️⃣ Scalability**

* **Works with Auto Scaling Groups (ASG) to add/remove instances automatically based on demand.**
* **LB always knows which instances are available to handle new traffic.**

**4️⃣ Centralized Security Control**

* **Acts as a single entry point, so you can apply:**
  + **SSL/TLS encryption**
  + **WAF (Web Application Firewall)**
  + **Access logs**
  + **Security groups**

**5️⃣ Better Performance**

* **Balances connections based on algorithms like:**
  + **Round Robin**
  + **Least Connections**
  + **IP Hash**
* **Helps maintain optimal response times.**

**6️⃣ Support for Multiple Protocols**

**Different AWS Load Balancers support different use cases:**

| **Type** | **Layer** | **Used For** |
| --- | --- | --- |
| **Application Load Balancer (ALB)** | **Layer 7 (HTTP/HTTPS)** | **Web apps, microservices** |
| **Network Load Balancer (NLB)** | **Layer 4 (TCP/UDP)** | **High-performance or low latency apps** |
| **Gateway Load Balancer (GLB)** | **Layer 3 (IP)** | **Integrating 3rd party firewalls, appliances** |
| **Classic Load Balancer (CLB)** | **Layer 4 & 7** | **Legacy support** |

**7️⃣ Zero Downtime Deployment**

* **Load balancers enable rolling updates — remove one instance for update, others serve traffic.**
* **Ensures continuous availability during maintenance.**

**🧠 Simple Example**

**Imagine you have an e-commerce website 🛍️:**

* **1 server = 100 users (slow and risky)**
* **3 servers behind a Load Balancer = 300 users (smooth and resilient)**

**If one server goes down, users are seamlessly redirected to others → no service interruption.**

**✅ Summary Table**

| **Reason** | **Benefit** |
| --- | --- |
| **Traffic distribution** | **Prevents overload** |
| **High availability** | **Handles instance failures** |
| **Auto scaling** | **Matches user demand** |
| **Central security** | **Single HTTPS endpoint** |
| **Performance** | **Faster response** |
| **Maintenance** | **Zero downtime updates** |

**155\* where you are storing your application in cloud**

**🟢 1️⃣ Start with a General Overview**

**“It depends on the type of application — static, dynamic, or containerized.  
In our AWS setup, we store and deploy applications using services like EC2, S3, ECS/Fargate, or Elastic Beanstalk, depending on the use case.”**

**🟣 2️⃣ Example Answers by Application Type**

**✅ A. Static Website / Frontend**

**“For our frontend (React or Angular) application, we store and host it in an S3 bucket with static website hosting enabled.  
We also use CloudFront as a CDN to distribute content globally with HTTPS and caching.”**

**Flow:**

**Code (HTML/JS/CSS) → S3 Bucket → CloudFront → End Users**

**✅ B. Backend / Web Application**

**“For the backend, which runs on Java/Python/Node.js, we deploy it on Amazon EC2 instances within a VPC.  
The application code is stored in a Git repository (GitHub/CodeCommit), and we use CodeDeploy or Jenkins for deployment to EC2.”**

**Flow:**

**GitHub / CodeCommit → CodeDeploy / Jenkins → EC2 (Application)**

**✅ C. Containerized Applications**

**“For microservices and containerized applications, we store Docker images in Amazon ECR (Elastic Container Registry) and deploy them using Amazon ECS or AWS Fargate.”**

**Flow:**

**Docker Image → Amazon ECR → ECS/Fargate → Service Runs Application**

**✅ D. Serverless Applications**

**“For serverless components, we use AWS Lambda to store and run the code, triggered by events from API Gateway, S3, or CloudWatch.”**

**Flow:**

**Code → Lambda → Triggered by S3/API Gateway/Event**

**🟢 3️⃣ Include Storage for Data or Artifacts**

**“Apart from code, application data is stored in:**

* **Amazon RDS (for relational databases like MySQL, PostgreSQL)**
* **DynamoDB (for NoSQL data)**
* **S3 (for logs, backups, and large files).”**

**✅ Sample Interview Answer (Complete)**

**“We store and deploy our application across multiple AWS services.  
The frontend static files are stored in an S3 bucket with CloudFront for global distribution.**

**The backend code is deployed to EC2 instances managed through Auto Scaling and Load Balancers.**

**For containerized services, we use ECR to store Docker images and deploy them via ECS Fargate.**

**All data and backups are stored securely in RDS and S3.  
This setup ensures scalability, high availability, and cost efficiency.”**

**156\* have you setup master & slave in jenkins? why slave machine required**

**🔹 1️⃣ Jenkins Master–Slave (Controller–Agent) Architecture**

**In modern Jenkins, the terms are:**

* **Master → now called Controller**
* **Slave → now called Agent**

**But the concept is the same.**

**🔸 2️⃣ What Is Jenkins Master (Controller)?**

**The Jenkins master (controller) is the central server responsible for:**

* **Managing Jenkins configuration and UI**
* **Scheduling jobs (builds)**
* **Dispatching jobs to available agents**
* **Monitoring job execution**
* **Managing plugins and user access**

**It doesn’t necessarily run the builds itself — that’s where agents (slaves) come in.**

**🔸 3️⃣ What Is a Jenkins Slave (Agent)?**

**A Jenkins slave (agent) is a separate machine (physical, VM, or container) that connects to the Jenkins master and executes build jobs.**

**It can be:**

* **Linux, Windows, or Docker container**
* **Connected via SSH, JNLP, or Kubernetes Pod**

**🔹 4️⃣ Why Slave Machines Are Required**

| **Reason** | **Explanation** |
| --- | --- |
| **1️⃣ Distribute workload** | **Offload build and test jobs from the master; improves performance.** |
| **2️⃣ Parallel execution** | **Run multiple jobs in parallel on different agents.** |
| **3️⃣ Environment isolation** | **Each agent can have its own environment (Java, Node.js, Python, Docker, etc.)** |
| **4️⃣ Scalability** | **Add more agents as workload grows without overloading the master.** |
| **5️⃣ OS-specific builds** | **Some jobs need Windows, others Linux → use different slaves for each.** |
| **6️⃣ Resource management** | **Prevent master from being overloaded by heavy build tasks.** |

**🔹 5️⃣ How to Set Up Master–Slave Configuration**

**Step-by-step (SSH-based agent example):**

1. **Install Jenkins on master (controller)**
   * **Ensure SSH access to the agent machines.**
2. **Add a new node (agent)**
   * **Go to: Manage Jenkins → Manage Nodes and Clouds → New Node**
   * **Choose: *Permanent Agent***
   * **Give it a name and description.**
3. **Configure node settings**
   * **Remote root directory: /home/jenkins**
   * **Labels (optional): linux, build, etc.**
   * **Launch method: *Launch agent via SSH***
4. **Provide credentials**
   * **Add SSH credentials for connecting to the agent.**
5. **Save and connect**
   * **Jenkins will automatically copy agent.jar to the slave and connect.**

**Example Jenkins File (to run on specific agent):**

**pipeline {**

**agent { label 'linux' }**

**stages {**

**stage('Build') {**

**steps {**

**echo 'Building on Linux agent...'**

**}**

**}**

**}**

**}**

**🔹 6️⃣ In Real Projects**

**You might use:**

* **Static agents → permanently attached (for dedicated builds)**
* **Dynamic agents → auto-created via Kubernetes plugin or EC2 plugin**

**For example:**

* **Build agent on EC2 → for compiling and packaging apps**
* **Docker agent → for isolated build environments**
* **Windows agent → for .NET or Windows builds**

**✅ Summary Table**

| **Component** | **Role** |
| --- | --- |
| **Jenkins Master (Controller)** | **Manages job scheduling, UI, and coordination** |
| **Jenkins Slave (Agent)** | **Executes build/test jobs** |
| **Why Needed** | **Scalability, isolation, parallel builds, and load balancing** |

**✅ In short:**

**Jenkins slaves (agents) are required to offload and distribute build workloads, run jobs in different environments, and scale Jenkins efficiently without overloading the master.**

**157\* cloud watch**

**Amazon CloudWatch is a monitoring and observability service in AWS.  
It collects and tracks metrics, logs, and events from your AWS resources and applications, helping you monitor performance, detect issues, and automate responses.**

**👉 In simple words:**

**CloudWatch = Monitor + Alert + Dashboard for AWS resources**

**🔹 1️⃣ Key Functions of CloudWatch**

| **Function** | **Description** |
| --- | --- |
| **Metrics** | **Numeric data points about resource performance (CPU, Memory, Network, etc.)** |
| **Logs** | **Captures log files from EC2, Lambda, ECS, applications, etc.** |
| **Alarms** | **Set thresholds to trigger notifications or actions (via SNS or Auto Scaling).** |
| **Events / Rules** | **React to changes in AWS environment automatically.** |
| **Dashboards** | **Custom visual dashboards to monitor resources in real-time.** |

**🔹 2️⃣ CloudWatch Components Explained**

**✅ Metrics**

* **Fundamental data points like:**
  + **EC2 → CPUUtilization, DiskReadOps, NetworkIn**
  + **RDS → FreeStorageSpace, ReadIOPS**
  + **ELB → RequestCount, Latency**
  + **Lambda → Invocations, Errors, Duration**
* **Default metric interval = 1 minute (can go up to 1 second in detailed monitoring).**

**✅ Logs**

* **Stores logs from:**
  + **EC2 instances (via CloudWatch Agent)**
  + **AWS Lambda**
  + **API Gateway**
  + **ECS, EKS, CloudTrail, and custom apps**
* **You can search and filter logs or set alerts from them.**

**Example:  
Monitor an application log for “ERROR” messages and trigger an alarm.**

**✅ Alarms**

* **You can set alarms on any metric.**
* **If threshold is breached → trigger an action like:**
  + **Send SNS notification (email/SMS)**
  + **Scale EC2 Auto Scaling Group**
  + **Stop/reboot instance**

**Example:  
If CPUUtilization > 80% for 5 mins → send alert or auto-scale.**

**✅ Events (EventBridge)**

* **Detects changes in AWS environment.**
* **Example:**
  + **EC2 instance stopped → send SNS alert or start Lambda function.**

**✅ Dashboards**

* **Visual interface to monitor multiple metrics.**
* **Can include graphs, numbers, alarms.**
* **Example: Create a dashboard showing EC2 CPU, RDS IOPS, and ALB latency.**

**🔹 3️⃣ CloudWatch Data Flow**

**AWS Services → CloudWatch (Metrics & Logs)**

**↓**

**CloudWatch Alarms**

**↓**

**SNS / Lambda / Auto Scaling Actions**

**🔹 4️⃣ CloudWatch Agent**

**To collect custom metrics or OS-level metrics (like memory or disk), you install the CloudWatch Agent on EC2 or on-prem servers.**

**Commands:**

**sudo yum install amazon-cloudwatch-agent**

**sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-config-wizard**

**sudo systemctl start amazon-cloudwatch-agent**

**This lets you monitor:**

* **Memory usage**
* **Disk space**
* **Swap utilization**
* **Application logs**

**🔹 5️⃣ CloudWatch Logs Insights**

**A query engine that allows you to analyze logs using SQL-like queries.**

**Example:**

**fields @timestamp, @message**

**| filter @message like /ERROR/**

**| sort @timestamp desc**

**| limit 10**

**🔹 6️⃣ CloudWatch vs CloudTrail**

| **Feature** | **CloudWatch** | **CloudTrail** |
| --- | --- | --- |
| **Purpose** | **Monitoring performance** | **Tracking API activity** |
| **Data Type** | **Metrics, Logs** | **API calls, user actions** |
| **Example** | **CPU usage, Errors** | **Who deleted S3 bucket** |
| **Use Case** | **Performance alerts** | **Security/audit trail** |

**✅ Use both together — CloudTrail for who did what, CloudWatch for how your system performs.**

**🔹 7️⃣ CloudWatch Pricing (Approximate)**

| **Resource** | **Free Tier** | **Paid** |
| --- | --- | --- |
| **Metrics** | **10 metrics** | **~$0.30 per metric/month** |
| **Logs** | **5 GB/month** | **~$0.50 per GB ingested** |
| **Alarms** | **10 alarms** | **~$0.10 per alarm/month** |
| **Dashboards** | **3 dashboards** | **~$3 per dashboard/month** |

**✅ In Short**

| **Feature** | **Description** |
| --- | --- |
| **Service Type** | **Monitoring & observability** |
| **Collects** | **Metrics, logs, events** |
| **Actions** | **Alarms, notifications, scaling** |
| **Integrations** | **SNS, Lambda, Auto Scaling** |
| **Agent** | **Used for custom system metrics** |
| **Best Use** | **Monitor, alert, and visualize AWS workloads** |

**Example Use Case:**

**You create a CloudWatch alarm for EC2 CPUUtilization > 80%.  
When triggered, it sends an SNS alert and triggers Auto Scaling to launch a new instance.**

**158\* do you have experience in lambda function & use case of lambda**

**Yes, I have hands-on experience working with AWS Lambda.**

**Lambda is a serverless compute service that lets you run code without provisioning or managing servers — you just upload your code, and it automatically scales and executes when triggered.**

**I’ve used Lambda in several use cases, such as:**

**🔹 S3 Event Trigger:  
When a file is uploaded to an S3 bucket, a Lambda function gets triggered to process or validate the file — for example, resizing images or moving the file to another S3 bucket.**

**🔹 CloudWatch Automation:  
A Lambda function is triggered by CloudWatch alarms to automatically start/stop EC2 instances at specific times to save cost.**

**🔹 API Gateway Integration:  
Used Lambda with API Gateway to build serverless REST APIs — no need to manage EC2 or containers.**

**🔹 Data Transformation (ETL):  
Lambda pulls data from S3 or DynamoDB, processes it, and loads it into another storage service like Redshift or RDS.**

**Lambda supports multiple languages — Python, Node.js, Java, etc., and integrates with services like SNS, SQS, DynamoDB, and EventBridge.**

**🗣️ Short Interview Answer:**

**Yes, I’ve used AWS Lambda mainly for S3 event-driven automation and API Gateway-based serverless APIs.  
For example, when a file is uploaded to S3, my Lambda function validates and moves it to a backup bucket automatically — no EC2 or manual effort required.**

**⚙️ Example Trigger Flow:**

**S3 → (upload event) → Lambda → Process File → Store in Another S3 / RDS**

**159\* when its advicable to use vpc endpoint or IGW**

**🔹 1️⃣ Internet Gateway (IGW)**

**✅ Purpose**

**An Internet Gateway allows instances in your VPC to connect to the public internet.**

**It provides:**

* **Outbound access (EC2 → Internet)**
* **Inbound access (Internet → EC2, if public IP & security rules allow)**

**⚙️ When to Use IGW**

**Use Internet Gateway when:**

* **You need public internet access for your resources.**
* **Your EC2 instances need to download packages, access APIs, or serve web traffic publicly.**
* **Your app is public-facing (e.g., website, ALB, public API server).**

**💡 Example**

* **Public subnet with EC2 running a web server**
* **Route table:**
* **0.0.0.0/0 → Internet Gateway (igw-xxxx)**
* **The EC2 has a public IP or Elastic IP.**

**🔹 2️⃣ VPC Endpoint**

**✅ Purpose**

**A VPC Endpoint allows private connectivity between your VPC and AWS services (like S3, DynamoDB, ECR, etc.)  
without using the Internet or public IPs.**

**VPC Endpoints come in two types:**

| **Type** | **Example** | **Used For** |
| --- | --- | --- |
| **Gateway Endpoint** | **S3, DynamoDB** | **Routes via route table** |
| **Interface Endpoint** | **SSM, EC2 API, CloudWatch, Secrets Manager** | **Uses ENI (Elastic Network Interface)** |

**⚙️ When to Use VPC Endpoint**

**Use VPC Endpoint when:**

* **Your EC2 is in a private subnet (no IGW) but needs to access AWS services like S3 or DynamoDB.**
* **You want secure, private communication that doesn’t go over the internet.**
* **You want to reduce data transfer cost and increase security (no public exposure).**

**💡 Example**

* **EC2 in private subnet needs to upload to S3:**
  + **Create a Gateway Endpoint for S3**
  + **Update route table:**
  + **Destination: com.amazonaws.ap-south-1.s3 → Target: vpce-xxxxxx**
  + **Now EC2 can access S3 without Internet Gateway or NAT Gateway.**

**🔹 3️⃣ Summary Table**

| **Feature** | **Internet Gateway (IGW)** | **VPC Endpoint** |
| --- | --- | --- |
| **Direction** | **Public internet access** | **Private AWS access** |
| **Security** | **Publicly exposed (if not managed carefully)** | **Stays within AWS network** |
| **Cost** | **Free (but NAT/Data transfer may cost)** | **Interface endpoints cost per hour + data** |
| **Usage** | **Web servers, public-facing apps** | **Private apps needing AWS service access** |
| **Example** | **EC2 → Internet** | **EC2 (private subnet) → S3** |

**✅ Interview-Ready Answer**

**“An Internet Gateway is used when your EC2 or application needs access to or from the public internet — for example, hosting a website or accessing external APIs.**

**A VPC Endpoint is used when your instance in a private subnet needs to access AWS services like S3 or DynamoDB privately without using the internet.**

**So, if security and private connectivity are priorities, I’d use a VPC Endpoint.  
If public access is required, I’d use an Internet Gateway.”**

**160\* we have deploy an application in EC2 so i want to block few http ip address**

**✅ Scenario**

**You have a web application running on an EC2 instance (say Apache, Nginx, or Tomcat).  
You want to block specific IP addresses from accessing it.**

**🔹 There are 3 main ways to block IPs:**

**1️⃣ Using Security Groups (Network Layer – Basic Blocking)**

**Best for blocking at AWS network level.**

**But note:**

**Security Groups work on allow rules only, not deny rules.  
You cannot explicitly block an IP — you only allow certain IPs.**

**Example:**

* **If you only want certain IPs to access:**
  + **In the Inbound rules, allow only:**
  + **HTTP (80) → 203.0.113.5/32**
  + **Remove 0.0.0.0/0.**
  + **This restricts access to only that IP.**

**If you want to block only specific IPs, go for NACLs or application-level blocking below 👇**

**2️⃣ Using Network ACL (NACL – Subnet Level Blocking)**

**NACLs allow both “Allow” and “Deny” rules (unlike Security Groups).**

**Steps:**

1. **Go to VPC → Network ACLs.**
2. **Choose the NACL associated with your subnet.**
3. **Add an Inbound rule:**
4. **Rule #: 100**
5. **Type: HTTP (80)**
6. **Source: <blocked-ip>/32**
7. **Action: DENY**
8. **Add a default “Allow” rule below it for other traffic.**
9. **Save changes.**

**✅ Result: Any request from that IP on port 80 is dropped before it reaches EC2.**

**3️⃣ Using Web Server Configuration (Application Layer Blocking)**

**a. Apache**

**Edit the configuration file (/etc/httpd/conf/httpd.conf or .htaccess):**

**<Directory "/var/www/html">**

**Order Allow,Deny**

**Allow from all**

**Deny from 203.0.113.5**

**Deny from 198.51.100.10**

**</Directory>**

**Then restart Apache:**

**sudo systemctl restart httpd**

**b. Nginx**

**In your Nginx config (/etc/nginx/nginx.conf or /etc/nginx/conf.d/default.conf):**

**server {**

**listen 80;**

**server\_name yourdomain.com;**

**location / {**

**deny 203.0.113.5;**

**deny 198.51.100.10;**

**allow all;**

**}**

**}**

**Restart Nginx:**

**sudo systemctl restart nginx**

**4️⃣ Optional: Use AWS WAF (Web Application Firewall)**

**If your EC2 is behind ALB or CloudFront, you can:**

* **Attach AWS WAF**
* **Create a Web ACL**
* **Add a “IP Set” rule → block those IPs**

**✅ This is the recommended enterprise solution (scalable, centralized, managed).**

**⚡ Summary Table**

| **Method** | **Level** | **Can Block IPs?** | **Best For** |
| --- | --- | --- | --- |
| **Security Group** | **Instance** | **❌ (allow only)** | **Simple access control** |
| **NACL** | **Subnet** | **✅ Yes** | **Blocking at network layer** |
| **Web Server Config** | **OS/App** | **✅ Yes** | **Per-app fine control** |
| **AWS WAF** | **Managed** | **✅ Yes** | **Scalable production setup** |

**🧠 Interview Short Answer**

**“To block specific HTTP IPs, I’d use a Network ACL to deny traffic from those IPs on port 80.  
Alternatively, at the application layer, I can configure the web server (Apache/Nginx) to deny requests from those IPs.  
For large-scale setups, I’d use AWS WAF attached to ALB or CloudFront for IP-based blocking.”**

**161\* have experience with docker images. you deployed into AWS?**

**“Yes, I have hands-on experience working with Docker.  
I built custom Docker images for my applications, pushed them to Amazon ECR (Elastic Container Registry), and then deployed the containers using AWS services like ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service).”**

**⚙️ Step-by-Step Process**

**1️⃣ Create Docker Image**

* **Write a Dockerfile for your application (e.g., Node.js, Python, Java, etc.)**
* **Example:**
* **FROM node:18**
* **WORKDIR /app**
* **COPY . .**
* **RUN npm install**
* **CMD ["npm", "start"]**
* **Build and tag the image:**
* **docker build -t myapp:latest .**

**2️⃣ Push Image to Amazon ECR**

* **Create a repository in ECR (private or public)**
* **Authenticate Docker to ECR:**
* **aws ecr get-login-password --region ap-south-1 | docker login --username AWS --password-stdin <account-id>.dkr.ecr.ap-south-1.amazonaws.com**
* **Tag and push the image:**
* **docker tag myapp:latest <account-id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:latest**
* **docker push <account-id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:latest**

**3️⃣ Deploy Container Using ECS**

* **Create a Task Definition → specifies:**
  + **Container image (from ECR)**
  + **CPU, memory**
  + **Port mappings (e.g., 80:8080)**
* **Create an ECS Service (Fargate or EC2 launch type)**
* **Attach a Load Balancer (ALB) to route traffic to the containers**

**4️⃣ (Optional) Deploy via EKS (Kubernetes)**

* **Push image to ECR**
* **Deploy via Kubernetes YAML manifest:**
* **apiVersion: apps/v1**
* **kind: Deployment**
* **metadata:**
* **name: myapp**
* **spec:**
* **replicas: 3**
* **template:**
* **spec:**
* **containers:**
* **- name: myapp**
* **image: <account-id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:latest**
* **ports:**
* **- containerPort: 8080**
* **Expose via LoadBalancer service for public access.**

**5️⃣ Continuous Deployment (Optional)**

**You can integrate CI/CD using:**

* **CodePipeline + CodeBuild → build image automatically on commit**
* **Push to ECR → ECS service auto-deploys new version**

**✅ In Summary**

| **Step** | **Task** | **AWS Service** |
| --- | --- | --- |
| **1** | **Create Docker image** | **Docker** |
| **2** | **Store image** | **Amazon ECR** |
| **3** | **Run container** | **ECS or EKS** |
| **4** | **Expose via LB** | **ALB / NLB** |
| **5** | **Automate pipeline** | **CodePipeline, CodeBuild** |

**💡 Example Answer (Concise for Interview)**

**“Yes, I’ve deployed Dockerized applications to AWS.  
I built the Docker image locally, pushed it to Amazon ECR, and then deployed it using ECS with Fargate.  
I also integrated CodePipeline for automated deployments.  
The setup was behind an Application Load Balancer, and I used CloudWatch for monitoring container logs.”**

**162\* tell me about lambda syncronous and asyncronous using word configuration.**

**🟢 1️⃣ Basic Concept**

**AWS Lambda invocation modes:**

* **Synchronous (Request-Response)**
* **Asynchronous (Event-based)**

**These determine how the function is invoked and how AWS handles the response.**

**⚙️ 2️⃣ Synchronous Invocation**

**➡ Definition:  
In synchronous invocation, the caller (client or service) waits for the Lambda function to complete and return a response.**

**➡ Example Services:**

* **API Gateway**
* **ALB (Application Load Balancer)**
* **AWS SDK / CLI (when you invoke directly)**

**➡ Flow:**

**Caller → Invoke Lambda → Wait for Result → Response Returned**

**➡ Configuration Aspect:**

* **There is no event queue.**
* **You can configure timeout and memory size in the Lambda function configuration.**
* **Caller directly handles retries (Lambda itself doesn’t retry in sync mode).**

**➡ Example Use Case:**

**Real-time API request processing, like user authentication or data validation.**

**Example:**

**API Gateway → Lambda → Returns HTTP response to client**

**⚙️ 3️⃣ Asynchronous Invocation**

**➡ Definition:  
In asynchronous invocation, the caller does not wait for the function to complete.  
Lambda queues the event and runs the function in the background.**

**➡ Example Services:**

* **S3 Event Notifications**
* **SNS (Simple Notification Service)**
* **CloudWatch Events / EventBridge**

**➡ Flow:**

**Caller → Lambda (event queued) → Lambda executes later → Result not returned to caller**

**➡ Configuration Aspect (Important 🔥):  
You can configure asynchronous invocation settings under Lambda’s Configuration → Asynchronous invocation section.**

**There you can set:**

* **Maximum retry attempts (default: 2 retries)**
* **Maximum event age (how long the event can remain in the queue, default: 6 hours)**
* **Dead-letter queue (DLQ) or Destination:**
  + **Success Destination – where to send success results**
  + **Failure Destination – where to send failed events (e.g., SNS or SQS)**

**➡ Example Use Case:**

**Background tasks like image resizing, sending emails, or log processing.**

**Example:**

**S3 → Event → Lambda → Process file asynchronously → Send result to SNS**

**⚙️ 4️⃣ Comparison Table**

| **Feature** | **Synchronous** | **Asynchronous** |
| --- | --- | --- |
| **Caller waits for response?** | **✅ Yes** | **❌ No** |
| **Examples** | **API Gateway, ALB, SDK** | **S3, SNS, CloudWatch Events** |
| **Retry behavior** | **Controlled by caller** | **Automatic (configurable retries)** |
| **Event queue** | **None** | **Managed by Lambda** |
| **Failure handling** | **Caller handles** | **DLQ or Destination configuration** |
| **Use case** | **Real-time APIs** | **Background processing** |

**✅ Sample Interview Answer**

**“In AWS Lambda, there are two types of invocation — synchronous and asynchronous, which can be configured under the Lambda’s Configuration tab.**

**In synchronous invocation, the caller waits for a response from Lambda — for example, when invoked by API Gateway. The result is returned immediately, and retries are handled by the caller.**

**In asynchronous invocation, Lambda queues the event and executes it in the background — for example, when triggered by S3 or SNS.**

**Under the Lambda Configuration → Asynchronous invocation settings, we can specify retry attempts, maximum event age, and destinations for success or failure events (like SQS or SNS).**

**This configuration helps in managing event retries and ensuring no data loss even if the Lambda fails.”**

**163\* how to create static website in s3**

**A static website is one that serves fixed content — HTML, CSS, JS, images, etc.  
There’s no backend code (like PHP, Node.js, or databases).**

**Example:  
Portfolio site, company landing page, documentation, etc.**

**🔹 2️⃣ Steps to Create & Host a Static Website on S3**

**Step 1: Create an S3 Bucket**

1. **Go to AWS Management Console → S3 → Create bucket**
2. **Enter a unique bucket name (e.g., my-static-website)**
3. **Choose your region**
4. **Uncheck “Block all public access”  
   → Confirm you want to make the bucket public (needed for website hosting)**
5. **Click Create bucket**

**Step 2: Upload Website Files**

* **Upload your HTML, CSS, JS, and image files to the bucket.  
  Example:**
* **index.html**
* **about.html**
* **styles.css**
* **script.js**
* **Use “Upload” → Add files → Upload”**

**Step 3: Enable Static Website Hosting**

1. **Go to Properties tab of your bucket**
2. **Scroll down to Static website hosting**
3. **Choose “Use this bucket to host a website”**
4. **Provide:**
   * **Index document: index.html**
   * **Error document: error.html *(optional)***
5. **Save the changes**
6. **Copy the Endpoint URL shown — something like:**
7. **http://my-static-website.s3-website-us-east-1.amazonaws.com**

**Step 4: Make Content Public**

1. **Go to Permissions → Bucket Policy**
2. **Add this JSON policy (replace your bucket name):**
3. **{**
4. **"Version": "2012-10-17",**
5. **"Statement": [**
6. **{**
7. **"Sid": "PublicReadGetObject",**
8. **"Effect": "Allow",**
9. **"Principal": "\*",**
10. **"Action": "s3:GetObject",**
11. **"Resource": "arn:aws:s3:::my-static-website/\*"**
12. **}**
13. **]**
14. **}**
15. **Save the policy — now your files are publicly readable.**

**Step 5: (Optional) Add Custom Domain Using Route 53**

**If you want your website at www.mywebsite.com:**

1. **Register domain in Route 53 (or use existing one)**
2. **Create a CNAME record pointing to your S3 endpoint**
   * **Record name: www**
   * **Value: my-static-website.s3-website-us-east-1.amazonaws.com**

**Step 6: (Optional) Use CloudFront for HTTPS**

**S3 website endpoints only support HTTP.  
To enable HTTPS, create a CloudFront distribution:**

* **Origin: your S3 bucket**
* **Viewer protocol policy: Redirect HTTP to HTTPS**
* **Attach ACM certificate for your custom domain**

**🔹 3️⃣ Verification**

**Visit the website URL:**

**http://my-static-website.s3-website-us-east-1.amazonaws.com**

**✅ Your static website should load successfully.**

**✅ Summary Table**

| **Step** | **Action** | **Purpose** |
| --- | --- | --- |
| **1** | **Create S3 bucket** | **Host website files** |
| **2** | **Upload files** | **Add HTML, CSS, JS** |
| **3** | **Enable static hosting** | **Make S3 serve website** |
| **4** | **Add bucket policy** | **Allow public read access** |
| **5** | **(Optional) Route 53** | **Use custom domain** |
| **6** | **(Optional) CloudFront** | **Enable HTTPS + caching** |

**✅ In short:**

**To host a static website on S3, create a bucket, upload your website files, enable static website hosting, make the content public, and optionally connect it to a domain with Route 53 and CloudFront.**

**164\* what are the use cases you used in s3**

**Amazon S3 (Simple Storage Service) is an object storage service that stores data as *objects* (files + metadata) in *buckets*.  
It’s:**

* **Highly durable (99.999999999% durability)**
* **Scalable**
* **Cost-effective**
* **Integrated with many AWS services**

**🔹 2️⃣ Common & Real-World Use Cases of Amazon S3**

**Here are key use cases you can confidently mention 👇**

**✅ 1. Static Website Hosting**

* **Host static content (HTML, CSS, JS) directly from an S3 bucket.**
* **Enable static website hosting under bucket properties.**
* **Use CloudFront for CDN and Route 53 for DNS.**

**Example:**

**Hosted a company’s marketing website and documentation portal using S3 + CloudFront.**

**✅ 2. Backup and Archival**

* **Store daily or weekly backups from EC2, RDS, or on-prem servers.**
* **Use Lifecycle policies to move old backups to Glacier or Glacier Deep Archive for cost savings.**

**Example:**

**Configured RDS automated backups to S3 and lifecycle policies to move data to Glacier after 30 days.**

**✅ 3. Data Lake / Big Data Storage**

* **S3 acts as the central data lake for analytics pipelines.**
* **Services like AWS Glue, Athena, EMR, and Redshift Spectrum directly read data from S3.**

**Example:**

**Used S3 as a data lake for storing raw and processed log data for analytics with Athena.**

**✅ 4. Application File Storage**

* **Store user-generated content like images, videos, documents, etc.**
* **Accessed through pre-signed URLs for security.**

**Example:**

**In an e-commerce app, user profile images and product images were stored in S3 with access managed via pre-signed URLs.**

**✅ 5. Log Storage and Analysis**

* **Centralized logging location for:**
  + **CloudTrail**
  + **ELB/ALB logs**
  + **VPC Flow Logs**
  + **Application logs**
* **Later processed using Athena or Lambda.**

**Example:**

**Collected CloudTrail and ALB logs in an S3 bucket and queried them using Athena for auditing.**

**✅ 6. Disaster Recovery (DR)**

* **Keep copies of important data in another region using Cross-Region Replication (CRR).**
* **Ensures business continuity in case of a region failure.**

**Example:**

**Configured cross-region replication from ap-south-1 to us-east-1 for DR compliance.**

**✅ 7. Software and Content Distribution**

* **Store downloadable files like installers, reports, videos, etc.**
* **Integrated with CloudFront for faster global access.**

**Example:**

**Used S3 to store app update files, distributed via CloudFront to end users.**

**✅ 8. Serverless Application Storage**

* **Used by AWS Lambda, Step Functions, and API Gateway as storage backend.**
* **Store code packages, event data, or function outputs.**

**Example:**

**Lambda function triggered on S3 file upload to process images and store results back in another bucket.**

**✅ 9. Machine Learning / AI Data Source**

* **S3 stores training data (images, CSV, JSON) used by SageMaker and other ML tools.**

**Example:**

**Trained ML model on data stored in S3 using SageMaker; output model artifacts also stored in S3.**

**✅ 10. Versioning and Audit Trails**

* **Enable versioning to retain older file versions.**
* **Helpful to recover accidentally deleted or overwritten files.**

**Example:**

**Enabled versioning to maintain multiple versions of configuration files used by app servers.**

**🔹 3️⃣ Additional Best Practices (Optional in Interviews)**

| **Feature** | **Usage** |
| --- | --- |
| **Lifecycle Policies** | **Move old data to cheaper tiers automatically** |
| **Replication** | **Maintain DR copy in another region** |
| **Encryption (SSE, KMS)** | **Secure data at rest** |
| **Bucket Policies / ACLs** | **Control access** |
| **S3 Transfer Acceleration** | **Speed up uploads from global users** |
| **Object Lock** | **Prevent accidental or malicious deletes** |

**✅ Sample Answer (Interview-Ready):**

**“In our project, we used Amazon S3 for multiple purposes.  
Primarily, we stored application backups and logs in S3.  
We also used it as a data lake for analytics, integrating it with AWS Glue and Athena for querying logs.  
Additionally, we hosted a static website using S3 with CloudFront for global delivery.  
To optimize cost, we applied lifecycle policies to move older backups to Glacier and enabled versioning for recovery.  
Cross-region replication was also configured for disaster recovery.”**

**165\* what are the use case of lambda. how you have used in your company**

**✅ Step 1: Core Idea**

**AWS Lambda is a serverless compute service that automatically runs code in response to events (like S3 uploads, API calls, DynamoDB changes, or CloudWatch triggers) — without provisioning or managing servers.**

**✅ Step 2: Common Use Cases of Lambda**

| **Use Case** | **Description** |
| --- | --- |
| **1️⃣ Event-driven file processing** | **When a file is uploaded to S3, a Lambda function gets triggered to process, compress, or move the file.** |
| **2️⃣ Scheduled jobs (automation)** | **Lambda triggered by CloudWatch Events for periodic tasks — e.g., stopping/starting EC2s to save cost.** |
| **3️⃣ Serverless APIs** | **Combine API Gateway + Lambda to build backend REST APIs without EC2.** |
| **4️⃣ Notification workflows** | **Lambda integrates with SNS/SQS to handle asynchronous events or send alerts.** |
| **5️⃣ Real-time data processing** | **Used with Kinesis Streams or DynamoDB Streams for real-time analytics.** |
| **6️⃣ Infrastructure automation** | **Run Lambda during CloudFormation stack events or automate tagging of resources.** |

**✅ Step 3: Example — How *You* Used It in Your Company**

**In our company, we used AWS Lambda for several automation and backend processes:**

**🏢 Example 1: S3 Event Processing**

**Whenever our application team uploaded log files to an S3 bucket, a Lambda function was triggered to:**

* **Parse and filter the logs**
* **Store summary data into DynamoDB**
* **Send notification via SNS if error rates exceeded threshold**

**🧰 Services used: S3 → Lambda → DynamoDB → SNS**

**🏢 Example 2: Scheduled EC2 Stop/Start**

**To reduce costs, we scheduled a Lambda function (triggered via CloudWatch Event) to stop EC2 instances at night and start them in the morning automatically.**

**🧰 Services used: CloudWatch → Lambda → EC2 API**

**🏢 Example 3: Serverless REST API**

**We built internal APIs using API Gateway + Lambda (Python) to fetch data from RDS and return JSON responses to the frontend, without needing an EC2 backend.**

**🧰 Services used: API Gateway → Lambda → RDS**

**🗣️ Short Interview Answer:**

**In my company, we used AWS Lambda mainly for automation and event-based processing.  
For example, we had a Lambda function that triggered when files were uploaded to S3 — it validated and processed those files automatically.  
We also used Lambda with CloudWatch for cost optimization (auto stop/start EC2s) and with API Gateway to build serverless APIs.**

**Would you like me to tailor your own project-based answer (**

**166\* explain complete setup of VPC how you are configure**

**A Virtual Private Cloud (VPC) is a logically isolated network within AWS, where you launch and manage all your resources (EC2, RDS, ELB, etc.) securely.**

**You can fully control:**

* **IP address ranges**
* **Subnets**
* **Route tables**
* **Internet access (via IGW or NAT)**
* **Security (via SGs & NACLs)**

**🔹 2️⃣ VPC Setup — Step-by-Step Configuration**

**Here’s the typical production-grade setup ⬇️**

**✅ Step 1: Create a VPC**

* **Go to VPC Dashboard → Create VPC**
* **Choose CIDR block, e.g. 10.0.0.0/16 (gives 65,536 IPs)**
* **Enable DNS hostnames if you’ll use public instances.**

**🗒 Example:**

**VPC Name: my-app-vpc**

**CIDR: 10.0.0.0/16**

**✅ Step 2: Create Subnets**

**Divide your VPC into smaller networks:**

| **Subnet Type** | **Example CIDR** | **Purpose** |
| --- | --- | --- |
| **Public Subnet 1** | **10.0.1.0/24** | **Web servers / ALB** |
| **Public Subnet 2** | **10.0.2.0/24** | **Redundancy in another AZ** |
| **Private Subnet 1** | **10.0.3.0/24** | **App / DB servers** |
| **Private Subnet 2** | **10.0.4.0/24** | **High availability (other AZ)** |

**💡 Use different Availability Zones (e.g., ap-south-1a, ap-south-1b) for HA.**

**✅ Step 3: Create Internet Gateway (IGW)**

* **Create an Internet Gateway and attach it to your VPC.**
* **This allows public subnet instances to access the internet.**

**igw-12345 → attached to my-app-vpc**

**✅ Step 4: Create Route Tables**

**Create two route tables:**

1. **Public Route Table**
   * **Add route:  
     0.0.0.0/0 → Internet Gateway**
   * **Associate this route table with public subnets.**
2. **Private Route Table**
   * **Add route:  
     0.0.0.0/0 → NAT Gateway (for outbound-only internet access)**
   * **Associate with private subnets.**

**✅ Step 5: Create NAT Gateway (for Private Subnets)**

* **Launch a NAT Gateway in one of the public subnets.**
* **Allocate an Elastic IP for it.**
* **This allows private instances to connect to the internet (for updates, etc.) but keeps them inaccessible from outside.**

**✅ Step 6: Configure Security**

* **Security Groups (SGs): instance-level firewalls**
  + **Example:**
    - **Web-SG: Allow inbound 80/443 from anywhere**
    - **App-SG: Allow inbound 8080 only from Web-SG**
    - **DB-SG: Allow inbound 3306 only from App-SG**
* **Network ACLs (NACLs): subnet-level firewalls (optional fine-grain control)**
  + **Usually kept stateless — define both inbound & outbound rules.**

**✅ Step 7: Launch EC2 Instances**

* **Launch EC2s:**
  + **Web servers → Public subnet (with public IP)**
  + **App/DB servers → Private subnet (no public IP)**
* **Attach appropriate security groups.**

**✅ Step 8: Configure Load Balancer**

* **Create an Application Load Balancer (ALB) in public subnets.**
* **Target group → private subnet EC2 instances.**
* **Route incoming traffic securely.**

**✅ Step 9: (Optional) Add VPC Endpoints**

* **If private instances need to access S3/DynamoDB → create VPC endpoints.**
  + **Keeps traffic within AWS network.**

**✅ Step 10: Monitoring & Logs**

* **Enable VPC Flow Logs to track traffic (store in CloudWatch or S3).**
* **Use CloudWatch metrics for network health & instance performance.**

**🔹 3️⃣ Example Architecture Summary**

| **Component** | **Purpose** |
| --- | --- |
| **VPC** | **10.0.0.0/16 network** |
| **Public Subnets** | **ALB + NAT** |
| **Private Subnets** | **App + DB** |
| **IGW** | **Internet access for public** |
| **NAT** | **Outbound internet for private** |
| **SGs** | **Controlled access between tiers** |
| **VPC Endpoints** | **Private AWS service access** |

**✅ Short, Interview-Ready Answer**

**“I configured a VPC with CIDR 10.0.0.0/16 and created 4 subnets across 2 Availability Zones — two public and two private.  
Public subnets have an Internet Gateway for the ALB and NAT Gateway, while private subnets host application and database servers.  
Route tables are configured accordingly, and security groups enforce least-privilege access between tiers.**

**This design ensures high availability, scalability, and security.”**

**167\* s3 is general, why we have to go for cross region.**

**Even though S3 is a global service, your data in a bucket is stored in one AWS Region only.  
If you want copies of that data in another AWS Region, you must use Cross-Region Replication (CRR).**

**🧠 Detailed Explanation**

**🔹 1️⃣ S3 Is a Global Service, but Buckets Are Regional**

* **S3 is globally available as a service, but your buckets and objects live in a specific region.**
* **Example:**
  + **Your bucket is in us-east-1 (Virginia).**
  + **The data is physically stored only in Virginia, not in other regions.**
* **So, if that region faces an outage, your data might not be accessible.**

**✅ CRR replicates your data automatically to another region for backup or disaster recovery.**

**🔹 2️⃣ Reasons to Use Cross-Region Replication**

| **Reason** | **Description** |
| --- | --- |
| **Disaster Recovery (DR)** | **Keeps a backup copy of your bucket in another AWS Region. If one region fails, data is still available.** |
| **Latency Optimization** | **Users in different geographic locations can access data faster from a nearby region.** |
| **Compliance & Data Residency** | **Some laws or company policies require data copies in specific countries or regions.** |
| **Analytics & Multi-Region Processing** | **Replicate data for processing or analytics in a different AWS Region without transferring manually.** |

**🔹 3️⃣ How It Works**

1. **Enable Versioning on both source and destination buckets.**
2. **Go to S3 → Management tab → Replication → Add rule.**
3. **Choose Cross-Region Replication.**
4. **Select Destination Region and Bucket.**
5. **Assign an IAM Role that allows replication.**
6. **(Optional) Enable replication of existing objects.**

**After that, whenever you upload an object to the source bucket, it’s automatically replicated to the destination bucket.**

**🔐 Important Notes**

* **Replication is asynchronous (not instant).**
* **You can also replicate within the same region using Same-Region Replication (SRR).**
* **You can filter which objects to replicate (by prefix, tag, etc.).**
* **You can encrypt replicated data using KMS keys.**

**⚡ Example Use Case**

**You have a primary bucket in Mumbai (ap-south-1) and replicate to Singapore (ap-southeast-1) for disaster recovery and global user access.**

**🧠 Interview Short Answer**

**“Although S3 is a global service, each bucket stores data in one region only.  
We use Cross-Region Replication to automatically copy data to another AWS region for disaster recovery, latency reduction, compliance, or backup.”**

**168\* Difference between public subnet and private subnet**

**A subnet is a range of IP addresses within your VPC (Virtual Private Cloud).  
You divide your VPC (like 10.0.0.0/16) into smaller subnets for better control and isolation.**

**Subnets can be:**

* **Public subnet**
* **Private subnet**

**☀️ 1️⃣ Public Subnet**

**🔹 Definition:**

**A public subnet is a subnet whose traffic can reach the Internet directly.**

**🔹 Key Requirement:**

**The subnet’s route table must have a route to the Internet Gateway (IGW).**

**Example Route Table Entry:**

**Destination: 0.0.0.0/0 → Target: igw-123456**

**🔹 Resources Commonly Placed:**

* **Web servers (EC2 hosting websites)**
* **Load balancers (ALB)**
* **Bastion hosts (for SSH access)**

**🔹 Example:**

**User → Internet → Internet Gateway → EC2 in Public Subnet**

**🔹 Need:**

**Used when resources must be accessible from the internet.**

**🌑 2️⃣ Private Subnet**

**🔹 Definition:**

**A private subnet is a subnet that cannot access the internet directly.**

**🔹 Key Point:**

* **No route to the Internet Gateway**
* **Optionally can use NAT Gateway (in public subnet) to access internet outbound only (for updates, patches, etc.)**

**Example Route Table Entry:**

**Destination: 0.0.0.0/0 → Target: nat-gw-123456**

**🔹 Resources Commonly Placed:**

* **Databases (RDS)**
* **Application servers (EC2 that don’t need public access)**
* **Internal microservices**

**🔹 Example:**

**App Server (Private Subnet) → NAT Gateway → Internet  
(but internet → App Server ❌ not allowed)**

**🧠 3️⃣ Key Differences Summary Table**

| **Feature** | **Public Subnet** | **Private Subnet** |
| --- | --- | --- |
| **Internet Access** | **Yes (via Internet Gateway)** | **No direct access** |
| **Route Table** | **Has route to IGW** | **No IGW route; may have NAT** |
| **Inbound Traffic** | **Can receive from Internet** | **Cannot receive directly** |
| **Outbound Internet** | **Direct access** | **Via NAT Gateway or NAT Instance** |
| **Use Case** | **Web/App servers needing public access** | **DB/Backend servers needing isolation** |
| **IP Type** | **Public IP or Elastic IP** | **Private IP only** |

**🧩 4️⃣ Simple Architecture Example**

**VPC (10.0.0.0/16)**

**├── Public Subnet (10.0.1.0/24)**

**│ ├── Internet Gateway (IGW)**

**│ ├── NAT Gateway**

**│ └── EC2 (Web Server)**

**│**

**└── Private Subnet (10.0.2.0/24)**

**├── RDS (Database)**

**└── EC2 (App Server)**

**💡 The private subnet instances connect outbound through NAT Gateway in the public subnet, but they can’t be accessed from the internet directly.**

**✅ In Short**

| **Public Subnet** | **Private Subnet** |
| --- | --- |
| **Accessible from internet** | **Not accessible directly** |
| **Has route to IGW** | **No route to IGW** |
| **Used for frontend/web servers** | **Used for backend/databases** |

**169\* What is ENI**

**ENI stands for Elastic Network Interface.  
It is a virtual network interface card (NIC) that you can attach to an EC2 instance within a VPC (Virtual Private Cloud).**

**In simple terms:**

**ENI = Virtual adapter that allows your EC2 instance to connect to a VPC network.**

**🟢 2️⃣ What It Contains**

**An ENI includes several network attributes:**

| **Component** | **Description** |
| --- | --- |
| **Primary private IPv4 address** | **Main IP assigned to the instance** |
| **Secondary private IPv4 addresses** | **Optional additional private IPs** |
| **Elastic IP (public IP)** | **Can be associated with a private IP** |
| **MAC address** | **Unique hardware address** |
| **Security groups** | **One or more attached SGs for inbound/outbound rules** |
| **IPv6 addresses** | **Optional** |
| **Network interface ID** | **Unique identifier for the ENI** |

**🟢 3️⃣ Key Points**

* **Each EC2 instance automatically gets one primary ENI (eth0).**
* **You can attach multiple ENIs to the same EC2 instance (depending on instance type).**
* **You can detach and reattach ENIs between instances (in the same AZ).**
* **You can create standalone ENIs (not yet attached) and preconfigure IPs and SGs.**

**🟢 4️⃣ Common Use Cases**

| **Use Case** | **Explanation** |
| --- | --- |
| **High Availability (HA)** | **Move ENI from one instance to another in case of failure.** |
| **Multi-Network Setup** | **Connect one instance to multiple subnets or VPCs.** |
| **Security Isolation** | **Assign different security groups for traffic separation.** |
| **Monitoring / Logging** | **Attach a separate ENI for management or monitoring tools.** |

**🟢 5️⃣ Example Scenario**

**You have two ENIs attached to an EC2 instance:**

* **eth0 → for public internet access**
* **eth1 → for internal database communication**

**If eth0’s instance fails, you can detach the ENI and attach it to another instance, keeping the same IP and network identity — ensuring quick failover.**

**🟢 6️⃣ AWS CLI Example**

**To describe ENIs:**

**aws ec2 describe-network-interfaces**

**To attach ENI to an instance:**

**aws ec2 attach-network-interface --network-interface-id eni-12345678 --instance-id i-01234567 --device-index 1**

**✅ Sample Interview Answer**

**“ENI stands for Elastic Network Interface.  
It’s a virtual network card that allows an EC2 instance to connect to a VPC.  
Each ENI has attributes like a private IP, optional public IP, MAC address, and security groups.**

**Every EC2 instance has one primary ENI, and we can attach additional ENIs for multi-network or failover setups.  
ENIs are useful for high availability because they can be detached from one instance and reattached to another in the same Availability Zone without changing IP configuration.”**

**170\* How to connect two vpc**

**🔹 Ways to Connect Two VPCs**

| **Method** | **Description** | **Use Case** |
| --- | --- | --- |
| **1. VPC Peering** | **Direct connection between two VPCs using private IPs** | **Simple communication between VPCs (same or different regions)** |
| **2. Transit Gateway (TGW)** | **Central hub that connects multiple VPCs** | **Large multi-VPC architectures** |
| **3. PrivateLink (VPC Endpoint Service)** | **Connects services privately via ENI** | **Sharing services securely (no route changes)** |
| **4. VPN Connection** | **Connects VPCs via IPsec tunnels** | **Secure encrypted traffic across regions/accounts** |
| **5. Inter-Region Peering** | **Like peering but between regions** | **Multi-region redundancy / DR setup** |

**🔸 1️⃣ VPC Peering (Most Common)**

**✅ How It Works**

* **You create a peering connection between VPC A and VPC B.**
* **Add routes in both VPC route tables to send traffic through the peering connection.**
* **Instances can communicate using private IP addresses.**

**⚙️ Steps**

1. **Go to VPC Console → Peering Connections → Create Peering Connection**
   * **Choose Requester VPC (A) and Accepter VPC (B).**
   * **Accept the request from the other side.**
2. **Update Route Tables in both VPCs:**
   * **In VPC A route table → Destination = CIDR of VPC B → Target = Peering connection.**
   * **In VPC B route table → Destination = CIDR of VPC A → Target = Peering connection.**
3. **Update Security Groups and Network ACLs to allow traffic.**

**⚠️ Limitations**

* **Non-transitive — if A↔B and B↔C exist, A can’t reach C.**
* **CIDR blocks must not overlap.**
* **No support for edge-to-edge routing (can’t connect through VPN/Direct Connect).**

**🔸 2️⃣ Transit Gateway (TGW)**

**✅ How It Works**

* **A central router that connects multiple VPCs, on-prem, or VPNs.**
* **You attach each VPC to the Transit Gateway.**
* **Routing is centralized in TGW.**

**⚙️ Use Case**

* **You have many VPCs and want to manage connectivity easily.**
* **For hub-and-spoke architecture.**

**⚙️ Steps**

1. **Create a Transit Gateway.**
2. **Create VPC Attachments for each VPC.**
3. **Update TGW Route Tables and VPC route tables to allow communication.**

**💰 Pricing**

* **You pay per attachment and data processed.**

**🔸 3️⃣ AWS PrivateLink**

**✅ How It Works**

* **Allows private service access between VPCs.**
* **Traffic goes through ENIs, not the internet.**
* **No route table changes needed.**

**⚙️ Use Case**

* **When you want to expose a service (e.g., API, database) securely to another VPC.**

**🔸 4️⃣ VPN Connection Between VPCs**

**✅ How It Works**

* **Use AWS Site-to-Site VPN.**
* **Create virtual private gateways or transit gateways on each side.**
* **Traffic encrypted via IPsec tunnel.**

**⚙️ Use Case**

* **Secure communication between VPCs across regions or accounts.**

**🔹 Example: VPC Peering Setup**

| **Component** | **VPC A** | **VPC B** |
| --- | --- | --- |
| **CIDR** | **10.0.0.0/16** | **172.31.0.0/16** |
| **Route Table** | **172.31.0.0/16 → pcx-xxxx** | **10.0.0.0/16 → pcx-xxxx** |
| **Security Group** | **Allow inbound from 172.31.0.0/16** | **Allow inbound from 10.0.0.0/16** |

**✅ Once configured — instances in both VPCs can ping or SSH each other using private IPs.**

**✅ Summary Table**

| **Method** | **Transitive** | **Cross-Region** | **Encryption** | **Best For** |
| --- | --- | --- | --- | --- |
| **VPC Peering** | **❌** | **✅** | **❌** | **Simple 1-to-1 connectivity** |
| **Transit Gateway** | **✅** | **✅** | **❌** | **Multi-VPC hub-spoke** |
| **PrivateLink** | **❌** | **✅** | **✅** | **Exposing services privately** |
| **VPN** | **✅** | **✅** | **✅** | **Secure connection via IPsec** |

**171\* How to upload files to S3 bucket**

**🖥️ 1. Using AWS Management Console (GUI method)**

**Steps:**

1. **Go to AWS Management Console → S3**
2. **Open your S3 bucket**
3. **Click “Upload”**
4. **Click Add files or Add folder**
5. **Choose your files → click Upload**
6. **You can set permissions (e.g., public/private) and storage class during upload.**

**✅ Best for: Quick manual uploads.**

**💻 2. Using AWS CLI (Command Line Interface)**

**First, configure your CLI:**

**aws configure**

**Then upload a file:**

**aws s3 cp myfile.txt s3://my-bucket-name/**

**Upload an entire folder:**

**aws s3 cp /home/ec2-user/myfolder s3://my-bucket-name/ --recursive**

**✅ Best for: Developers and automation scripts.**

**🧰 3. Using AWS SDKs (Python, Java, Node.js, etc.)**

**Example – Python (Boto3):**

**import boto3**

**s3 = boto3.client('s3')**

**s3.upload\_file('localfile.txt', 'my-bucket-name', 'remote-file.txt')**

**✅ Best for: Application-level uploads or automation.**

**🔄 4. Using AWS CLI with Sync (for multiple files)**

**To sync a local folder to S3:**

**aws s3 sync ./myfolder s3://my-bucket-name/**

**This uploads only new or changed files.**

**⚙️ 5. Using Automation / Services**

* **AWS Lambda: Upload files automatically when triggered by another event.**
* **AWS SDK / API Gateway: For user uploads through a web app.**
* **Jenkins / CI-CD: Store build artifacts into S3.**

**🔐 Security Notes**

* **Ensure correct IAM permissions:**
* **{**
* **"Effect": "Allow",**
* **"Action": ["s3:PutObject", "s3:PutObjectAcl"],**
* **"Resource": "arn:aws:s3:::my-bucket-name/\*"**
* **}**
* **Use Bucket Policy or IAM Role instead of access keys.**
* **For private connections → use VPC endpoint for S3.**

**🗣️ Short Interview Answer:**

**I can upload files to S3 using the AWS Console, AWS CLI, or SDKs like Boto3.  
For example, with CLI I use aws s3 cp myfile.txt s3://bucket-name/.  
I also ensure the user or EC2 instance has proper IAM permissions (s3:PutObject) before uploading.**

**172\* How do you give access to S3 buckets**

**🔹 1️⃣ Ways to Give Access to an S3 Bucket**

**There are multiple methods to grant access to an S3 bucket depending on *who or what* needs access.**

| **Access Type** | **Used For** | **Where Configured** |
| --- | --- | --- |
| **IAM Policy** | **To allow AWS users, roles, or groups to access S3** | **In IAM console** |
| **Bucket Policy** | **To allow cross-account or public access** | **In S3 bucket permissions** |
| **ACL (Access Control List)** | **Legacy method, used rarely now** | **On individual buckets or objects** |
| **Presigned URL** | **Temporary, time-limited access to files** | **Generated via AWS SDK/CLI** |
| **VPC Endpoint Policy** | **Restricts access from specific VPCs** | **In VPC endpoint configuration** |

**🔹 2️⃣ Granting Access Using IAM Policy (Best Practice)**

**✅ Steps:**

1. **Go to IAM → Users / Roles → Add Permissions**
2. **Attach a policy like:**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": [**

**"s3:ListBucket",**

**"s3:GetObject",**

**"s3:PutObject"**

**],**

**"Resource": [**

**"arn:aws:s3:::my-bucket",**

**"arn:aws:s3:::my-bucket/\*"**

**]**

**}**

**]**

**}**

1. **This gives access only to the specific bucket.**

**✅ Use IAM users/roles for internal or application access.**

**🔹 3️⃣ Granting Access Using Bucket Policy**

**Use this when you need to:**

* **Allow cross-account access, or**
* **Make public or restricted access to external entities.**

**✅ Example: Give Access to Another AWS Account**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Sid": "CrossAccountAccess",**

**"Effect": "Allow",**

**"Principal": { "AWS": "arn:aws:iam::123456789012:root" },**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-bucket",**

**"arn:aws:s3:::my-bucket/\*"**

**]**

**}**

**]**

**}**

**✅ Used when another AWS account or service needs access to your bucket.**

**🔹 4️⃣ Presigned URL (Temporary Access)**

**If you want to give temporary access to download or upload a file:**

**CLI Example:**

**aws s3 presign s3://my-bucket/data.txt --expires-in 3600**

**This URL gives 1-hour access to the file — very useful for limited-time file sharing.**

**🔹 5️⃣ Using S3 Access Points (for Large Environments)**

**If many users or applications need different access levels:**

* **Create Access Points for the bucket.**
* **Attach specific IAM policies per access point.**

**🔹 6️⃣ Restricting Public Access (Important!)**

**Always ensure Block Public Access is enabled unless public access is intentionally needed (like a public website).**

**Path:  
S3 → Bucket → Permissions → Block Public Access**

**🔹 7️⃣ Example Interview Answer**

**“There are several ways to give access to an S3 bucket depending on the use case.  
For internal users or applications, I use IAM policies attached to roles or users.  
For cross-account or external access, I use bucket policies.  
For temporary access, I generate presigned URLs.**

**I always make sure to follow the principle of least privilege and keep Block Public Access enabled unless required.”**

**173\* Different kinds of load balancer**

**✅ Answer:**

**AWS offers three main types of Load Balancers under the Elastic Load Balancing (ELB) service:**

| **Type** | **Layer** | **Description** | **Best For** |
| --- | --- | --- | --- |
| **1. Application Load Balancer (ALB)** | **Layer 7 (Application Layer)** | **Routes traffic based on HTTP/HTTPS, path, host, or header. Supports microservices, containers, and WebSocket.** | **Web apps, APIs, containerized workloads (ECS/EKS)** |
| **2. Network Load Balancer (NLB)** | **Layer 4 (Transport Layer)** | **Routes based on TCP/UDP and handles millions of requests per second with ultra-low latency.** | **Real-time apps, gaming, IoT, or when needing static IPs** |
| **3. Gateway Load Balancer (GWLB)** | **Layer 3 (Network Layer)** | **Distributes traffic to third-party virtual appliances like firewalls, intrusion detection systems, etc.** | **Network inspection, security appliances** |

**1️⃣ Application Load Balancer (ALB)**

* **Works at Layer 7 (HTTP/S).**
* **Can route traffic:**
  + **Based on URL path (e.g., /api/\*, /images/\*)**
  + **Based on host (e.g., api.myapp.com, www.myapp.com)**
* **Supports:**
  + **Sticky sessions**
  + **WebSockets**
  + **AWS WAF (Web Application Firewall)**
  + **Target groups (EC2, ECS, Lambda)**

**2️⃣ Network Load Balancer (NLB)**

* **Works at Layer 4 (TCP, UDP, TLS).**
* **Handles very high performance and low latency workloads.**
* **Provides static IP addresses or Elastic IPs.**
* **Common use cases:**
  + **Real-time financial apps**
  + **Gaming**
  + **VoIP / IoT**
  + **TLS termination with high performance**

**3️⃣ Gateway Load Balancer (GWLB)**

* **Works at Layer 3 (Network).**
* **Designed to deploy, scale, and manage network appliances.**
* **Combines transparency (gateway) + scaling (load balancer).**
* **Example: Send traffic through firewalls or security inspection appliances before reaching backend.**

**Bonus: Classic Load Balancer (CLB)**

**✅ Note: CLB is legacy and being replaced by ALB/NLB.**

* **Operates at Layer 4 & 7.**
* **Used for simple, older apps.**
* **Not recommended for new deployments.**

**Quick Summary Table**

| **Feature** | **ALB** | **NLB** | **GWLB** | **CLB (Legacy)** |
| --- | --- | --- | --- | --- |
| **OSI Layer** | **7** | **4** | **3** | **4 & 7** |
| **Protocols** | **HTTP/HTTPS** | **TCP/UDP/TLS** | **IP** | **HTTP/TCP** |
| **Routing** | **Path/Host-based** | **IP/Port-based** | **Transparent** | **Basic** |
| **Static IP** | **❌** | **✅** | **✅** | **❌** |
| **Target Types** | **EC2, ECS, Lambda** | **EC2, IP** | **Appliance** | **EC2** |
| **WAF Support** | **✅** | **❌** | **❌** | **❌** |

**Interview Short Answer:**

**“AWS provides three main types of load balancers — Application (Layer 7), Network (Layer 4), and Gateway (Layer 3).  
ALB is for HTTP/HTTPS apps, NLB for high-performance TCP/UDP traffic, and GWLB for routing traffic through security appliances. Classic Load Balancer is legacy and not used for new applications.”**

**174\* Difference between alb n nlb**

**⚖️ ALB vs NLB — Quick Comparison**

| **Feature** | **Application Load Balancer (ALB)** | **Network Load Balancer (NLB)** |
| --- | --- | --- |
| **Layer (OSI Model)** | **Layer 7 – *Application Layer*** | **Layer 4 – *Transport Layer*** |
| **Protocols Supported** | **HTTP, HTTPS, WebSocket** | **TCP, UDP, TLS** |
| **Routing Type** | **Based on application data — e.g., URL, host header, path, or query string** | **Based on IP address and port only** |
| **Use Case** | **Web applications, microservices, APIs** | **High-performance, low-latency apps (gaming, IoT, databases)** |
| **Latency** | **Slightly higher (content-based routing)** | **Extremely low (connection-based)** |
| **Targets** | **EC2, ECS, EKS, Lambda (Layer 7 aware)** | **EC2, IP addresses, container pods, private link** |
| **Static IP Support** | **❌ No static IP (uses DNS name)** | **✅ Yes (each AZ gets one static IP)** |
| **TLS Termination** | **✅ Supported (can manage SSL certs via ACM)** | **✅ Supported** |
| **Sticky Sessions** | **✅ Supported (via cookies)** | **✅ Supported (via source IP)** |
| **Cross-Zone Load Balancing** | **✅ Enabled by default** | **Optional (disabled by default)** |
| **Logging** | **Access logs (detailed HTTP info)** | **Flow logs only** |
| **Integration with WAF** | **✅ Yes** | **❌ No (directly not supported)** |
| **Pricing** | **Slightly higher for Layer 7 processing** | **Slightly cheaper per connection** |

**🧠 In Simple Terms**

| **Scenario** | **Recommended LB** |
| --- | --- |
| **HTTP/HTTPS traffic** | **ALB** |
| **Need routing by URL path or domain** | **ALB** |
| **Need to load balance non-HTTP protocols (TCP/UDP)** | **NLB** |
| **Need static IPs** | **NLB** |
| **Very high performance, low latency** | **NLB** |
| **Integrate with WAF** | **ALB** |

**🧩 Example Use Cases**

**ALB**

* **Web apps and APIs (e.g., api.example.com → backend EC2s)**
* **Microservices (path-based or host-based routing)**
* **HTTPS termination (ACM certificates)**

**NLB**

* **Database connections (MySQL, RDP, etc.)**
* **Gaming, financial applications**
* **IoT or custom TCP/UDP protocols**
* **When you need static IP addresses**

**📈 Architecture Example**

**Internet**

**│**

**┌────────┴────────┐**

**│ ALB (L7) │**

**│ HTTP/HTTPS │**

**└────────┬────────┘**

**│**

**EC2 / ECS / Lambda**

**Internet**

**│**

**┌────────┴────────┐**

**│ NLB (L4) │**

**│ TCP/UDP/TLS │**

**└────────┬────────┘**

**│**

**EC2**

**175\* Difference between TCP and UDP**

| **Feature** | **TCP (Transmission Control Protocol)** | **UDP (User Datagram Protocol)** |
| --- | --- | --- |
| **Connection type** | **Connection-oriented (requires handshake)** | **Connectionless (no handshake)** |
| **Reliability** | **Reliable – ensures data delivery, retransmits lost packets** | **Unreliable – no retransmission if packet lost** |
| **Data transfer** | **Data sent as a stream of bytes** | **Data sent as independent packets (datagrams)** |
| **Order of delivery** | **Guarantees order of packets** | **No guarantee of order** |
| **Error checking** | **Performs error checking and correction** | **Performs only basic checksum** |
| **Speed** | **Slower (because of connection setup, acknowledgment, and retransmission)** | **Faster (no connection or acknowledgment overhead)** |
| **Header size** | **Larger (20 bytes minimum)** | **Smaller (8 bytes)** |
| **Use cases** | **When reliability matters: web (HTTP/HTTPS), email (SMTP), file transfer (FTP, SFTP), SSH** | **When speed matters: streaming (YouTube), DNS, VoIP, gaming** |
| **Handshake** | **Uses 3-way handshake (SYN, SYN-ACK, ACK)** | **No handshake** |
| **Flow control & congestion control** | **Yes** | **No** |

**🧠 Example:**

**TCP example:**

**When you open a website:**

**Client → SYN → Server**

**Server → SYN-ACK → Client**

**Client → ACK → Server**

**Then the connection is established → data starts flowing (HTTP, HTTPS).**

**UDP example:**

**When you do a DNS lookup or video streaming:**

**Client → Request → Server**

**Server → Response → Client**

**No acknowledgment, just fast one-way communication.**

**✅ Summary:**

| **TCP** | **UDP** |
| --- | --- |
| **Reliable, ordered, slower** | **Unreliable, unordered, faster** |
| **Connection required** | **No connection** |
| **Use for web, file, email** | **Use for DNS, streaming, gaming** |

**176\* terraform for AWS lambda**

**Terraform is an Infrastructure as Code (IaC) tool that lets you define and provision cloud resources (like Lambda, S3, IAM, etc.) in a declarative way using .tf files.**

**You can automate, version-control, and reproduce your AWS infrastructure easily.**

**🔹 2️⃣ Use Case: Deploying AWS Lambda with Terraform**

**You can use Terraform to:**

* **Create a Lambda function**
* **Attach IAM role/policy**
* **Optionally create a trigger (like S3, API Gateway, or EventBridge)**
* **Upload Lambda code (from local ZIP or S3)**

**🔹 3️⃣ Basic Terraform Configuration for AWS Lambda**

**Here’s an example .tf configuration:**

**# ---- Provider ----**

**provider "aws" {**

**region = "us-east-1"**

**}**

**# ---- IAM Role for Lambda ----**

**resource "aws\_iam\_role" "lambda\_role" {**

**name = "lambda\_basic\_execution\_role"**

**assume\_role\_policy = jsonencode({**

**Version = "2012-10-17"**

**Statement = [{**

**Action = "sts:AssumeRole"**

**Effect = "Allow"**

**Principal = {**

**Service = "lambda.amazonaws.com"**

**}**

**}]**

**})**

**}**

**# ---- IAM Policy Attachment ----**

**resource "aws\_iam\_role\_policy\_attachment" "lambda\_policy" {**

**role = aws\_iam\_role.lambda\_role.name**

**policy\_arn = "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"**

**}**

**# ---- Lambda Function ----**

**resource "aws\_lambda\_function" "my\_lambda" {**

**function\_name = "my-lambda-func"**

**handler = "lambda\_function.lambda\_handler"**

**runtime = "python3.9"**

**role = aws\_iam\_role.lambda\_role.arn**

**filename = "lambda\_function\_payload.zip" # local zip file**

**source\_code\_hash = filebase64sha256("lambda\_function\_payload.zip")**

**environment {**

**variables = {**

**ENV = "dev"**

**}**

**}**

**}**

**🔹 4️⃣ Folder Structure Example**

**terraform/**

**│**

**├── main.tf**

**├── variables.tf**

**├── outputs.tf**

**└── lambda\_function/**

**└── lambda\_function.py**

**Your lambda\_function.py might look like:**

**def lambda\_handler(event, context):**

**return {**

**'statusCode': 200,**

**'body': 'Hello from Lambda!'**

**}**

**Then zip it:**

**cd lambda\_function**

**zip ../lambda\_function\_payload.zip lambda\_function.py**

**🔹 5️⃣ Commands to Deploy**

**terraform init # Initialize provider and plugins**

**terraform plan # Preview resources to be created**

**terraform apply # Create Lambda function**

**After deployment, Terraform will output:**

* **Lambda ARN**
* **IAM Role ARN**

**🔹 6️⃣ Add Trigger (Example: S3 or API Gateway)**

**Example: Trigger from S3 upload**

**resource "aws\_s3\_bucket" "lambda\_bucket" {**

**bucket = "my-lambda-trigger-bucket"**

**}**

**resource "aws\_s3\_bucket\_notification" "lambda\_trigger" {**

**bucket = aws\_s3\_bucket.lambda\_bucket.id**

**lambda\_function {**

**lambda\_function\_arn = aws\_lambda\_function.my\_lambda.arn**

**events = ["s3:ObjectCreated:\*"]**

**}**

**depends\_on = [aws\_lambda\_permission.allow\_s3]**

**}**

**resource "aws\_lambda\_permission" "allow\_s3" {**

**statement\_id = "AllowExecutionFromS3"**

**action = "lambda:InvokeFunction"**

**function\_name = aws\_lambda\_function.my\_lambda.function\_name**

**principal = "s3.amazonaws.com"**

**source\_arn = aws\_s3\_bucket.lambda\_bucket.arn**

**}**

**🔹 7️⃣ Key Points for Interviews**

| **Concept** | **Explanation** |
| --- | --- |
| **Provider** | **Defines which cloud (AWS, Azure, etc.) you’re using** |
| **IAM Role** | **Lambda needs an execution role to run** |
| **Handler** | **Entry point for your function (file.function\_name)** |
| **Runtime** | **Environment (e.g., python3.9, nodejs18.x)** |
| **Source Code Hash** | **Ensures Terraform detects changes in your code** |
| **Triggers** | **You can connect Lambda to S3, API Gateway, CloudWatch, etc.** |
| **State Management** | **Terraform keeps track of created resources in terraform.tfstate** |

**177\* S3 encryption**

**S3 encryption means protecting your data at rest (inside S3) or in transit (when moving to/from S3) so unauthorized users can’t read it.**

**There are two main types:**

1. **Encryption in Transit → Protects data while being transferred.**
2. **Encryption at Rest → Protects stored data inside the bucket.**

**🔸 1️⃣ Encryption in Transit (SSL/TLS)**

* **Data is encrypted when uploaded or downloaded to S3.**
* **Uses HTTPS (TLS/SSL).  
  ✅ Example:**
  + **Access via https://s3.amazonaws.com/mybucket/file.txt instead of HTTP.**

**🔸 2️⃣ Encryption at Rest**

**AWS provides four types of encryption for S3 data at rest:**

| **Type** | **Name** | **Key Management** | **Description** |
| --- | --- | --- | --- |
| **SSE-S3** | **Server-Side Encryption with Amazon S3 Managed Keys** | **AWS manages keys automatically** | **Simplest option — just enable “Default Encryption” in bucket settings.** |
| **SSE-KMS** | **Server-Side Encryption with AWS KMS Keys** | **Keys managed in AWS KMS (Customer Managed or AWS Managed)** | **Provides audit trail & fine-grained access control.** |
| **SSE-C** | **Server-Side Encryption with Customer-Provided Keys** | **You manage your own keys** | **AWS doesn’t store the key — you provide it with each request.** |
| **Client-Side Encryption** | **Encryption done before upload** | **Application encrypts data before sending to S3** | **AWS never sees unencrypted data.** |

**🔹 Server-Side Encryption (SSE)**

**✅ 1. SSE-S3 (AES-256)**

* **AWS handles encryption and key management.**
* **Uses AES-256 algorithm.**
* **Easiest to use — just enable in the S3 bucket settings or via policy.**

**Example Policy:**

**{**

**"BucketEncryption": {**

**"ServerSideEncryptionConfiguration": [**

**{**

**"ServerSideEncryptionByDefault": {**

**"SSEAlgorithm": "AES256"**

**}**

**}**

**]**

**}**

**}**

**✅ 2. SSE-KMS**

* **Uses AWS Key Management Service (KMS) to manage encryption keys.**
* **You can use AWS-managed keys (aws/s3) or your own customer-managed keys.**
* **You get key rotation, audit logs, and access control via IAM.**

**Example Policy:**

**{**

**"BucketEncryption": {**

**"ServerSideEncryptionConfiguration": [**

**{**

**"ServerSideEncryptionByDefault": {**

**"SSEAlgorithm": "aws:kms",**

**"KMSMasterKeyID": "arn:aws:kms:region:account-id:key/key-id"**

**}**

**}**

**]**

**}**

**}**

**✅ 3. SSE-C**

* **You bring your own encryption keys.**
* **AWS uses your key to encrypt the object, but doesn’t store it.**
* **You must provide the same key when retrieving the object.**

**🔹 Client-Side Encryption**

* **Encryption is done before uploading to S3.**
* **Your application (SDK or third-party tool) handles encryption/decryption.**
* **Useful if you want to ensure AWS never has access to plaintext data.**

**🔹 How to Enable Default Encryption**

**You can set encryption for all new objects in a bucket automatically:**

1. **Go to S3 Console → Bucket → Properties → Default Encryption**
2. **Choose either:**
   * **AES-256 (SSE-S3) or**
   * **AWS-KMS (select a KMS key).**
3. **Save changes — all new uploads will be encrypted by default.**

**✅ Example Scenario**

| **Requirement** | **Solution** |
| --- | --- |
| **Simple bucket encryption** | **SSE-S3** |
| **Need audit trail and key rotation** | **SSE-KMS** |
| **You own/manage keys outside AWS** | **SSE-C** |
| **Encrypt data before sending to AWS** | **Client-side encryption** |

**🔒 Summary Table**

| **Type** | **Key Managed By** | **AWS Access to Key?** | **Use Case** |
| --- | --- | --- | --- |
| **SSE-S3 (AES-256)** | **AWS** | **Yes** | **Default encryption for most buckets** |
| **SSE-KMS** | **AWS KMS / Customer** | **Controlled** | **Compliance, auditing, fine-grained access** |
| **SSE-C** | **Customer** | **No** | **Bring-your-own-key scenarios** |
| **Client-Side** | **Customer** | **No** | **End-to-end data privacy** |

**178\* how other team member access S3**

**Other team members can access the S3 bucket in several secure ways, depending on their role and permissions.  
The main goal is to ensure secure, controlled access using IAM policies, bucket policies, or roles.**

**🔐 1. Using IAM Users and Policies**

**Each team member gets an IAM user with permissions to access the S3 bucket.**

**Steps:**

1. **Create an IAM user for each team member.**
2. **Attach an IAM policy that allows S3 access.**

**Example Policy (Read-only access):**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": ["s3:GetObject", "s3:ListBucket"],**

**"Resource": [**

**"arn:aws:s3:::my-team-bucket",**

**"arn:aws:s3:::my-team-bucket/\*"**

**]**

**}**

**]**

**}**

**✅ This lets the user view and download files but not upload or delete.**

**🧰 2. Using IAM Roles (for EC2, Lambda, etc.)**

**If your teammates use EC2 instances or applications, you can assign an IAM Role to the instance instead of user credentials.**

**Example:**

* **EC2 instance running a Jenkins server or app that needs to upload to S3**
* **Assign role with s3:PutObject permissions**

**✅ This avoids sharing credentials manually.**

**🪣 3. Using Bucket Policy (for Cross-Account or Group Access)**

**If team members are from another AWS account, you can allow access using a Bucket Policy.**

**Example – Allow access from another AWS Account:**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Principal": { "AWS": "arn:aws:iam::111122223333:root" },**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-team-bucket",**

**"arn:aws:s3:::my-team-bucket/\*"**

**]**

**}**

**]**

**}**

**✅ This lets all users in that AWS account access the bucket.**

**🔑 4. Using Pre-Signed URLs (Temporary Access)**

**For one-time or short-term access:**

**aws s3 presign s3://my-team-bucket/report.pdf --expires-in 3600**

**✅ Generates a temporary URL valid for a set duration (e.g., 1 hour).**

**🌐 5. Using VPC Endpoint (Private Access)**

**If users are in the same private VPC, you can connect to S3 privately using a Gateway Endpoint.  
Traffic stays inside AWS — no public internet access.**

**🧠 Access Control Options Summary**

| **Access Method** | **Use Case** | **Security** |
| --- | --- | --- |
| **IAM Policy** | **For individual users within AWS account** | **✅ Best practice** |
| **IAM Role** | **For EC2, Lambda, or ECS tasks** | **✅ Best practice** |
| **Bucket Policy** | **For cross-account or org-wide access** | **✅ Secure if limited** |
| **Pre-Signed URL** | **For temporary public sharing** | **⚠️ Temporary only** |
| **VPC Endpoint** | **For private network access** | **✅ Highly secure** |

**🗣️ Short Interview Answer:**

**Other team members can access the S3 bucket through IAM users or roles with specific permissions like s3:GetObject or s3:PutObject.  
For cross-account access, we use a bucket policy.  
For short-term access, we provide pre-signed URLs.  
All access is controlled by IAM and bucket policies to maintain security.**

**179\* how u have used IAM**

**IAM (Identity and Access Management) is the AWS service that helps you securely control access to AWS resources.  
It lets you manage users, groups, roles, and policies to define *who can do what* in your AWS account.**

**🔹 2️⃣ How I Have Used IAM in My Projects**

**Here’s how you can explain it practically (and interviewers love this structure):**

**✅ a. Creating and Managing Users & Groups**

* **I created IAM users for developers, testers, and DevOps engineers.**
* **Grouped them by role (e.g., *Developers*, *Admins*, *Ops*).**
* **Attached least-privilege IAM policies to each group so users only get the access they need.**

**Example:**

* **Developer → Read/Write access to S3 & CloudWatch**
* **Admin → Full access to EC2, RDS, IAM**
* **Ops → Access to CloudWatch metrics and logs only**

**✅ b. Using IAM Roles for EC2 / Lambda**

* **Instead of storing credentials on EC2 or Lambda, I used IAM roles.**
* **Attached a role with specific permissions (like AmazonS3ReadOnlyAccess) to the EC2 instance.**
* **The application running on EC2 automatically got temporary credentials to access S3 securely.**

**Example Use Case:  
EC2 instance uploads application logs to S3 using the attached IAM role.**

**✅ c. Setting Up Cross-Service Access**

* **Used IAM roles to allow one AWS service to access another securely.**
  + **Example: Lambda reading from S3 and writing to DynamoDB.**
* **Configured trust policies so that one service can “assume” another’s role.**

**✅ d. Cross-Account Access**

* **Configured IAM roles with trust relationships to allow access between two AWS accounts (e.g., dev ↔ prod).**
* **Example: A role in *Prod Account* allowed developers in *Dev Account* to assume it for temporary access.**

**✅ e. Enforcing MFA and Password Policies**

* **Enabled MFA (Multi-Factor Authentication) for IAM users to enhance security.**
* **Created strong password policies (minimum length, rotation, uppercase, numbers, etc.).**

**✅ f. Managing Access via IAM Policies**

* **Wrote custom JSON policies for fine-grained control.  
  Example:**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": ["ec2:Describe\*", "s3:ListBucket"],**

**"Resource": "\*"**

**}**

**]**

**}**

**✅ g. Using IAM with AWS CLI / SDK**

* **Configured IAM user credentials to authenticate AWS CLI commands.**
* **Used STS (Security Token Service) for temporary access tokens.**

**🔹 3️⃣ Example Interview Answer**

**“I’ve used IAM extensively for securing access across our AWS environment.  
I created users and groups with least-privilege policies, used IAM roles for EC2 and Lambda to access S3 and DynamoDB securely, and set up cross-account access using trust policies.**

**We also enforced MFA for all IAM users and monitored permissions using IAM Access Analyzer.  
I always follow the principle of least privilege to keep our environment secure.”**

**🔹 Bonus: IAM Best Practices**

* **Always use roles instead of static keys.**
* **Enable MFA for all IAM users.**
* **Review IAM policies periodically.**
* **Use AWS Organizations + SCPs for account-level control.**
* **Use IAM Access Analyzer to detect unused or overly broad permissions.**

**180\* how encryption takes place in S3 for the team user operating on S3.**

**Amazon S3 provides two main types of encryption to protect data:**

1. **Server-Side Encryption (SSE) – managed by AWS.**
2. **Client-Side Encryption (CSE) – managed by you or your application.**

**When a team uses S3, SSE with AWS KMS (SSE-KMS) is most commonly used for controlled access and auditing.**

**🔐 1. Server-Side Encryption (SSE)**

**S3 automatically encrypts data at rest after receiving it, and decrypts it on retrieval — transparently to the user.**

**Types of SSE:**

| **Type** | **Key Managed By** | **Description** | **Use Case** |
| --- | --- | --- | --- |
| **SSE-S3** | **AWS** | **S3 manages keys internally using AES-256.** | **Simple, default encryption.** |
| **SSE-KMS** | **AWS KMS (Key Management Service)** | **You control keys, IAM permissions, and audit key usage.** | **Ideal for team environments.** |
| **SSE-C** | **Customer** | **You provide your own key for each upload/download.** | **Rarely used; for strict compliance.** |

**🧠 For Teams — SSE-KMS is the Best Practice**

**When multiple users operate on an S3 bucket:**

* **You enable default encryption = SSE-KMS on the bucket.**
* **Each request to upload/download uses KMS-managed keys (CMK).**
* **You can control which IAM users or roles can:**
  + **Use the key**
  + **Encrypt/Decrypt data**
  + **Rotate or disable keys**

**2. Client-Side Encryption (CSE)**

**The application encrypts data before uploading to S3 and decrypts it after downloading.  
Used when:**

* **You don’t trust AWS to handle encryption keys.**
* **You need local key control.**

**Example: Encrypt with AWS Encryption SDK or client-side KMS integration.**

**3. Encryption in Transit (HTTPS)**

**Data is encrypted in transit using SSL/TLS (HTTPS) when users or apps interact with S3.**

**4. Example Team Setup (SSE-KMS)**

**Scenario:  
Team members use S3 for storing confidential project data.**

**Configuration:**

1. **Enable Bucket Default Encryption → SSE-KMS.**
2. **Create a Customer Managed Key (CMK) in AWS KMS.**
3. **Grant KMS key access only to specific IAM users/roles.**
4. **Define IAM + KMS Key Policies to:**
   * **Allow only authorized users to PutObject / GetObject.**
   * **Restrict who can encrypt/decrypt data.**

**Result:  
Each upload/download is logged in CloudTrail, showing *which user used which KMS key* — providing both security and auditability.**

**5. Short Interview Answer**

**“In S3, data can be encrypted using Server-Side Encryption or Client-Side Encryption.  
For team use, we typically enable SSE-KMS, where AWS KMS manages encryption keys and IAM policies control who can use those keys.  
This ensures data is encrypted at rest, in transit (via HTTPS), and access is auditable per user.”**

**182\* Cost analysis for AWS services.**

**Cost analysis means tracking, understanding, and optimizing how much you spend on AWS resources such as EC2, S3, RDS, Lambda, etc.**

**You use AWS’s billing and cost management tools to:**

* **Monitor usage**
* **Identify high-cost services**
* **Forecast future spend**
* **Optimize cost (by right-sizing or using discounts)**

**🧰 2️⃣ Tools for AWS Cost Analysis**

| **Tool** | **Purpose / Description** |
| --- | --- |
| **AWS Cost Explorer** | **Visualize your AWS spending by service, linked account, region, or tag. Create charts and reports.** |
| **AWS Budgets** | **Set custom budgets and receive alerts when spending exceeds a limit.** |
| **AWS Cost & Usage Report (CUR)** | **The most detailed cost data (hourly/daily), exported to S3.** |
| **Billing Dashboard** | **Quick view of total monthly cost and usage.** |
| **AWS Pricing Calculator** | **Estimate cost before deployment (what-if analysis).** |
| **Trusted Advisor (Cost Optimization)** | **Gives recommendations for unused resources or cheaper alternatives.** |
| **Compute Optimizer** | **Suggests right-sizing for EC2, Lambda, EBS based on usage.** |
| **Cost Anomaly Detection** | **Detects unusual spending automatically using ML.** |

**📊 3️⃣ Key Cost Components per Service**

| **Service** | **Cost Factors** |
| --- | --- |
| **EC2** | **Instance type, hours used, region, EBS volume size, data transfer** |
| **S3** | **Storage class, data stored (GB), requests, data transfer out** |
| **RDS** | **DB instance type, storage, backups, I/O requests** |
| **Lambda** | **Number of requests, execution time, memory allocated** |
| **EBS** | **Volume size, type (gp3, io2), IOPS** |
| **CloudFront** | **Data transfer out + requests** |
| **DynamoDB** | **Read/write capacity units, data storage, backup** |
| **VPC** | **NAT gateway hours, data processed, VPN connections** |
| **ECS / EKS (Fargate)** | **vCPU and memory usage per task or pod** |

**📈 4️⃣ Cost Optimization Strategies**

| **Area** | **Best Practice** |
| --- | --- |
| **EC2** | **Use Auto Scaling, Spot Instances, Reserved Instances, or Savings Plans** |
| **S3** | **Use Lifecycle policies to move data to cheaper storage (IA / Glacier)** |
| **RDS** | **Enable Stop when idle, use Reserved Instances, delete unused snapshots** |
| **Lambda** | **Optimize memory and execution time** |
| **EBS** | **Delete unattached volumes or snapshots** |
| **Monitoring** | **Enable CloudWatch alarms for unexpected usage** |
| **Tagging** | **Tag resources by project/team to identify who is spending what** |
| **Cross-Region Data Transfer** | **Minimize to reduce network costs** |

**📘 5️⃣ Example: Quick Cost Analysis**

**Scenario:**

* **EC2 (2 t3.medium, 24x7)**
* **S3 (500 GB Standard storage)**
* **RDS (db.t3.small, 100 GB)**

**Estimated monthly cost:**

| **Service** | **Approx. Cost (USD)** |
| --- | --- |
| **EC2** | **$60–$70** |
| **S3** | **$11–$12** |
| **RDS** | **$30–$35** |
| **Total** | **~$110/month** |

**You can calculate this using the AWS Pricing Calculator.**

**🧠 6️⃣ Cost Analysis Workflow**

1. **Collect usage data: Enable Cost Explorer & CUR.**
2. **Tag resources: By project, team, or environment.**
3. **Analyze patterns: Identify top services and trends.**
4. **Optimize: Apply right-sizing, lifecycle rules, or reserved pricing.**
5. **Monitor: Use Budgets and Anomaly Detection for continuous cost control.**

**✅ In Short**

| **Goal** | **Tool** | **Example** |
| --- | --- | --- |
| **View monthly spend** | **Billing Dashboard** | **See total $ for October** |
| **Analyze by service** | **Cost Explorer** | **Find that EC2 uses 60% of cost** |
| **Detect overspend** | **AWS Budgets** | **Alert when >$500** |
| **Forecast future** | **Cost Explorer forecast** | **Predict next month** |
| **Optimize** | **Trusted Advisor** | **Suggest removing idle EC2s** |

**184\* Waht are the limitaions of s services?**

**🔹 1️⃣ Definition**

**Every AWS service has default limits (quotas) set per account and region.  
These limits control how many resources you can create or how much capacity you can use.**

**💡 These limits exist to:**

* **Prevent accidental overuse or billing surprises.**
* **Protect AWS infrastructure from abuse.**
* **Manage resource allocation per customer.**

**🔹 2️⃣ Common Examples of Service Limits**

| **Service** | **Resource / Limit** | **Default Limit (approx.)** |
| --- | --- | --- |
| **EC2** | **Instances per region** | **20 (varies by instance type)** |
| **EBS** | **Volume size** | **16 TB per volume** |
| **Elastic IPs** | **Number per region** | **5 per account** |
| **S3** | **Buckets per account** | **100 buckets** |
| **RDS** | **DB instances per region** | **40 (varies by engine)** |
| **VPC** | **VPCs per region** | **5 (can increase)** |
| **Subnets per VPC** | **200** |  |
| **Security Groups** | **250 per VPC** |  |
| **Lambda** | **Memory per function** | **10 GB** |
| **CloudFormation** | **Stacks per region** | **200** |
| **IAM Roles** | **1000 per account** |  |
| **SNS Topics** | **10 million subscriptions** |  |

***(These are approximate and vary by AWS account type and region.)***

**🔹 3️⃣ How to Check Limits**

**You can view your current service quotas in:**

* **AWS Console → Service Quotas  
  or**
* **AWS CLI:**
* **aws service-quotas list-service-quotas --service-code ec2**

**🔹 4️⃣ How to Increase a Limit**

**If you hit a limit, you can request an increase:**

**In AWS Console:**

* **Go to Service Quotas**
* **Select the service (e.g., EC2)**
* **Click Request quota increase**

**AWS will review and approve (usually within minutes or hours).**

**🔹 5️⃣ Example in Real-Life**

**If you try to launch 30 EC2 instances but the default limit is 20 →  
you’ll get an error:**

**“You have exceeded your On-Demand Instance limit.”**

**You must then:**

* **Open Service Quotas,**
* **Request a higher limit for EC2.**

**✅ Summary Table**

| **Aspect** | **Details** |
| --- | --- |
| **What are limits?** | **Max number of resources per service/account** |
| **Why exist?** | **Prevent overuse, control capacity** |
| **Where to check?** | **AWS Console → Service Quotas** |
| **How to increase?** | **Request limit increase** |
| **Example** | **20 EC2s per region, 5 Elastic IPs** |

**185\* How you monitor unhealthy instance using cloud watch?**

**You can monitor unhealthy EC2 instances using Amazon CloudWatch together with ELB (Load Balancer), Auto Scaling Groups (ASG), or directly on the EC2 instance itself.**

**Let’s break it down clearly 👇**

**🔸 1️⃣ Monitoring EC2 Health using CloudWatch Metrics**

**Every EC2 instance automatically sends basic metrics to CloudWatch (at 5-minute or 1-minute intervals if detailed monitoring is enabled).**

**🔍 Key Metrics to Watch:**

| **Metric** | **Description** | **Threshold Example** |
| --- | --- | --- |
| **StatusCheckFailed** | **Fails if the instance is not reachable (either system or instance-level)** | **> 0** |
| **StatusCheckFailed\_Instance** | **OS-level failure (e.g., SSH/RDP not responding)** | **> 0** |
| **StatusCheckFailed\_System** | **Hardware or network issue on AWS host** | **> 0** |
| **CPUUtilization** | **High CPU can cause unresponsiveness** | **> 90%** |
| **DiskReadOps / DiskWriteOps** | **Helps detect storage-related issues** | **—** |

**⚙️ Steps to Monitor Unhealthy Instance:**

1. **Go to CloudWatch → Metrics → EC2 → Per-Instance Metrics**
2. **Check StatusCheckFailed metric**
   * **0 → Healthy**
   * **1 → Unhealthy**
3. **Create an Alarm on this metric.**

**🔸 2️⃣ Create a CloudWatch Alarm for Unhealthy Instance**

**Example:**

**Trigger an alarm when an EC2 instance fails a health check.**

**Steps:**

1. **Open CloudWatch → Alarms → Create alarm**
2. **Choose metric → EC2 → Per-Instance Metrics → StatusCheckFailed**
3. **Select your instance**
4. **Define threshold:**
   * **Threshold: > 0**
   * **Evaluation period: 1 of 1 datapoints**
5. **Add notification:**
   * **Send alert via Amazon SNS topic**
   * **E.g., notify-admins**

**✅ When the instance fails a status check → CloudWatch alarm state = ALARM → SNS sends notification (email/SMS/Lambda trigger).**

**🔸 3️⃣ Monitoring via Auto Scaling Group (ASG) + CloudWatch**

**If your EC2 is part of an Auto Scaling Group:**

* **ASG uses EC2 health checks and/or ELB health checks.**
* **When an instance becomes unhealthy:**
  + **ASG terminates the bad instance.**
  + **Launches a new one automatically.**
* **You can monitor these events in CloudWatch or CloudTrail logs.**

**Useful Metric:**

* **GroupInServiceInstances**
* **GroupTerminatingInstances**

**🔸 4️⃣ Monitoring via Load Balancer (ALB/ELB/NLB)**

**If instances are behind a Load Balancer, CloudWatch provides:**

| **Metric** | **Description** |
| --- | --- |
| **UnHealthyHostCount** | **Number of targets failing health checks** |
| **HealthyHostCount** | **Number of healthy targets** |

**You can create an alarm:**

* **If UnHealthyHostCount > 0 for a period  
  → Send SNS notification  
  → Or trigger Lambda to replace instance.**

**🔸 5️⃣ Optional: Automate Recovery**

**You can automatically recover unhealthy instances using:**

* **CloudWatch Alarm + EC2 Recovery Action**
* **Metric: StatusCheckFailed\_System > 0**
* **Action: Recover the instance**

**This will restart the instance on new healthy hardware automatically.**

**Or**

* **CloudWatch Alarm → Lambda function → Replace or reboot instance**

**✅ Summary Table**

| **Use Case** | **Metric** | **Action** |
| --- | --- | --- |
| **EC2 instance health** | **StatusCheckFailed** | **Alarm → SNS → Admin alert** |
| **Auto Scaling Group** | **GroupInServiceInstances** | **Replace unhealthy instance automatically** |
| **Load Balancer targets** | **UnHealthyHostCount** | **Alarm → Notify / Trigger Lambda** |
| **Auto recovery** | **StatusCheckFailed\_System** | **Auto recover instance** |

**186\* NAT gateway?**

**NAT (Network Address Translation) Gateway allows instances in a private subnet to connect to the internet (for updates, downloads, etc.) without exposing them to incoming internet traffic.**

**In short:**

**🟢 Private EC2 → Internet (Outbound only)  
🔴 Internet → Private EC2 (Blocked)**

**🔸 Why We Need NAT Gateway**

**If you have:**

* **A public subnet (with Internet Gateway), and**
* **A private subnet (without Internet access),**

**Then the private subnet cannot reach the internet directly.  
To allow private instances to:**

* **Download software updates**
* **Access external APIs securely  
  👉 You use a NAT Gateway.**

**🔹 How NAT Gateway Works (Architecture)**

**Example Setup:**

| **Subnet Type** | **Contains** | **Internet Access** |
| --- | --- | --- |
| **Public Subnet** | **NAT Gateway, ALB** | **✅ Outbound + Inbound (via IGW)** |
| **Private Subnet** | **EC2, DB, App servers** | **✅ Outbound only (via NAT)** |

**Flow:**

1. **Private instance sends request → to Internet (say yum update).**
2. **Request goes to Route Table → points to NAT Gateway.**
3. **NAT Gateway sends request via Internet Gateway (IGW).**
4. **Response comes back through NAT → to private instance.**

**✅ Outbound allowed  
❌ Inbound blocked (no direct access from Internet)**

**🔸 Steps to Create NAT Gateway**

1. **Go to VPC Console → NAT Gateways → Create NAT Gateway**
2. **Choose:**
   * **Subnet: Must be a Public Subnet**
   * **Elastic IP: Assign or create one**
3. **Create NAT Gateway**
4. **Update Private Route Table:**
   * **Destination: 0.0.0.0/0**
   * **Target: NAT Gateway**

**Now, all instances in the private subnet can access the internet securely.**

**🔸 NAT Gateway vs NAT Instance**

| **Feature** | **NAT Gateway** | **NAT Instance** |
| --- | --- | --- |
| **Managed by AWS** | **✅ Yes** | **❌ No (You manage EC2)** |
| **Scalability** | **Auto scales** | **Manual** |
| **Availability** | **Highly available in AZ** | **You must manage** |
| **Bandwidth** | **Up to 45 Gbps** | **Limited by instance type** |
| **Maintenance** | **None** | **You patch/secure it** |
| **Cost** | **Slightly higher** | **Cheaper (but manual work)** |

**✅ Best Practice: Use NAT Gateway — it’s managed, scalable, and reliable.  
Use NAT Instance only in very custom setups.**

**🔹 NAT Gateway Types**

| **Type** | **Description** |
| --- | --- |
| **Public NAT Gateway** | **Allows private instances to access the internet (via IGW).** |
| **Private NAT Gateway** | **Used for VPC-to-VPC communication (no IGW).** |

**🔹 High Availability Setup**

* **NAT Gateway is AZ-specific.  
  ➡️ Create one NAT Gateway per Availability Zone for fault tolerance.**
* **Update route tables in each private subnet to use the local NAT Gateway.**

**🔹 Pricing (Approx)**

| **Component** | **Description** | **Example** |
| --- | --- | --- |
| **Per Hour** | **~$0.045/hour** | **~$32/month per NAT GW** |
| **Data Processing** | **~$0.045 per GB** | **Based on traffic** |

**💡 To save cost — create NAT Gateways only in active AZs or schedule them off during non-usage hours.**

**✅ In Short**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | **Enables private subnet instances to access internet securely** |
| **Deployed In** | **Public subnet** |
| **Needs** | **Elastic IP + Route to IGW** |
| **Traffic Type** | **Outbound only** |
| **Best Practice** | **One NAT per AZ for redundancy** |

**Example Routing:**

**🔸 Private Subnet Route Table**

**Destination: 0.0.0.0/0 → Target: nat-xxxxxxxx**

**🔸 Public Subnet Route Table**

**Destination: 0.0.0.0/0 → Target: igw-xxxxxxxx**

**187\* Router?**

**A Router is a networking device that forwards data packets between different networks — typically between your local network (LAN) and the internet (WAN).**

**🧠 Simple Example:**

**When you connect your laptop to Wi-Fi at home:**

* **Your router connects all your devices (laptop, phone, TV) within the local network (LAN).**
* **It then routes traffic from your LAN to the internet (WAN) through your ISP (Internet Service Provider).**

**⚙️ Functions of a Router:**

| **Function** | **Description** |
| --- | --- |
| **Routing** | **Determines the best path for data packets to reach their destination.** |
| **Packet Forwarding** | **Sends packets from one network to another based on routing tables.** |
| **Network Segmentation** | **Connects multiple networks — e.g., separates LANs for security or efficiency.** |
| **NAT (Network Address Translation)** | **Converts private IPs into a public IP so multiple devices can share one internet connection.** |
| **Firewall Filtering** | **Some routers block or allow traffic based on configured rules.** |

**🏗️ Types of Routers:**

| **Type** | **Description** |
| --- | --- |
| **Core Router** | **Operates within the backbone of a large network.** |
| **Edge Router** | **Connects internal networks to external ones (like the internet).** |
| **Wireless Router** | **Provides Wi-Fi connectivity.** |
| **Virtual Router** | **Software-based router (used in cloud environments like AWS).** |

**☁️ In AWS Context:**

* **The VPC Route Table acts like a virtual router.**
* **It decides how traffic moves between subnets, internet gateway, and NAT gateway.**

**Example:  
If you have a route**

**Destination: 0.0.0.0/0 → Target: igw-123456**

**it means internet traffic is routed through the Internet Gateway — like a router sending data out to the internet.**

**🗣️ Short Interview Answer:**

**A router is a device that connects multiple networks and directs data packets between them.  
It decides the best path for network traffic using routing tables.  
In cloud environments like AWS, the VPC Route Table performs the role of a virtual router.**

**188\* subnetmask? which one has more ip values 24 or 25?**

**A subnet mask defines how many bits of an IP address represent the network portion and how many represent the host portion.**

**It helps divide a large network (like 192.168.0.0) into smaller subnets.**

**🔹 CIDR Notation**

**Instead of writing the subnet mask as 255.255.255.0, we often use CIDR notation, like /24.**

| **CIDR** | **Binary Meaning** | **Host Bits** | **Usable IPs** |
| --- | --- | --- | --- |
| **/24** | **255.255.255.0** | **8 bits** | **256 total → 254 usable** |
| **/25** | **255.255.255.128** | **7 bits** | **128 total → 126 usable** |

**🔹 Comparison: /24 vs /25**

| **Property** | **/24** | **/25** |
| --- | --- | --- |
| **Total IPs** | **256** | **128** |
| **Usable IPs** | **254** | **126** |
| **Network bits** | **24** | **25** |
| **Host bits** | **8** | **7** |
| **Range example** | **192.168.1.0 – 192.168.1.255** | **192.168.1.0 – 192.168.1.127 (1st subnet) / 192.168.1.128 – 192.168.1.255 (2nd subnet)** |

**✅ So:  
/24 subnet has more IP addresses than /25.**

**🔹 Quick Formula to Calculate IPs**

**Number of hosts = 2^(32 - subnet mask bits)**

**So:**

* **/24 → 2^(32-24) = 256**
* **/25 → 2^(32-25) = 128**

**(Then subtract 2 for network & broadcast addresses → usable IPs)**

**✅ Final Answer**

**“Subnet mask defines how many IPs are in a network.  
A /24 network (255.255.255.0) provides 256 IPs (254 usable), while /25 provides 128 IPs (126 usable).**

**Therefore, /24 has more IP addresses than /25.”**

**189\* How to configure autoscaling not to happen too rapidly?**

**To prevent rapid or frequent scaling (flapping), you use several stabilization mechanisms in AWS Auto Scaling:**

**1️⃣ Cooldown Period**

* **Definition: Time Auto Scaling waits after one scaling activity before starting another.**
* **Default: 300 seconds (5 minutes).**
* **Purpose: Gives new instances time to launch, warm up, and stabilize metrics.**

**📘 Example:**

**Cooldown period = 300 seconds**

**If a scale-out event adds 2 EC2 instances, Auto Scaling waits 5 minutes before triggering another event — even if CPU is still high.**

**2️⃣ Scaling Policies with Target Tracking**

* **Use Target Tracking Scaling Policies instead of simple step scaling.**
* **AWS automatically adds/removes capacity gradually to reach the target metric (e.g., 50% CPU).**
* **It avoids over-scaling by monitoring the metric’s trend.**

**📘 Example:**

**Target tracking metric = Average CPU utilization = 50%**

**Auto Scaling adjusts smoothly to maintain that level, not aggressively.**

**3️⃣ Instance Warm-up Time**

* **Set the warm-up period to tell Auto Scaling how long a new instance takes to become operational.**
* **During this warm-up, the instance’s metrics are ignored to avoid triggering more scaling too early.**

**📘 Example:**

**Instance warm-up = 300 seconds**

**4️⃣ Scaling Cooldowns per Policy**

* **Each scaling policy can have its own cooldown.**
* **Example: Scale-out cooldown = 180s; Scale-in cooldown = 600s (to avoid removing instances too soon).**

**5️⃣ CloudWatch Alarm Tuning**

* **Increase alarm evaluation periods to make scaling decisions more stable.  
  📘 Example:**
  + **Instead of 1 datapoint > 70%, require 3 consecutive datapoints above 70% before scaling out.**

**✅ Summary Table**

| **Technique** | **Purpose** |
| --- | --- |
| **Cooldown Period** | **Wait time after scaling event** |
| **Instance Warm-up** | **Ignore new instance metrics during startup** |
| **Target Tracking** | **Automatic smooth scaling** |
| **Policy-specific Cooldowns** | **Control timing for each scaling direction** |
| **CloudWatch Evaluation Periods** | **Prevent false alarms** |

**💬 Interview-Style Short Answer**

**“To prevent Auto Scaling from happening too rapidly, I configure a proper cooldown period and instance warm-up time, tune CloudWatch alarms to use multiple evaluation periods, and use target tracking policies for gradual scaling.  
This ensures the group scales smoothly without frequent scale-in/scale-out events.”**

**190\* AWS- vpc , ebs and s3 diff**

**🔹 Definition:**

**A Virtual Private Cloud (VPC) is your own private network inside AWS.  
It lets you control networking — IP addresses, subnets, routing, and security.**

**🔹 Purpose:**

**To securely host AWS resources (like EC2, RDS, etc.) inside an isolated virtual network.**

**🔹 Key Components:**

* **Subnets – Divide network into public/private sections.**
* **Route Tables – Define how traffic moves.**
* **Internet Gateway (IGW) – Allows public internet access.**
* **NAT Gateway – Lets private instances access the internet securely.**
* **Security Groups / NACLs – Control inbound/outbound traffic.**

**🔹 Example:**

**When you launch an EC2 instance, it runs inside a VPC subnet.**

**💽 2️⃣ AWS EBS (Elastic Block Store)**

**🔹 Definition:**

**EBS provides block-level storage volumes for EC2 instances — like a hard drive attached to a virtual machine.**

**🔹 Purpose:**

**To store operating system files, application data, and databases that need fast access and low latency.**

**🔹 Key Features:**

* **Acts as a virtual disk (mountable on EC2).**
* **Persists data even if instance stops.**
* **Supports snapshots (backups) to S3.**
* **Types: gp3 (general purpose SSD), io2 (high IOPS), st1/sc1 (HDD).**

**🔹 Example:**

**When you start an EC2 instance, the root volume is an EBS disk.**

**🪣 3️⃣ AWS S3 (Simple Storage Service)**

**🔹 Definition:**

**S3 is an object storage service used to store and retrieve any amount of data — images, videos, backups, logs, etc.**

**🔹 Purpose:**

**To store unstructured data and backup/archive content globally with high durability.**

**🔹 Key Features:**

* **Stores files as objects in buckets.**
* **99.999999999% (11 nines) durability.**
* **Integrates with CloudFront, Glacier, Athena, etc.**
* **Supports versioning, lifecycle rules, encryption, cross-region replication.**

**🔹 Example:**

**Hosting website images or storing backups.**

**⚖️ 4️⃣ Comparison Table**

| **Feature** | **VPC** | **EBS** | **S3** |
| --- | --- | --- | --- |
| **Type** | **Networking** | **Block Storage** | **Object Storage** |
| **Purpose** | **Network isolation for AWS resources** | **Disk for EC2 instances** | **Store files, backups, data** |
| **Access** | **Used by EC2, RDS, etc.** | **Attached to a single EC2 instance** | **Accessed via HTTP/HTTPS API** |
| **Persistence** | **N/A** | **Persistent** | **Persistent** |
| **Scalability** | **Manual (CIDR-based)** | **Limited to volume size** | **Virtually unlimited** |
| **Performance** | **Depends on EC2 & subnets** | **Low-latency block storage** | **High throughput, not low latency** |
| **Example Use Case** | **Create private network** | **Store database files** | **Store backups or media** |
| **Pricing Basis** | **No cost (only resources inside cost)** | **Per GB provisioned** | **Per GB stored + requests** |

**🧠 In Short**

| **Service** | **Think of it as** | **Used for** |
| --- | --- | --- |
| **VPC** | **Your AWS private network** | **Isolating and securing resources** |
| **EBS** | **Your hard disk drive (HDD/SSD)** | **Storing OS, app, DB data for EC2** |
| **S3** | **Your cloud file cabinet** | **Storing files, backups, media, logs** |

**191\* Lamda function how u used in project**

**🟢 1️⃣ Start with a Short Definition**

**“AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers. It automatically scales and you pay only for the execution time.”**

**🟢 2️⃣ Explain the Use Case in Your Project**

**Here are common, real-world examples — pick one (or combine a few) that match your type of project 👇**

**Example 1: S3 + Lambda (Image/File Processing)**

**In my project, we used AWS Lambda with Amazon S3.  
Whenever a new file was uploaded to an S3 bucket (like an image or CSV file), it triggered a Lambda function.  
The Lambda function processed the file — for example, resizing images, validating data, or moving the file to another bucket.**

**We configured the S3 bucket to trigger the Lambda automatically using an S3 Event Notification.**

**Flow:**

**S3 → Event Trigger → Lambda → Process File → Store in another S3 bucket / DynamoDB**

**Example 2: API Gateway + Lambda (Serverless API)**

**We created REST APIs using Amazon API Gateway and AWS Lambda.  
Each API endpoint (like /users, /orders) invoked a Lambda function that handled business logic — connecting to DynamoDB or RDS, processing data, and returning JSON responses.**

**This allowed us to build a completely serverless backend with no EC2 instances.**

**Flow:**

**Client → API Gateway → Lambda → DynamoDB / RDS → Response to Client**

**Example 3: Event-Driven Automation (CloudWatch / SNS / SQS)**

**We used Lambda for automation.  
For example, when CloudWatch detected high CPU usage on EC2, it triggered a Lambda function that automatically stopped, started, or scaled instances.**

**Similarly, when a message arrived in an SQS queue, the Lambda function consumed and processed it.**

**Flow:**

**CloudWatch / SNS / SQS → Lambda → Execute Action (e.g., EC2 scale-up)**

**Example 4: Data Pipeline (ETL Processing)**

**In our data pipeline, we used Lambda to transform data before storing it in S3 or Redshift.  
The pipeline was triggered whenever new data arrived in S3. Lambda parsed, cleaned, and reformatted the data.**

**🟢 3️⃣ Mention Language and Tools**

**We wrote Lambda functions in Python and Node.js.  
Deployed using AWS SAM (Serverless Application Model) or Terraform.  
Integrated with CloudWatch Logs for monitoring and troubleshooting.**

**🟢 4️⃣ Mention Advantages You Gained**

* **No servers to manage**
* **Auto scaling**
* **Low cost (pay only per execution)**
* **Easy to integrate with other AWS services**

**✅ Sample Complete Answer (Interview Style)**

**“In my project, I used AWS Lambda with S3 and API Gateway. Whenever a file was uploaded to an S3 bucket, it triggered a Lambda function written in Python. The Lambda validated and processed the file, then stored the results in DynamoDB.**

**I also used Lambda behind an API Gateway to create a few serverless REST endpoints. We deployed the Lambda using Terraform and monitored logs through CloudWatch. This setup helped us achieve a fully serverless, auto-scaling architecture without managing EC2 servers.”**

**192\* Ec2 trouble shooting if u r not able to login**

**🔸 1️⃣ Check Network Configuration**

**✅ a. Security Group Rules**

**Go to EC2 → Security Groups → Inbound rules**

* **For Linux (SSH) → Allow TCP port 22**
* **For Windows (RDP) → Allow TCP port 3389**
* **Source should be your IP address or trusted range**

**✅ b. Network ACLs**

* **Inbound & Outbound rules must allow your port (22 or 3389)**
* **Make sure not accidentally blocked by “Deny” rules**

**✅ c. Route Table**

* **Check your VPC Route Table**
  + **For public subnet:  
    0.0.0.0/0 → igw-xxxxxxxx (Internet Gateway)**
  + **For private subnet:  
    Route through NAT Gateway or Bastion Host**

**🔸 2️⃣ Verify Instance Accessibility**

**✅ a. Check Public IP**

* **Ensure the EC2 has a Public IPv4 address (or Elastic IP)**
* **If connecting via Bastion Host, ensure that host is reachable**

**✅ b. Check DNS / Elastic IP Mapping**

* **Sometimes the Elastic IP may have been detached or reassigned**

**🔸 3️⃣ Verify Key Pair & SSH Configuration**

**✅ a. Check SSH Key**

* **Use the exact .pem file used when creating the instance**
* **Set correct permissions:**
* **chmod 400 my-key.pem**
* **Connect:**
* **ssh -i my-key.pem ec2-user@<public-ip>**

**✅ b. Wrong User Name**

**Different AMIs use different default usernames:**

| **OS** | **Username** |
| --- | --- |
| **Amazon Linux** | **ec2-user** |
| **Ubuntu** | **ubuntu** |
| **CentOS** | **centos** |
| **RHEL** | **ec2-user** |
| **SUSE** | **ec2-user** |
| **Windows** | **Administrator** |

**🔸 4️⃣ Check Instance Status**

**Go to EC2 → Instances → Status Checks**

* **System Status Check: Hardware/network issue**
* **Instance Status Check: OS-level issue**

**If failed → investigate with EC2 serial console or recover instance.**

**🔸 5️⃣ OS-Level Issues**

**If instance boots but you still can’t connect:**

* **OS firewall (iptables / firewalld) may block SSH/RDP**
* **/etc/ssh/sshd\_config may have been misconfigured**
* **Wrong permissions on .ssh/authorized\_keys**
* **Disk full or corrupted filesystem may prevent login**

**🔸 6️⃣ Use EC2 Serial Console (for Linux)**

**If networking is broken:**

1. **Go to EC2 → Instances → Connect → EC2 Serial Console**
2. **You get a terminal directly to the OS (no SSH)**
3. **From there you can:**
   * **Check /etc/ssh/sshd\_config**
   * **Restart SSH service:**
   * **sudo systemctl restart sshd**
   * **Fix firewall or routing issues**

***(Enable “EC2 Serial Console Access” in IAM if not already)***

**🔸 7️⃣ Use AWS Systems Manager (SSM) Session Manager**

**If SSM Agent is installed and IAM role has AmazonSSMManagedInstanceCore:**

* **You can connect directly via Session Manager (no SSH needed)**
* **From there, check logs, networking, and restart services.**

**🔸 8️⃣ Use Root Volume Recovery Method (Last Resort)**

**If SSH/RDP is broken and SSM not working:**

1. **Stop the instance**
2. **Detach its root EBS volume**
3. **Attach that volume to another healthy EC2 instance**
4. **Mount it and:**
   * **Fix /etc/ssh/sshd\_config**
   * **Remove firewall rules**
   * **Check /home/ec2-user/.ssh/authorized\_keys**
5. **Detach and reattach the volume back to original instance**
6. **Start the instance and retry login**

**🔸 9️⃣ Check CloudWatch Logs**

**If CloudWatch Agent or SSM is configured, review logs for:**

* **SSH failures**
* **Disk/CPU spikes**
* **Network errors**

**✅ Summary Table**

| **Category** | **Check** | **Command / Fix** |
| --- | --- | --- |
| **Network** | **Security Group, NACL, Route** | **Verify ports & routing** |
| **Key Pair** | **Correct PEM/PPK** | **chmod 400 key.pem** |
| **User** | **Correct login user** | **ssh -i key.pem ec2-user@ip** |
| **Instance** | **Health checks** | **View in AWS Console** |
| **OS** | **Firewall / SSH config** | **sudo systemctl restart sshd** |
| **Recovery** | **Volume attach / SSM / Serial console** | **Fix from another instance** |

**193\* EKS**

**Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service by AWS that lets you run Kubernetes clusters easily without having to install, operate, or maintain your own Kubernetes control plane.**

**👉 In short:**

**EKS = AWS-managed Kubernetes**

**🔸 1️⃣ Why EKS? (Purpose)**

**Normally, when you set up Kubernetes yourself, you must manage:**

* **The control plane (API Server, etcd, Controller Manager, Scheduler)**
* **Security patches and upgrades**
* **High availability and scaling**

**EKS handles all that for you automatically.**

**✅ EKS manages:**

* **Control plane (master nodes)**
* **Automatic scaling**
* **High availability across AZs**
* **Integration with AWS IAM, VPC, CloudWatch, and more.**

**You manage only:**

* **Worker nodes (EC2 or Fargate)**
* **Applications and workloads**

**🔹 2️⃣ EKS Architecture**

**Main Components**

| **Component** | **Description** |
| --- | --- |
| **Control Plane (Master)** | **Managed by AWS — handles scheduling, API, cluster state.** |
| **Worker Nodes** | **EC2 instances or AWS Fargate pods running your containers.** |
| **EKS Cluster** | **Combination of control plane + worker nodes.** |
| **Node Groups** | **Set of worker nodes in an Auto Scaling group.** |
| **Pods** | **Smallest deployable unit in Kubernetes (containers).** |

**EKS Architecture Diagram (Conceptually)**

**+------------------------------------------------+**

**| AWS EKS Cluster |**

**|------------------------------------------------|**

**| Control Plane (Managed by AWS) |**

**| - API Server |**

**| - etcd (Cluster state) |**

**| - Scheduler & Controllers |**

**|------------------------------------------------|**

**| Worker Nodes (EC2 / Fargate) |**

**| - kubelet, kube-proxy |**

**| - Pods / Containers |**

**+------------------------------------------------+**

**🔸 3️⃣ How EKS Works**

1. **Create EKS Cluster**
   * **AWS provisions the Kubernetes control plane automatically.**
2. **Launch Worker Nodes**
   * **Add EC2 or Fargate nodes to your cluster.**
3. **Connect kubectl**
   * **Configure your local machine using:**
   * **aws eks update-kubeconfig --name <cluster-name>**
4. **Deploy Applications**
   * **Use Kubernetes manifests (.yaml files) to deploy pods, services, etc.**
   * **kubectl apply -f deployment.yaml**
5. **EKS manages scaling, updates, monitoring, IAM roles, and networking.**

**🔹 4️⃣ EKS Deployment Options**

| **Type** | **Description** |
| --- | --- |
| **EKS on EC2** | **Worker nodes run on EC2 instances (you manage scaling).** |
| **EKS on Fargate** | **Serverless — AWS runs containers without EC2 nodes.** |
| **EKS Anywhere** | **Run EKS on your on-prem servers (hybrid).** |

**🔹 5️⃣ Key AWS Integrations**

| **AWS Service** | **Integration** |
| --- | --- |
| **IAM** | **Manage access using IAM roles for service accounts (IRSA).** |
| **VPC** | **EKS clusters run inside your VPC.** |
| **CloudWatch** | **Logs and metrics from pods.** |
| **ALB/NLB** | **For exposing Kubernetes services externally.** |
| **ECR** | **Container image storage for your pods.** |

**🔹 6️⃣ EKS vs ECS**

| **Feature** | **EKS (Kubernetes)** | **ECS (AWS Native)** |
| --- | --- | --- |
| **Orchestration Engine** | **Kubernetes** | **AWS proprietary** |
| **Flexibility** | **Multi-cloud / Open-source** | **AWS-only** |
| **Control Plane** | **Managed by AWS** | **Managed by AWS** |
| **Learning Curve** | **Higher (Kubernetes concepts)** | **Easier** |
| **Use Case** | **Multi-cloud portability, complex workloads** | **Simple AWS-only container orchestration** |

**🔹 7️⃣ Networking in EKS**

* **Each pod gets an ENI (Elastic Network Interface) via the VPC CNI plugin.**
* **You can use ClusterIP, NodePort, or LoadBalancer type services.**
* **Ingress Controller (ALB Ingress) manages routing external traffic into the cluster.**

**🔹 8️⃣ Monitoring EKS**

**Use:**

* **CloudWatch Container Insights (metrics and logs)**
* **Prometheus + Grafana (custom dashboards)**
* **AWS X-Ray (tracing)**

**🔹 9️⃣ Pricing**

* **You pay $0.10 per hour per EKS cluster (control plane).**
* **Plus EC2 / Fargate / EBS costs for your workloads.**

**✅ Summary Table**

| **Feature** | **Description** |
| --- | --- |
| **Service Type** | **Managed Kubernetes** |
| **Control Plane** | **Managed by AWS** |
| **Worker Nodes** | **EC2 or Fargate** |
| **Scaling** | **Auto Scaling Groups / HPA** |
| **Security** | **IAM roles, KMS, Security Groups** |
| **Monitoring** | **CloudWatch, Prometheus, Grafana** |
| **Pricing** | **$0.10/hour per cluster** |

**✅ In Short**

**Amazon EKS = Kubernetes on AWS, managed by AWS.**

* **You get full Kubernetes power**
* **Without managing master nodes**
* **Integrated with AWS services (IAM, CloudWatch, ALB, ECR)**
* **Runs on EC2 or Fargate**

**194\* Which region ur applications are hosted - tell atleast two regions**

**✅ Sample Answer (Professional and Realistic):**

**Our applications are hosted primarily in the Asia Pacific (Mumbai) region — that’s ap-south-1, since most of our user base is in India.**

**For high availability and disaster recovery, we also have a secondary deployment in the US East (N. Virginia) region — us-east-1.**

**We use Route 53 for DNS-based failover between these regions, and S3 Cross-Region Replication to back up data from Mumbai to Virginia.**

**🌍 Other Common AWS Regions (for reference):**

| **Region Name** | **Code** |
| --- | --- |
| **Asia Pacific (Mumbai)** | **ap-south-1** |
| **Asia Pacific (Singapore)** | **ap-southeast-1** |
| **US East (N. Virginia)** | **us-east-1** |
| **US West (Oregon)** | **us-west-2** |
| **Europe (Frankfurt)** | **eu-central-1** |
| **Europe (London)** | **eu-west-2** |

**🗣️ Short Interview Answer:**

**Our applications are deployed in Mumbai (ap-south-1) and N. Virginia (us-east-1) regions for better availability and disaster recovery.**

**195\* Which region kubernetes cluster set up**

**“We set up our Kubernetes cluster in the ap-south-1 (Mumbai) region, since our majority of users and application backend are based in India.**

**Within that region, we deployed worker nodes across multiple Availability Zones (for example, ap-south-1a and ap-south-1b) to ensure high availability and fault tolerance.**

**We used Amazon EKS (Elastic Kubernetes Service) for the control plane and configured autoscaling groups for worker nodes in private subnets.**

**For networking, we used VPC CNI plugin and integrated CloudWatch + Prometheus for monitoring.”**

**🔹 Alternative Example (If You Used Another Region)**

**“Our Kubernetes cluster is deployed in us-east-1 (N. Virginia) region because it provides maximum AWS service availability and lowest latency for global users.**

**We also chose this region since most of our AWS resources (like S3, RDS, CloudFront origin) are in the same region to reduce inter-region data transfer cost.”**

**🔹 Why That Region**

**You should mention one or more of these reasons depending on your setup:**

* **🌍 User proximity → to reduce latency**
* **💰 Cost optimization → cheaper instance pricing**
* **⚙️ Service availability → some services (like EKS, FSx) may not be in all regions**
* **🔁 Redundancy → multi-AZ for fault tolerance**
* **📊 Compliance → data residency requirements**

**✅ Short, Interview-Ready Answer**

**“We deployed our Kubernetes (EKS) cluster in the ap-south-1 (Mumbai) region across two Availability Zones for high availability.  
The region was chosen because most of our users are in India, and we wanted to minimize latency while keeping data local.”**

**196\* CDN in AWS- cloud front**

**Amazon CloudFront is AWS’s Content Delivery Network (CDN) that delivers content (web pages, videos, APIs, images, etc.) to users with low latency and high transfer speed using a global network of edge locations.**

**🌍 How It Works**

1. **User requests content (e.g., image, video, or website).**
2. **The request goes to the nearest CloudFront edge location.**
3. **If the edge location has cached content, it serves it immediately.**
4. **If not, CloudFront fetches it from the origin (like S3 bucket, EC2, or ALB), caches it at the edge, and serves it to the user.**
5. **Next user nearby gets it directly from the cache.**

**🧩 Key Components**

| **Component** | **Description** |
| --- | --- |
| **Origin** | **The source of the content (e.g., S3 bucket, EC2 instance, ALB, or custom HTTP server).** |
| **Edge Location** | **Global data centers where CloudFront caches content closer to users.** |
| **Distribution** | **The configuration unit that tells CloudFront what to cache and from where (the origin).** |
| **Cache Behavior** | **Rules for caching, TTLs, and which methods (GET, POST) are allowed.** |
| **Invalidation** | **Used to remove outdated content from cache manually.** |

**⚡ Benefits**

| **Feature** | **Explanation** |
| --- | --- |
| **Low Latency** | **Delivers content from the nearest edge location.** |
| **Improved Performance** | **Caches frequently accessed data.** |
| **Scalability** | **Automatically scales to handle traffic spikes.** |
| **Security** | **Integrated with AWS Shield, WAF, and ACM for HTTPS.** |
| **Cost Optimization** | **Reduces data transfer load from your origin servers.** |

**🔐 Security Integrations**

* **AWS WAF → For web application firewall protection.**
* **AWS Shield → For DDoS protection.**
* **SSL/TLS Certificates (ACM) → For HTTPS encryption.**
* **Origin Access Control (OAC) → Restricts direct access to your S3 bucket.**

**💡 Common Use Cases**

* **Static websites hosted on S3 + CloudFront**
* **Media streaming (videos, images)**
* **API acceleration**
* **Software downloads**
* **Multi-region application delivery**

**🧠 Example Interview Answer**

**“CloudFront is AWS’s global CDN service. It caches content at edge locations worldwide, reducing latency and improving performance for end users.  
It integrates with services like S3, ALB, and API Gateway as origins, and supports security through HTTPS, WAF, and Shield.”**

**197\* In real time examples for cloud front**

**CloudFront caches and serves your content (images, videos, APIs, or entire websites) from edge locations close to users — improving:**

* **Speed (reduced latency)**
* **Security (with AWS Shield, WAF)**
* **Availability (content served even if origin is slow)**

**💡 Real-Time Use Cases of CloudFront**

**1️⃣ Website Content Delivery**

**Example:  
An e-commerce site (like Amazon, Flipkart, or Myntra) uses CloudFront to deliver:**

* **Product images**
* **CSS and JavaScript files**
* **Videos and static assets**

**📍 Benefit:  
Customers around the world load the site faster because content is cached at nearest edge locations.**

**2️⃣ Video Streaming (OTT Platforms)**

**Example:  
Platforms like Hotstar, Netflix, or YouTube (similar architecture) use CloudFront to:**

* **Stream on-demand videos**
* **Handle millions of viewers globally**
* **Cache media segments close to users**

**📍 Benefit:  
Reduces buffering, improves user experience.**

**3️⃣ API Acceleration**

**Example:  
A mobile app that fetches data from an AWS API Gateway or EC2 backend.**

**📍 How CloudFront helps:  
CloudFront can cache API responses or accelerate HTTPS requests to your APIs — reducing latency for users globally.**

**4️⃣ Secure File Downloads**

**Example:  
A software company distributing application updates or patches (like antivirus updates or drivers).**

**📍 CloudFront advantage:**

* **Secure download URLs**
* **Geolocation restriction**
* **High speed and reliability**

**5️⃣ Static Website Hosting**

**Example:  
You host a static website (HTML, CSS, JS) on S3, and use CloudFront as a CDN in front of it.**

**📍 Benefit:**

* **Faster delivery**
* **HTTPS support (SSL certs via ACM)**
* **Lower S3 costs (due to caching)**

**6️⃣ Protecting Web Applications**

**Example:  
A financial web portal uses CloudFront with AWS WAF and Shield to protect from:**

* **DDoS attacks**
* **SQL injection**
* **Malicious bots**

**📍 Benefit:  
Adds a security layer before requests hit your origin servers.**

**🧠 Common Architecture Example**

**User → CloudFront → S3 / ALB / EC2 / API Gateway → RDS**

**CloudFront acts as a global caching and security layer in front of your backend.**

**✅ In Summary**

| **Use Case** | **Origin** | **CloudFront Benefit** |
| --- | --- | --- |
| **Static website hosting** | **S3** | **Faster loading, global delivery** |
| **API acceleration** | **API Gateway / EC2** | **Low latency** |
| **Video streaming** | **S3 / Media Services** | **Smooth playback** |
| **File downloads** | **S3 / EC2** | **Secure and fast** |
| **Web app protection** | **Any origin** | **WAF + DDoS protection** |

**198\* Have you deployed any application to S3 bucket.**

**🟢 1️⃣ Start with a Simple Yes and Context**

**“Yes, I have deployed a static web application to an Amazon S3 bucket.”**

**🟢 2️⃣ Explain What Kind of Application**

**You can say something like:**

**“It was a front-end web application built using HTML, CSS, and JavaScript (React or Angular).  
We used S3 to host it as a static website.”**

**🟢 3️⃣ Step-by-Step — How You Did It**

| **Step** | **Description** |
| --- | --- |
| **1. Created S3 Bucket** | **From the AWS Console → Gave a unique name (e.g., myapp-frontend-bucket)** |
| **2. Enabled Static Website Hosting** | **Under Properties → Static website hosting, selected “Host a static website”** |
| **3. Uploaded Website Files** | **Uploaded index.html, error.html, CSS, JS files** |
| **4. Set Permissions (Make Public)** | **Updated Bucket Policy to allow public read access or used CloudFront for secure distribution** |
| **5. Got Website Endpoint** | **Example: http://myapp-frontend-bucket.s3-website-us-east-1.amazonaws.com** |
| **6. (Optional) Used CloudFront** | **Created a CloudFront distribution to improve performance, security, and use HTTPS with a custom domain** |
| **7. (Optional) Used Route 53** | **Mapped custom domain (e.g., www.myapp.com) to CloudFront** |

**🧠 Bucket Policy Example (if public hosting):**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Sid": "PublicReadGetObject",**

**"Effect": "Allow",**

**"Principal": "\*",**

**"Action": ["s3:GetObject"],**

**"Resource": ["arn:aws:s3:::myapp-frontend-bucket/\*"]**

**}**

**]**

**}**

**🟢 4️⃣ Add Deployment Automation (if you used CI/CD)**

**“We automated the deployment using AWS CodePipeline and CodeBuild.  
Whenever code was pushed to the GitHub repository, CodePipeline automatically built and deployed the static files to the S3 bucket.”**

**🟢 5️⃣ End with Benefits**

**“Hosting on S3 was very cost-effective, scalable, and required no servers.  
Combined with CloudFront, it gave us a secure and globally distributed website.”**

**✅ Sample Interview Answer (Complete)**

**“Yes, I deployed a static website to an S3 bucket.  
It was a React front-end for our web application. I created an S3 bucket, enabled static website hosting, and uploaded the build files (index.html, JS, CSS).**

**I then configured the bucket policy for public access and used CloudFront for HTTPS and caching.  
We automated the deployment using AWS CodePipeline so every Git push automatically updated the S3 site.**

**This approach gave us a simple, highly available, and low-cost static hosting solution.”**

**199\* What kind of data stored in s3 bucket- dynamic or static**

**Amazon S3 is primarily used to store static data, not dynamic data.**

**🔹 1️⃣ Static Data (Stored in S3)**

**Static content means files or objects that do not change frequently and can be served directly to users without server-side processing.**

**🧩 Examples:**

* **HTML, CSS, JavaScript files**
* **Images, videos, PDFs, documents**
* **Static website assets**
* **Backups, logs, or exported data**
* **Data lake files (CSV, JSON, Parquet, etc.)**

**📦 These files are stored as objects in S3 buckets, and S3 is excellent for highly available, durable, and scalable static storage.**

**🔹 2️⃣ Dynamic Data**

**Dynamic data changes frequently or is generated in real time by applications (for example, content that depends on user input or database queries).**

**Examples:**

* **User login responses**
* **Personalized dashboards**
* **Live feeds / real-time API responses**

**These are not stored directly in S3 — they are usually handled by:**

* **Databases (e.g., RDS, DynamoDB)**
* **Application servers or Lambda functions**

**However, the *output* or *snapshot* of dynamic data (like exported reports, generated images, or logs) can be saved to S3.**

**🔹 3️⃣ Typical Architecture Example**

| **Component** | **Function** | **Storage** |
| --- | --- | --- |
| **S3** | **Stores static website content** | **Static files** |
| **EC2 / Lambda / API Gateway** | **Handles dynamic requests** | **Processes & saves outputs** |
| **RDS / DynamoDB** | **Stores dynamic application data** | **Dynamic database** |

**✅ In short:**

| **Type** | **Stored in S3?** | **Examples** |
| --- | --- | --- |
| **Static Data** | **✅ Yes** | **HTML, JS, Images, PDFs** |
| **Dynamic Data** | **❌ No (but results can be stored)** | **Real-time user data, DB records** |

**👉 Summary:  
S3 is designed to store static data — files that can be directly served or downloaded.  
Dynamic content is typically generated and served via compute services, not stored in S3 directly.**

**200\* Types of servers used : app and web server Configuration details for jboss server**

**🧩 1️⃣ Types of Servers Used**

**In a typical three-tier architecture, we have:**

| **Layer** | **Type of Server** | **Purpose** | **Examples** |
| --- | --- | --- | --- |
| **Presentation Layer** | **Web Server** | **Serves static content (HTML, CSS, JS) and routes requests to app layer** | **Apache HTTPD, Nginx, IIS** |
| **Application Layer** | **Application Server** | **Executes business logic and dynamic requests** | **JBoss (WildFly), Tomcat, WebLogic, WebSphere** |
| **Database Layer** | **Database Server** | **Stores and manages data** | **MySQL, PostgreSQL, Oracle, RDS** |

**💡 Example: AWS Deployment**

| **Layer** | **AWS Component** |
| --- | --- |
| **Web Server** | **EC2 (Apache/Nginx) behind ALB** |
| **App Server** | **EC2 with JBoss/WildFly** |
| **DB Server** | **Amazon RDS (MySQL/PostgreSQL)** |

**Traffic flow:  
User → Route53 → ALB → Web Server → App Server (JBoss) → RDS**

**⚙️ 2️⃣ Web Server Configuration (Apache/Nginx Example)**

**Purpose:**

* **Handles static files (CSS/HTML)**
* **Reverse proxies dynamic requests to JBoss**
* **Improves performance with caching & load balancing**

**Example Apache Virtual Host Config:**

**<VirtualHost \*:80>**

**ServerName app.example.com**

**ProxyPreserveHost On**

**ProxyPass / http://localhost:8080/**

**ProxyPassReverse / http://localhost:8080/**

**ErrorLog /var/log/httpd/app\_error.log**

**CustomLog /var/log/httpd/app\_access.log combined**

**</VirtualHost>**

**Here, Apache listens on port 80 and forwards requests to JBoss running on port 8080.**

**🔧 3️⃣ Application Server Configuration — JBoss (WildFly)**

**JBoss/WildFly is a Java EE Application Server used to deploy .war or .ear applications.**

**🔹 Key Configuration Files**

| **File** | **Purpose** |
| --- | --- |
| **standalone.xml** | **Core server configuration (ports, datasources, logging, security, etc.)** |
| **standalone.conf** | **JVM settings (heap, GC, etc.)** |
| **application-users.properties** | **Defines users for web console and management** |
| **logging.properties** | **Configures log levels and log file locations** |

**🔹 Common Configuration Steps**

**1. Directory Structure**

**/opt/jboss/**

**├── bin/**

**├── standalone/**

**│ ├── configuration/**

**│ ├── deployments/**

**│ └── log/**

**└── modules/**

**2. Set JVM Memory (standalone.conf)**

**JAVA\_OPTS="-Xms1024m -Xmx2048m -XX:MaxPermSize=512m"**

**3. Define Ports (standalone.xml)**

**Default JBoss ports:**

| **Service** | **Port** |
| --- | --- |
| **Web (HTTP)** | **8080** |
| **HTTPS** | **8443** |
| **Management** | **9990** |
| **AJP** | **8009** |

**Example snippet:**

**<socket-binding-group name="standard-sockets" default-interface="public">**

**<socket-binding name="http" port="8080"/>**

**<socket-binding name="https" port="8443"/>**

**</socket-binding-group>**

**4. Configure Database Connection (Datasource)**

**In standalone.xml:**

**<datasource jndi-name="java:/MyDS" pool-name="MyDS" enabled="true">**

**<connection-url>jdbc:mysql://db.example.com:3306/mydb</connection-url>**

**<driver>mysql</driver>**

**<security>**

**<user-name>admin</user-name>**

**<password>password</password>**

**</security>**

**</datasource>**

**5. Deploy Application**

**Copy the .war file to:**

**/opt/jboss/standalone/deployments/**

**JBoss automatically deploys it.**

**🔹 6. Start & Stop JBoss**

**# Start in standalone mode**

**./standalone.sh &**

**# Stop JBoss**

**pkill -f jboss**

**You can also run it as a systemd service:**

**systemctl start jboss**

**systemctl enable jboss**

**🔹 7. Security & Logging**

* **Secure console: https://<server-ip>:9990**
* **Log file: /opt/jboss/standalone/log/server.log**
* **Use CloudWatch Agent to stream logs to AWS CloudWatch.**

**🌐 4️⃣ Example Architecture Summary**

**User**

**↓**

**Route53**

**↓**

**Application Load Balancer**

**↓**

**[Web Server - Apache/Nginx on EC2]**

**↓**

**[App Server - JBoss/WildFly on EC2]**

**↓**

**[Database - RDS MySQL/PostgreSQL]**

* **Auto Scaling on web and app layers**
* **ALB health checks**
* **IAM roles for EC2 → RDS access**
* **CloudWatch alarms for monitoring**

**🗣️ 5️⃣ Short Interview Answer (30 Seconds)**

**“We use a 3-tier setup — web servers with Apache/Nginx to handle static content and proxy requests, JBoss (WildFly) as the application server to deploy Java-based applications, and RDS for database storage.  
JBoss is configured through standalone.xml for ports and datasources, and standalone.conf for JVM tuning.  
The application is deployed as a WAR under standalone/deployments, with auto-scaling and monitoring integrated via CloudWatch.”**

**202\* Target groups and autoscaling groups and difference betn them**

**✅ Short Summary Answer**

| **Feature** | **Target Group** | **Auto Scaling Group (ASG)** |
| --- | --- | --- |
| **Purpose** | **Routes traffic to registered targets (like EC2, IPs, Lambda)** | **Automatically manages the number of EC2 instances** |
| **Main Function** | **Load balancing and health check routing** | **Scaling (increase/decrease) instances based on demand** |
| **Who Uses It** | **Used by Load Balancers (ALB/NLB)** | **Works with Load Balancers or standalone** |
| **Manages** | **Targets (EC2s, IPs, Lambda)** | **Instance lifecycle (launch/terminate)** |
| **Health Check** | **Performed by Load Balancer** | **Can use ELB health check or EC2 status check** |
| **Example** | **ALB routes traffic to target group instances** | **ASG scales instances in/out and registers them to target group** |

**🔹 1. What is a Target Group?**

**A Target Group is a logical grouping of targets (like EC2 instances, IPs, or Lambda functions) that a Load Balancer sends traffic to.**

**✳️ Key Points**

* **Used by Application Load Balancer (ALB), Network Load Balancer (NLB), or Gateway Load Balancer.**
* **You define a health check path (e.g., /health) to monitor targets.**
* **Targets can be:**
  + **EC2 instances**
  + **IP addresses**
  + **Lambda functions**
  + **Containers (ECS tasks)**

**Example:  
If an ALB is fronting your web app:**

* **Load balancer → Target Group → EC2 instances**

**CLI Example:**

**aws elbv2 create-target-group --name my-tg --protocol HTTP --port 80 --vpc-id vpc-xxxx**

**🔹 2. What is an Auto Scaling Group (ASG)?**

**An Auto Scaling Group automatically launches or terminates EC2 instances based on:**

* **Load (CPU, memory, request count)**
* **CloudWatch alarms**
* **Schedules (like peak hours)**

**✳️ Key Points**

* **You define:**
  + **Launch Template (AMI, instance type, security group, etc.)**
  + **Desired Capacity, Min, and Max instance counts**
* **It ensures the right number of healthy instances are running.**

**Example:  
If CPU > 70%, ASG adds 2 more EC2 instances.  
If CPU < 30%, it removes 1 instance.**

**🔹 3. How They Work Together**

**They often work in combination:**

**Auto Scaling Group creates and manages EC2 instances,  
and Target Group (used by Load Balancer) routes traffic to those instances.**

**So:**

**Users → Load Balancer → Target Group → EC2 (managed by Auto Scaling Group)**

* **ASG automatically registers new instances into the Target Group.**
* **When an instance is unhealthy or terminated, ASG replaces it automatically.**

**🔹 4. Difference Summary**

| **Feature** | **Target Group** | **Auto Scaling Group** |
| --- | --- | --- |
| **Purpose** | **Traffic routing** | **Instance scaling** |
| **Component of** | **Load Balancer (ALB/NLB)** | **EC2 Auto Scaling service** |
| **Controls** | **Traffic flow** | **Number of EC2s** |
| **Health Check Done By** | **Load Balancer** | **ELB or EC2 health check** |
| **Automatic Replacement** | **No** | **Yes** |
| **Example** | **ALB routes requests to healthy targets** | **ASG adds/removes instances based on CPU usage** |

**💬 Sample Interview Answer**

**“A Target Group is used by a Load Balancer to route requests to one or more registered targets like EC2 instances.  
An Auto Scaling Group, on the other hand, automatically adjusts the number of EC2 instances based on demand.**

**They often work together — the ASG adds or removes instances, and the Target Group ensures traffic only goes to healthy ones.  
For example, in my project, our ALB forwards traffic to a target group linked with an ASG that scales EC2s automatically during high load.”**

**203\* Diff betn subnet and security group**

**🧩 1️⃣ Subnet – Network-Level Division**

**🔹 Definition:**

**A Subnet is a logical subdivision of a VPC (Virtual Private Cloud) network.  
It defines where your resources live and which part of the network (public or private) they belong to.**

**🔹 Purpose:**

* **To divide your VPC’s IP range into smaller ranges.**
* **To organize and isolate resources (like EC2 instances).**
* **To control routing — whether the subnet has internet access or not.**

**🔹 Example:**

* **VPC CIDR → 10.0.0.0/16**
* **Public Subnet → 10.0.1.0/24**
* **Private Subnet → 10.0.2.0/24**

**Instances in:**

* **Public subnet → have Internet Gateway route (public access).**
* **Private subnet → use NAT Gateway or no Internet route (internal only).**

**🧠 Think of Subnet as:**

**“Where your instance lives inside the network.”**

**🧩 2️⃣ Security Group – Instance-Level Firewall**

**🔹 Definition:**

**A Security Group (SG) is a virtual firewall that controls inbound and outbound traffic for EC2 instances, RDS, or other resources.**

**🔹 Purpose:**

* **To define who can access your instance and where your instance can send data.**
* **Works at instance level, not subnet level.**
* **Only has Allow rules (no explicit deny).**

**🔹 Example:**

| **Direction** | **Type** | **Port** | **Source/Destination** | **Action** |
| --- | --- | --- | --- | --- |
| **Inbound** | **SSH** | **22** | **203.0.113.10/32** | **Allow** |
| **Outbound** | **All** | **All** | **0.0.0.0/0** | **Allow** |

**🧠 Think of Security Group as:**

**“What traffic is allowed to and from your instance.”**

**⚙️ Key Differences:**

| **Feature** | **Subnet** | **Security Group** |
| --- | --- | --- |
| **Layer** | **Network layer (Layer 3)** | **Firewall layer (Layer 4–7)** |
| **Purpose** | **Divides network (public/private)** | **Controls access to resources** |
| **Controls** | **Routing (via route table)** | **Inbound/outbound traffic** |
| **Applied To** | **Entire subnet / resources in it** | **Individual EC2, RDS, etc.** |
| **Rules Type** | **Route-based (IP destination)** | **Allow-only rules (ports/IPs)** |
| **Stateful?** | **Depends on routing** | **✅ Stateful (return traffic auto allowed)** |
| **Used For** | **Organizing network, managing traffic flow** | **Securing resources (who can connect)** |
| **Example** | **Public or private network segment** | **Allow SSH (22), HTTPS (443), etc.** |

**🧩 In Simple Terms**

| **Subnet** | **Security Group** |
| --- | --- |
| **“Where” your instance lives** | **“Who” can talk to your instance** |

**💼 Example from a Real Setup:**

* **Subnet:**
  + **10.0.1.0/24 → Public subnet (has route to Internet Gateway).**
  + **10.0.2.0/24 → Private subnet (no direct internet access).**
* **Security Group:**
  + **Allow inbound SSH (22) from office IP.**
  + **Allow inbound HTTPS (443) from anywhere.**

**So:**

**The Subnet defines the network zone,  
and the Security Group defines access rules for each instance inside it.**

**204\* AWS fargate**

**🧠 What is AWS Fargate?**

**AWS Fargate is a serverless compute engine for containers that works with:**

* **Amazon ECS (Elastic Container Service) and**
* **Amazon EKS (Elastic Kubernetes Service)**

**👉 You can run containers without managing EC2 servers — AWS takes care of provisioning, scaling, and managing the infrastructure.**

**⚙️ How It Works**

**Normally, in ECS or EKS:**

* **You launch containers on EC2 instances (you manage servers).  
  With Fargate:**
* **You just define task definition (CPU, memory, container image, etc.)**
* **Fargate automatically launches containers in isolated compute environments.**
* **No need to create or manage EC2 instances at all.**

**🚀 Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Serverless containers** | **No EC2 management; AWS handles the infrastructure.** |
| **Scalability** | **Automatically scales based on demand.** |
| **Isolation** | **Each task runs in its own lightweight compute environment.** |
| **Pay per use** | **You pay only for vCPU and memory your containers use.** |
| **Integration** | **Works seamlessly with ECS, EKS, CloudWatch, IAM, and ALB.** |
| **Security** | **Built-in IAM role for each task; strong isolation between workloads.** |

**🧩 Use Case Example**

**Let’s say you’re running a microservices-based web app:**

* **Each service runs as a Docker container.**
* **Instead of launching an EC2 cluster, you deploy containers using Fargate.**
* **Fargate automatically provisions compute, runs tasks, scales them, and stops unused resources.**

**Example workflow:**

1. **Push Docker image → Amazon ECR**
2. **Create ECS Task Definition → specify image, CPU, memory**
3. **Run ECS Service → Launch Type = FARGATE**
4. **Attach ALB → distributes traffic to tasks**
5. **Fargate manages scaling and health automatically.**

**🏗️ Architecture Overview**

**Users → ALB → ECS Service (Fargate Launch Type) → Tasks → Containers**

**No EC2 instances are created by you — everything runs on AWS-managed compute.**

**💰 Pricing**

**You pay for:**

* **vCPU and Memory per second**
* **Optional Storage (Ephemeral or EFS)**

**Example:  
If your task uses 1 vCPU and 2GB RAM for 1 hour → you pay only for that usage.**

**🧾 When to Use Fargate**

**✅ Use Fargate when:**

* **You want to avoid EC2 management.**
* **You have microservices or event-driven workloads.**
* **You need quick deployment and automatic scaling.**
* **You want fine-grained security per container.**

**❌ Avoid Fargate when:**

* **You need custom networking or GPU optimization.**
* **You want full control over the host EC2 instance.**

**🔐 Integration Examples**

* **ECS + Fargate + ALB → Web Apps**
* **EKS + Fargate → Serverless Kubernetes Pods**
* **Fargate + CloudWatch → Logs & metrics**
* **Fargate + IAM Roles for Tasks → Secure resource access**

**🧠 In One Line**

**AWS Fargate = Run containers without managing servers (fully serverless container compute engine for ECS & EKS).**

**205\* Single thread in AWS lambda**

**AWS Lambda runs your function code inside a lightweight, isolated runtime environment (called an execution environment).**

**Each Lambda invocation runs in a single-threaded environment by default — meaning:**

* **One request = One thread (one instance of your function).**
* **It cannot process multiple requests concurrently within the same instance.**

**However, AWS automatically scales horizontally — by creating multiple concurrent instances of your Lambda function if multiple events come in.**

**⚙️ Detailed Explanation**

**🧩 1️⃣ Single-threaded Execution**

* **When a Lambda function is invoked, AWS assigns it one vCPU and runs it in a single thread.**
* **Your code executes synchronously — one operation at a time (unless you use async programming inside your code).**

**Example:**

**def handler(event, context):**

**# This Lambda handles one request (single thread)**

**print("Processing event:", event)**

**Even if you use Python’s multithreading module, you’re still limited to the same vCPU — so parallelism is limited.**

**🧩 2️⃣ Concurrency via Multiple Instances**

* **When multiple events are triggered at once, AWS Lambda creates multiple instances of your function.**
* **Each instance still runs single-threaded, but now you have many in parallel.**

**Example:**

* **10 incoming requests → 10 separate Lambda environments.**
* **Each runs independently and scales automatically.**

**🧩 3️⃣ CPU Scaling with Memory**

**Lambda doesn’t let you choose vCPUs directly — they scale proportionally with memory:**

| **Memory (MB)** | **vCPU (approx)** |
| --- | --- |
| **128–1769** | **1 vCPU (single thread)** |
| **1769–10,240** | **Up to 6 vCPUs (multi-threaded allowed)** |

**✅ So, if you allocate more than 1769 MB of memory, your function can use multiple threads or parallel execution (e.g., for data processing).**

**🧩 4️⃣ Implication**

* **Lambda is single-threaded by default.**
* **But can perform asynchronous or multi-threaded tasks if memory allows (especially for compute-intensive jobs).**

**Example (Python multi-threading inside Lambda):**

**import threading**

**def task(n):**

**print(f"Running task {n}")**

**def handler(event, context):**

**threads = []**

**for i in range(3):**

**t = threading.Thread(target=task, args=(i,))**

**threads.append(t)**

**t.start()**

**for t in threads:**

**t.join()**

**return "Completed"**

**✅ Works — but still runs within one Lambda instance and within assigned vCPU limits.**

**🧾 Summary Table**

| **Concept** | **Explanation** |
| --- | --- |
| **Thread Model** | **Single-threaded per invocation** |
| **Concurrency** | **Handled by AWS (multiple Lambda instances)** |
| **vCPU** | **Scales with memory (1–6 vCPUs)** |
| **Multithreading Support** | **Possible inside function, but limited by CPU allocation** |
| **Best Practice** | **Keep each Lambda focused on one task; rely on concurrency, not threading** |

**💡 In Simple Words**

**Each AWS Lambda invocation runs on a single thread, but AWS handles concurrency by running multiple instances in parallel when traffic increases.**

**206\* what is SNS used for?**

🧩 1️⃣ SNS — Full Form

SNS stands for Simple Notification Service.  
It’s a fully managed messaging and notification service by AWS.

⚙️ 2️⃣ Purpose of SNS

SNS is used to send notifications or messages automatically from AWS services or applications to:

* Users
* Applications
* Other AWS services

It’s mainly used for event-based communication — when something happens, SNS notifies other systems immediately.

💬 3️⃣ Simple Example

When an EC2 instance fails or a Lambda function finishes, SNS can automatically send an email, SMS, or trigger another process.

🧩 4️⃣ How SNS Works (Flow)

1. Publisher (Sender) — sends a message to an SNS Topic
2. Topic — acts as a communication channel
3. Subscribers (Receivers) — get notified when a message is published

📦 Message Flow:

Publisher (Lambda, CloudWatch, S3, etc.)

↓

SNS Topic

↓

Subscribers (Email, SMS, Lambda, SQS, HTTPS endpoint, etc.)

🧠 5️⃣ Common Use Cases

| Use Case | Example |
| --- | --- |
| Monitoring Alerts | CloudWatch alarm → SNS → Email/SMS |
| Application Notifications | Order confirmation → SNS → Customer email |
| Fan-out Architecture | SNS topic sends message to multiple SQS queues |
| Serverless workflows | S3 upload → SNS → Lambda function triggers |
| Cross-account notifications | Send alerts between AWS accounts securely |

🔧 6️⃣ SNS Subscription Types

SNS supports multiple protocols for subscribers:

* Email / Email-JSON
* SMS (mobile)
* HTTP / HTTPS endpoint
* AWS Lambda
* Amazon SQS (Simple Queue Service)
* Platform application endpoints (e.g., mobile push notifications)

🪄 7️⃣ Real Example in AWS

* CloudWatch Alarm triggers when EC2 CPU > 80%
* Alarm sends message to SNS topic → “High CPU alert”
* SNS topic has two subscribers:
  + 1 Email to Admin
  + 1 Lambda function to automatically add another instance (auto-heal)

✅ This creates automated incident response with no manual intervention.

🔐 8️⃣ Key Benefits

| Feature | Description |
| --- | --- |
| Scalable | Delivers millions of messages per second |
| Reliable | Highly available and fault tolerant |
| Decoupled | Publishers and subscribers don’t depend on each other |
| Secure | IAM-based access + message encryption |
| Cost-effective | Pay only for what you use |

🗣️ 9️⃣ Short Interview Answer (20 Seconds)

“SNS (Simple Notification Service) is used for sending automated notifications or triggering actions between AWS services.  
It works on a publish–subscribe model, where publishers send messages to topics, and subscribers (like email, SQS, or Lambda) receive them instantly.  
It’s commonly used for alerts, automation, and event-driven communication.”

🧩 10️⃣ Real Example

“In my environment, CloudWatch alarms publish to SNS when an EC2 instance goes down — SNS sends an email to the admin team and triggers a Lambda function to start a replacement instance.”

**207\*** **Can we check disk space n usage from cloud watch ?**

✅ Short Answer

By default, CloudWatch does *not* show disk space or usage metrics for EC2 instances.  
You can see CPU, network, and status checks, but for disk usage, you must install the CloudWatch Agent on the EC2 instance.

🔹 1. What CloudWatch Monitors by Default

When you launch an EC2 instance, CloudWatch automatically collects these basic metrics every 5 minutes (or 1 minute with detailed monitoring):

| Metric | Description |
| --- | --- |
| CPUUtilization | CPU usage percentage |
| NetworkIn/Out | Data in/out |
| DiskReadOps, DiskWriteOps | Number of disk I/O operations |
| StatusCheckFailed | Instance health |

🟡 These are EC2-level metrics, not actual *disk space usage* inside the OS.

🔹 2. To Check Disk Space Usage

To see actual disk space usage (e.g., /, /var/log) —  
you must install and configure the CloudWatch Agent.

✅ Steps:

1. Install the CloudWatch Agent  
   On Linux:
2. sudo yum install amazon-cloudwatch-agent -y
3. Create the configuration file  
   Example /opt/aws/amazon-cloudwatch-agent/bin/config.json:
4. {
5. "metrics": {
6. "append\_dimensions": {
7. "InstanceId": "${aws:InstanceId}"
8. },
9. "metrics\_collected": {
10. "disk": {
11. "measurement": ["used\_percent"],
12. "metrics\_collection\_interval": 60,
13. "resources": ["\*"]
14. }
15. }
16. }
17. }
18. Start the agent
19. sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
20. -a fetch-config -m ec2 -c file:/opt/aws/amazon-cloudwatch-agent/bin/config.json -s
21. Go to CloudWatch → Metrics → CWAgent → Disk  
    You’ll now see disk\_used\_percent and other detailed OS-level metrics.

🔹 3. Optional: Set an Alarm

You can create an alarm to alert if disk usage goes beyond a threshold:

* Metric: disk\_used\_percent
* Threshold: e.g. > 80%
* Action: Send SNS notification or trigger Lambda/AutoHealing

🔹 4. Bonus – EC2 vs EBS Metrics

| Metric Source | What It Shows | Notes |
| --- | --- | --- |
| EC2 metrics | Instance-level stats (CPU, I/O) | Default |
| EBS metrics | Volume-level performance (read/write, latency) | Default |
| CloudWatch Agent | OS-level stats (disk space, memory, swap, processes) | Must be installed manually |

💬 Sample Interview Answer

“By default, CloudWatch only collects basic EC2 metrics like CPU and network usage.  
To monitor actual disk space usage inside the instance, we need to install the CloudWatch Agent and configure it to collect the disk\_used\_percent metric.  
Then we can create alarms if usage crosses thresholds, like 80%.”

**208\* In general section ec2, they have separate section on CPU credits, what are CPU credits on ec2?**

**🧩 What are CPU Credits in EC2?**

**CPU Credits are a way AWS manages CPU performance for burstable instances (T-series: t2, t3, t4g).**

**They allow your EC2 instance to run at a low baseline CPU performance most of the time,  
and “burst” to higher performance when needed — without always paying for full CPU usage.**

**⚙️ How It Works**

**Each burstable instance earns CPU credits when it runs below its baseline performance.**

* **🟢 1 CPU Credit = 1 vCPU running at 100% for 1 minute.**
* **When your instance needs more CPU (temporary high load), it can spend these credits.**
* **When it’s idle again, it earns back credits.**

**💡 Example**

**Let’s take a t3.micro instance:**

* **It has 2 vCPUs.**
* **It earns 12 CPU credits per hour.**
* **Its baseline performance is around 10% of a full vCPU.**

**So:**

* **When idle → earns CPU credits.**
* **When busy (traffic spike, cron job, etc.) → uses credits to burst temporarily to 100% CPU.**

**⚖️ Credit Balance and Modes**

| **Mode** | **Description** |
| --- | --- |
| **Standard mode** | **Instance earns and spends credits normally. When credits run out, performance drops to baseline.** |
| **Unlimited mode** | **Instance can burst beyond credit balance (AWS charges small extra cost for extra CPU).** |

**🧮 You can view credit balance in:  
👉 EC2 Console → Instances → Monitoring tab → “CPU credit balance”.**

**📊 In Simple Terms**

| **Term** | **Meaning** |
| --- | --- |
| **CPU Credit** | **A unit that lets your instance use full CPU for a short period.** |
| **Earn Credits** | **When CPU is below baseline usage.** |
| **Spend Credits** | **When CPU goes above baseline usage.** |
| **Baseline Performance** | **Guaranteed minimal CPU even if credits = 0.** |
| **Unlimited Mode** | **Allows bursting beyond credits (extra charges).** |

**💼 Example Use Case**

**In one of my projects, we used t3.medium instances for lightweight web apps.  
During low traffic → instances earned CPU credits.  
During peak load → they used those credits to handle bursts smoothly.  
This saved costs while keeping good performance.**

**🧠 Quick Summary**

| **Feature** | **Description** |
| --- | --- |
| **Instance Family** | **T2, T3, T4g** |
| **1 CPU Credit** | **1 vCPU @ 100% for 1 min** |
| **Baseline** | **Constant small CPU share** |
| **Burst** | **Uses stored credits for high performance** |
| **Modes** | **Standard / Unlimited** |
| **Benefit** | **Cost-efficient performance for variable workloads** |

**✅ In one line (for interview answer):**

**“CPU credits in EC2 are performance units earned and used by burstable instance types (T-series) to allow short-term bursts of high CPU performance while keeping costs low.”**

**🧠 Definition: EC2 Tenancy Types**

**When you launch an EC2 instance, you choose how the underlying hardware (host) is allocated — this is called tenancy.  
There are 2 main classifications:**

| **Type** | **Description** |
| --- | --- |
| **Shared (Default / Shared Tenancy)** | **Multiple AWS customers share the same physical host, but isolation is achieved at the virtualization layer.** |
| **Dedicated (Dedicated Instances / Dedicated Host)** | **Hardware is physically dedicated to a single customer — no sharing with others.** |

**⚙️ Detailed Difference**

| **Feature** | **Shared Tenancy** | **Dedicated Tenancy** |
| --- | --- | --- |
| **Host Type** | **Shared physical server** | **Dedicated physical server** |
| **Isolation** | **Logical isolation via hypervisor** | **Physical isolation (no other customers on the same hardware)** |
| **Cost** | **💰 Cheaper (default option)** | **💸 More expensive (dedicated hardware)** |
| **Use Case** | **General workloads** | **Compliance, licensing, or security-sensitive workloads** |
| **Performance** | **May vary slightly due to host sharing** | **Consistent (no noisy neighbors)** |
| **Example Setting** | **tenancy = default** | **tenancy = dedicated** |
| **Can mix with VPC?** | **Yes (default for all new VPCs)** | **Yes — set at VPC or instance level** |
| **Licensing Benefits** | **—** | **Allows BYOL (Bring Your Own License) like Oracle, SQL Server, etc.** |

**🔹 Shared Tenancy Example**

**Default mode — when you launch an instance without specifying tenancy:**

**aws ec2 run-instances \**

**--image-id ami-xxxxxxx \**

**--instance-type t3.micro \**

**--key-name MyKey \**

**--subnet-id subnet-xxxxxxx**

**✅ AWS places your instance on a shared physical server — you don’t know or care who else shares that hardware.**

**🔸 Dedicated Tenancy Example**

**If you need isolation (for compliance or licensing):**

**aws ec2 run-instances \**

**--image-id ami-xxxxxxx \**

**--instance-type m5.large \**

**--placement "Tenancy=dedicated"**

**✅ AWS guarantees that the underlying host runs only your instances.**

**🧩 Dedicated Tenancy Has Two Subtypes**

| **Subtype** | **Description** |
| --- | --- |
| **Dedicated Instances** | **Your instances run on dedicated hardware, but AWS manages the host.** |
| **Dedicated Host** | **You get visibility and control over the physical host itself (host ID, sockets, cores). Useful for custom licensing.** |

**🧾 Quick Summary Table**

| **Criteria** | **Shared (Default)** | **Dedicated Instance** | **Dedicated Host** |
| --- | --- | --- | --- |
| **Hardware Sharing** | **Shared with others** | **Dedicated to you** | **Dedicated to you** |
| **Cost** | **Lowest** | **Higher** | **Highest** |
| **Compliance** | **Standard workloads** | **Compliant workloads** | **Strict compliance (BYOL)** |
| **Control** | **No visibility** | **AWS managed** | **Full control** |
| **Use Case** | **Web, dev, test** | **Prod with compliance** | **Licensing-heavy apps (Oracle, SQL Server)** |

**💡 In Simple Words**

* **Shared tenancy → You share the physical server with other AWS users (default, low-cost).**
* **Dedicated tenancy → You get your own physical hardware for compliance, licensing, or performance isolation.**

**209\* There are 2 major classification of ec2s, that is shared and dedicated. What is the diff between them?**

**🧩 1️⃣ Basic Definition**

**EC2 instances can be launched on shared or dedicated hardware — this is called tenancy in AWS.**

| **Tenancy Type** | **Description** |
| --- | --- |
| **Shared** | **Multiple AWS customers share the same underlying physical host (hardware). Each gets their own isolated virtual machine (VM).** |
| **Dedicated** | **Your EC2 instances run on hardware reserved exclusively for your account — no other AWS customer shares that physical server.** |

**🧠 2️⃣ Shared Tenancy (Default)**

**✅ Features:**

* **Default mode for all EC2 instances**
* **Multiple AWS customers share the same physical host**
* **Each instance is securely isolated using the Xen or Nitro hypervisor**
* **Cost-effective (cheaper)**
* **Ideal for most workloads**

**💡 Example:**

**aws ec2 run-instances --instance-type t3.medium --image-id ami-xyz**

**(no tenancy specified → shared tenancy)**

**💬 Used For:**

* **General workloads**
* **Dev/test environments**
* **Non-compliance or low-security applications**

**🔒 3️⃣ Dedicated Tenancy**

**There are two ways to get dedicated hardware in AWS:**

| **Type** | **Description** |
| --- | --- |
| **Dedicated Instance** | **Runs on hardware that is physically isolated — only your account’s instances share that host. You don’t control which host.** |
| **Dedicated Host** | **Gives you full control of a specific physical server — useful for licensing (BYOL), compliance, and audit requirements.** |

**⚙️ Dedicated Instance**

* **AWS automatically assigns hardware that’s used only by your account.**
* **You don’t manage the host directly.**
* **Still billed per-instance.**

**Example:**

**aws ec2 run-instances --instance-type m5.large --tenancy dedicated**

**⚙️ Dedicated Host**

* **Gives full control of physical server.**
* **You can see and manage sockets, cores, VMs.**
* **Required for compliance or per-socket software licenses (like Oracle, SQL Server, etc.)**
* **Billed per host, not per instance.**

**Example:**

**aws ec2 allocate-hosts --instance-type m5.large --quantity 1**

**💰 4️⃣ Cost Difference**

| **Tenancy Type** | **Cost** | **Best For** |
| --- | --- | --- |
| **Shared** | **Cheapest** | **Regular workloads** |
| **Dedicated Instance** | **More expensive** | **Security/compliance workloads** |
| **Dedicated Host** | **Highest** | **Licensing & audit requirements** |

**🧩 5️⃣ Security & Compliance Difference**

| **Feature** | **Shared** | **Dedicated** |
| --- | --- | --- |
| **Hardware shared?** | **Yes** | **No** |
| **Physical isolation** | **❌** | **✅** |
| **Custom host control** | **❌** | **✅ (Dedicated Host only)** |
| **Best for** | **Cost optimization** | **Compliance / Licensing** |
| **Example** | **Web app** | **Banking / Healthcare / Govt apps** |

**🗣️ 6️⃣ Short Interview Answer (20 seconds)**

**“EC2 instances can run on shared or dedicated hardware.  
In shared tenancy, your instance shares the physical host with others but remains logically isolated — this is default and cost-effective.  
In dedicated tenancy, your instances run on hardware reserved only for your account, which provides extra compliance, licensing, and security control.”**

**🧩 7️⃣ Bonus Tip (for extra points)**

**“If you need full host-level visibility or want to bring your own software licenses (like Oracle or SQL Server), use Dedicated Host.  
For simple isolation without host management, Dedicated Instance is enough.”**

**212\* How you monitor complete e commerce website.**

🧩 1️⃣ Basic Definition

EC2 instances can be launched on shared or dedicated hardware — this is called tenancy in AWS.

| Tenancy Type | Description |
| --- | --- |
| Shared | Multiple AWS customers share the same underlying physical host (hardware). Each gets their own isolated virtual machine (VM). |
| Dedicated | Your EC2 instances run on hardware reserved exclusively for your account — no other AWS customer shares that physical server. |

🧠 2️⃣ Shared Tenancy (Default)

✅ Features:

* Default mode for all EC2 instances
* Multiple AWS customers share the same physical host
* Each instance is securely isolated using the Xen or Nitro hypervisor
* Cost-effective (cheaper)
* Ideal for most workloads

💡 Example:

aws ec2 run-instances --instance-type t3.medium --image-id ami-xyz

(no tenancy specified → shared tenancy)

💬 Used For:

* General workloads
* Dev/test environments
* Non-compliance or low-security applications

🔒 3️⃣ Dedicated Tenancy

There are two ways to get dedicated hardware in AWS:

| Type | Description |
| --- | --- |
| Dedicated Instance | Runs on hardware that is physically isolated — only your account’s instances share that host. You don’t control which host. |
| Dedicated Host | Gives you full control of a specific physical server — useful for licensing (BYOL), compliance, and audit requirements. |

⚙️ Dedicated Instance

* AWS automatically assigns hardware that’s used only by your account.
* You don’t manage the host directly.
* Still billed per-instance.

Example:

aws ec2 run-instances --instance-type m5.large --tenancy dedicated

⚙️ Dedicated Host

* Gives full control of physical server.
* You can see and manage sockets, cores, VMs.
* Required for compliance or per-socket software licenses (like Oracle, SQL Server, etc.)
* Billed per host, not per instance.

Example:

aws ec2 allocate-hosts --instance-type m5.large --quantity 1

💰 4️⃣ Cost Difference

| Tenancy Type | Cost | Best For |
| --- | --- | --- |
| Shared | Cheapest | Regular workloads |
| Dedicated Instance | More expensive | Security/compliance workloads |
| Dedicated Host | Highest | Licensing & audit requirements |

🧩 5️⃣ Security & Compliance Difference

| Feature | Shared | Dedicated |
| --- | --- | --- |
| Hardware shared? | Yes | No |
| Physical isolation | ❌ | ✅ |
| Custom host control | ❌ | ✅ (Dedicated Host only) |
| Best for | Cost optimization | Compliance / Licensing |
| Example | Web app | Banking / Healthcare / Govt apps |

🗣️ 6️⃣ Short Interview Answer (20 seconds)

“EC2 instances can run on shared or dedicated hardware.  
In shared tenancy, your instance shares the physical host with others but remains logically isolated — this is default and cost-effective.  
In dedicated tenancy, your instances run on hardware reserved only for your account, which provides extra compliance, licensing, and security control.”

🧩 7️⃣ Bonus Tip (for extra points)

“If you need full host-level visibility or want to bring your own software licenses (like Oracle or SQL Server), use Dedicated Host.  
For simple isolation without host management, Dedicated Instance is enough.”

**213\* How you provide authentication and authorisation for users using your e commerce website**

Authentication & Authorization for an e-commerce website

Below is a compact, practical design you can implement end-to-end — covering user flows, data model, tokens/sessions, security hardening, role/permission model, monitoring, and tradeoffs.

1) High-level approach

* Authentication (AuthN): verify *who* the user is. Use secure password storage + optional social/OAuth + multi-factor auth (MFA).
* Authorization (AuthZ): control *what* the authenticated user can do. Use Role-Based Access Control (RBAC) for most e-commerce needs (Customer, Seller, Admin), optionally ABAC for complex rules.
* Protect all endpoints & UI with TLS, input validation, CSRF/XSS mitigation, rate limits, and audit logging.

2) Core flows (concrete, recommended)

Registration

1. Client submits email, password (and optional name/phone).
2. Server validates, enforces password policy.
3. Hash password with bcrypt/argon2id + unique salt. Store only hash.
4. Send email verification token (short lived, signed). Account inactive until verified.

Login

Option A — Secure HttpOnly cookie + short-lived session token

* Authenticate credentials → create server session (or issue JWT stored in an HttpOnly, Secure, SameSite cookie).
* Cookie contains session id or access token; refresh token also delivered in HttpOnly cookie if using short access tokens.
* Pros: safe from XSS (cookie flagged), simpler logout/revocation on server.

Option B — Access + Refresh JWT (Bearer tokens)

* Issue short-lived access JWT (e.g., 5–15 min) + long-lived refresh token (rotate/replace on use).
* Store refresh token hashed in DB for revocation.
* Pros: stateless scaling; Cons: careful revocation logic required, more XSS risk if stored in localStorage.

Recommendation: Use HttpOnly cookies for browser clients; use OAuth2 client credentials / API keys for server-to-server.

MFA (optional but recommended)

* Offer TOTP (Authenticator app) or SMS (SMS less secure) for high-value actions (password change, large order changes).
* Enforce MFA for admin roles.

Password reset

* Request reset → send signed, single-use token link (short expiry). On use, invalidate token and require strong new password.

Logout & Revocation

* Invalidate session server side or delete refresh token record. Rotate tokens on sensitive changes (password change, MFA change).

3) Authorization design (RBAC example)

* Roles: guest, customer, seller, support\_agent, admin.
* Permissions examples:
  + customer: browse, place\_order, view\_own\_orders, add\_to\_cart
  + seller: manage\_products, view\_own\_orders
  + support\_agent: view\_orders, issue\_refund (limited)
  + admin: full access

Store:

users(id, email, password\_hash, is\_verified, role\_id, ... )

roles(id, name)

permissions(id, name)

role\_permissions(role\_id, permission\_id)

Check perms at API gateway or inside services. For fine-grained checks combine role + resource ownership (e.g., order.user\_id == current\_user.id).

4) API / token protection & best practices

* Always use HTTPS everywhere.
* Use HttpOnly, Secure, SameSite=Strict/Lax cookies for browser sessions.
* Implement CSRF tokens for state-changing forms if not using SameSite Strict or if using cookies.
* Protect against XSS to prevent token theft.
* Rate-limit auth endpoints (login, password reset) by IP and account.
* Account lockout / exponential backoff on repeated failed logins.
* Use strong signing keys (rotate keys regularly), keep secrets in vault (AWS Secrets Manager, HashiCorp Vault).
* Implement token rotation and refresh token revocation lists.

5) Infrastructure / scalability & tooling

* Session store: Redis (fast, TTL, supports revoke via key deletion). For JWT stateless approach, keep a revocation/blacklist for compromised tokens.
* Identity providers: consider managed services to speed up (AWS Cognito, Auth0, Okta) — they handle MFA, social login, compliance.
* For social login: support OAuth2 providers (Google, Facebook, Apple).
* Use API Gateway / WAF to block attacks and implement centralized authZ checks for microservices.

6) Security & compliance checklist (must-haves)

* Passwords hashed with Argon2id or bcrypt (work factor tuned).
* TLS 1.2+/HSTS on all endpoints.
* MFA for admin and optional for customers.
* Audit logs for auth events (login, fail, password reset, role change).
* Monitor suspicious patterns and notify users of new device sign-ins.
* Protect payment workflows to be PCI-DSS compliant (never store card data unless necessary; use PCI-certified processors).
* Penetration testing and periodic secret/key rotation.

7) Example: Minimal endpoint contract (cookie session flow)

* POST /api/register → create user, send verification email.
* GET /api/verify?token=... → activate account.
* POST /api/login → set Set-Cookie: session=<session-id>; HttpOnly; Secure; SameSite=Strict
* POST /api/logout → delete session server side and clear cookie.
* POST /api/orders → require auth, check role == customer and order.user\_id == current\_user.id.
* GET /api/admin/reports → require role == admin.

8) Monitoring, logging & incident response

* Log auth events (successful/failed logins, password resets, MFA changes) with IP, user agent.
* Alert on brute force attempts, token reuse, impossible travel (logins from distant countries in short time).
* Provide user controls: devices/sessions list, ability to force logout everywhere.

9) Tradeoffs & when to choose what

* Stateless JWTs → simpler scaling, fewer DB hits, but harder to revoke quickly.
* Stateful sessions (Redis) → easy immediate revocation and session management; requires session store.
* Managed IdP → quicker to implement, strong security features, but vendor lock-in and cost.

10) Quick prioritized implementation plan (MVP)

1. Implement secure registration/login with email verification.
2. Use HttpOnly cookie sessions stored in Redis.
3. Build RBAC for customer, seller, admin.
4. Add password reset and login audit logs.
5. Add MFA for admins and optional for customers.
6. Harden with rate limits, CSRF protection, TLS and monitoring.

**215\* How monitoring using promotheus and grafana works**

**How Monitoring Works Using Prometheus and Grafana**

**1️⃣ Purpose**

**Together, Prometheus and Grafana are used to collect, store, analyze, and visualize metrics from your servers, containers, and applications.**

**2️⃣ Components and Flow**

**Here’s the flow step-by-step:**

**Step 1: Prometheus collects metrics**

* **Prometheus is a time-series database + monitoring system.**
* **It pulls metrics from targets (like EC2 instances, Kubernetes pods, databases, etc.).**
* **Targets must expose metrics endpoints (usually /metrics URL) in Prometheus format (key-value pairs).**

**🔹 Example:**

**http\_requests\_total{method="GET", status="200"} 1027**

**cpu\_usage\_seconds\_total 2456.3**

**Step 2: Prometheus stores metrics**

* **Prometheus stores all collected metrics in its local time-series database (TSDB).**
* **Each metric is stored along with a timestamp and labels (metadata like instance name, job name, etc.).**

**Step 3: Alerting (Optional)**

* **Prometheus has a built-in Alertmanager component.**
* **You define alert rules in Prometheus configuration, like:**
* **- alert: HighCPUUsage**
* **expr: cpu\_usage\_seconds\_total > 80**
* **for: 2m**
* **labels:**
* **severity: warning**
* **annotations:**
* **description: "CPU usage is above 80% for 2 minutes"**
* **When triggered, Alertmanager sends notifications to:**
  + **Email**
  + **Slack**
  + **PagerDuty**
  + **Webhooks**

**Step 4: Grafana for Visualization**

* **Grafana connects to Prometheus as a data source.**
* **Grafana uses Prometheus’s query language (PromQL) to fetch metrics.**
* **You can build dashboards showing:**
  + **CPU usage**
  + **Memory consumption**
  + **Network traffic**
  + **Application request latency**
  + **Error rates**

**Example PromQL:**

**rate(http\_requests\_total[5m])**

**→ Shows request rate over last 5 minutes.**

**Step 5: Dashboards and Alerts**

* **Grafana provides beautiful interactive dashboards.**
* **You can set Grafana alerts directly on dashboard panels.**
* **Alerts can notify you when metrics exceed thresholds.**

**3️⃣ Example Architecture**

**[Applications/Servers]**

**|**

**Expose metrics at /metrics**

**|**

**<---- Prometheus (scrapes data every 15s)**

**|**

**Stores metrics in time-series DB**

**|**

**[Alertmanager] --> Email / Slack alerts**

**|**

**Grafana (queries Prometheus)**

**|**

**Dashboards + Visual alerts**

**4️⃣ Use Cases in Real Systems**

* **Monitor EC2 instances (CPU, RAM, Disk, Network)**
* **Track Kubernetes pod status**
* **Measure HTTP request latency or errors**
* **Monitor database performance (MySQL, PostgreSQL exporters)**
* **Observe microservices health**

**5️⃣ Key Advantages**

**✅ Open-source and free  
✅ Pull model (Prometheus controls what and when to collect)  
✅ Flexible queries (PromQL)  
✅ Easy visualization (Grafana)  
✅ Integration with cloud exporters (AWS, Kubernetes, Docker, etc.)**

**6️⃣ Common Exporters Used**

| **Service** | **Exporter** |
| --- | --- |
| **Linux servers** | **Node Exporter** |
| **MySQL** | **MySQL Exporter** |
| **PostgreSQL** | **PostgreSQL Exporter** |
| **Nginx** | **Nginx Exporter** |
| **Docker** | **cAdvisor** |
| **AWS** | **CloudWatch Exporter** |
| **Kubernetes** | **kube-state-metrics** |

**✅ In short**

**Prometheus = Collect + Store + Alert  
Grafana = Query + Visualize + Analyze**

**Together, they give real-time observability for your entire infrastructure.**

**216\* How you will give access using Iam to a user who wants to access only Jenkins master**

**🧩 Scenario:**

**You have a Jenkins master (hosted on an EC2 or other server) and you want to give IAM access to a user such that:**

* **The user can access only the Jenkins master instance**
* **The user cannot see or modify any other AWS resources**

**✅ Step-by-step solution:**

**Step 1: Identify the Jenkins master resource**

**If your Jenkins master runs on an EC2 instance, note its:**

* **Instance ID → e.g., i-0abcd1234efgh5678**
* **Region → e.g., ap-south-1**

**Step 2: Create a custom IAM policy**

**Create a policy that gives access only to this specific EC2 instance.**

**Example JSON policy:**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Sid": "AllowViewSpecificInstance",**

**"Effect": "Allow",**

**"Action": [**

**"ec2:DescribeInstances",**

**"ec2:GetConsoleOutput"**

**],**

**"Resource": "\*"**

**},**

**{**

**"Sid": "AllowAccessToSpecificInstance",**

**"Effect": "Allow",**

**"Action": [**

**"ec2:StartInstances",**

**"ec2:StopInstances",**

**"ec2:RebootInstances",**

**"ec2:DescribeInstances"**

**],**

**"Resource": "arn:aws:ec2:ap-south-1:123456789012:instance/i-0abcd1234efgh5678"**

**}**

**]**

**}**

**📝 Replace the Region, Account ID, and Instance ID with your actual values.**

**This policy ensures the user can view and manage only that Jenkins EC2 instance.**

**Step 3: Create a new IAM user**

* **Go to IAM → Users → Add user**
* **Choose access type:**
  + **✅ AWS Management Console access (so they can log in to AWS)**
* **Attach the custom policy you created above**

**Step 4: (Optional) Restrict login further**

**If you don’t want the user to log in and see other services:**

* **Use Service Control Policies (SCPs) if you’re in an AWS Organization**
* **Or attach an additional deny policy to block other services:**

**{**

**"Effect": "Deny",**

**"Action": "\*",**

**"NotAction": [**

**"ec2:\*"**

**],**

**"Resource": "\*"**

**}**

**This ensures they can only interact with EC2.**

**Step 5: Jenkins-level access**

**If you meant access inside Jenkins (not AWS console):**

* **Create a Jenkins user via Jenkins UI (Manage Jenkins → Manage Users)**
* **Assign that user permissions using Role-Based Strategy Plugin**
  + **Role: e.g., jenkins-admin**
  + **Scope: only master node**

**💡 Summary**

| **Access Type** | **How to Give Access** | **Notes** |
| --- | --- | --- |
| **AWS Console (EC2 only)** | **Create IAM user + custom EC2 policy** | **Restricts AWS access** |
| **Jenkins UI** | **Create Jenkins user + role strategy plugin** | **Restricts Jenkins-level access** |

**217\* how do u configure cloudfront**

**🔹 What is CloudFront?**

**Amazon CloudFront is a Content Delivery Network (CDN) that delivers your content (like website data, images, videos, APIs) to users with low latency and high transfer speed through a network of edge locations worldwide.**

**🔹 Steps to Configure CloudFront**

**Step 1: Open CloudFront Console**

**Go to AWS Management Console → CloudFront → Create Distribution.**

**Step 2: Choose the Origin**

**You must specify where your content comes from (Origin).  
Common origins:**

* **S3 Bucket → for static websites or files**
* **Application Load Balancer or EC2 instance → for dynamic content**
* **Custom Origin (any public HTTP server)**

**Example:**

* **Origin domain: mybucket.s3.amazonaws.com**
* **Enable origin access control (OAC) if using private S3 bucket.**

**Step 3: Configure Default Cache Behavior**

**Define how CloudFront should handle requests:**

* **Viewer protocol policy → Redirect HTTP to HTTPS (recommended)**
* **Allowed HTTP methods → GET, HEAD (or include POST, PUT if needed)**
* **Cache policy → Use default caching or create custom policy**
* **Origin request policy → Control what headers, cookies, and query strings go to origin**

**Step 4: Set Distribution Settings**

* **Price class → Choose regions (e.g., only North America & Europe to reduce cost)**
* **Alternate domain name (CNAME) → e.g., cdn.example.com**
* **SSL Certificate → Use ACM certificate for HTTPS**
* **Default Root Object → e.g., index.html for websites**

**Step 5: Create the Distribution**

**Click Create Distribution.  
It will take a few minutes (status: *In Progress*) before becoming *Deployed*.**

**Step 6: Update DNS**

**Once deployed, CloudFront gives you a Domain Name like:**

**d1234567890.cloudfront.net**

**In Route 53 (or DNS provider), create a CNAME record:**

**cdn.example.com → d1234567890.cloudfront.net**

**🔹 Optional: Security & Performance**

* **Restrict viewer access → Signed URLs / Signed Cookies**
* **Enable WAF → Protect against web attacks**
* **Enable Logging → Store access logs in S3**
* **Enable Compression → Optimize bandwidth usage**

**✅ Example Use Case**

**If you have a static website hosted in S3, and you want to serve it globally:**

1. **Create S3 bucket and upload files.**
2. **Make the bucket private.**
3. **Create CloudFront distribution with the S3 bucket as origin.**
4. **Enable Origin Access Control (OAC).**
5. **Update DNS to point to CloudFront.  
   Now your site loads faster from the nearest edge location.**

**218\* You are building a solution for a customer to extend their on-premises data center to AWS.**

**1. Objective**

**The goal is to securely connect the customer’s on-premises data center to their AWS VPC, enabling hybrid workloads — i.e., resources can communicate privately as if they’re on the same network.**

**2. Options to Connect On-Prem to AWS**

| **Option** | **Description** | **When to Use** |
| --- | --- | --- |
| **AWS Site-to-Site VPN** | **Encrypted connection over the public internet between on-premises router/firewall and AWS Virtual Private Gateway** | **Quick setup, lower cost, but latency depends on internet** |
| **AWS Direct Connect (DX)** | **Dedicated private fiber link from on-premises data center to AWS** | **For high throughput, consistent low latency** |
| **Hybrid (DX + VPN)** | **Direct Connect for primary link, VPN as backup** | **For enterprise-grade, highly available setups** |

**3. Core Components in AWS**

* **VPC (Virtual Private Cloud) – your AWS network.**
* **VGW (Virtual Private Gateway) – AWS end of VPN/Direct Connect.**
* **Customer Gateway (CGW) – represents your on-prem router/firewall.**
* **Route Tables – to route traffic between subnets and the gateway.**
* **Transit Gateway (optional) – if connecting multiple VPCs or hybrid regions.**

**4. Steps for Site-to-Site VPN Setup**

1. **Create a VPC in AWS.**
2. **Create a Virtual Private Gateway (VGW) and attach it to the VPC.**
3. **Create a Customer Gateway (CGW) — provide your on-prem public IP and routing details.**
4. **Create a VPN Connection between VGW and CGW.**
5. **Update Route Tables in VPC and on-prem router to send private traffic through the VPN.**
6. **Test the Connectivity using ping or traceroute between EC2 and on-prem systems.**

**5. Steps for Direct Connect Setup**

1. **Order a Direct Connect connection through the AWS console.**
2. **Connect to the AWS DX location (colocation facility) using a dedicated line.**
3. **Create a Virtual Interface (VIF) — public or private.**
4. **Associate the VIF with VGW or Transit Gateway.**
5. **Configure BGP for dynamic routing.**
6. **Optionally, set up VPN as a backup in case DX fails.**

**6. Security Considerations**

* **Use IPSec encryption for VPN traffic.**
* **Enable Network ACLs and Security Groups for least-privilege communication.**
* **Use AWS Identity and Access Management (IAM) for user access control.**
* **Enable CloudTrail and VPC Flow Logs for monitoring.**

**7. Monitoring and Maintenance**

* **Monitor connection health using CloudWatch metrics.**
* **Use AWS Config to track changes in VPC components.**
* **Setup failover between VPN and Direct Connect (BGP routing priority).**

**8. Example Architecture Diagram (Conceptual)**

**+-----------------------------+ +-------------------------+**

**| On-Prem Data Center | | AWS Cloud |**

**| +-----------------------+ | | +-------------------+ |**

**| | Router / Firewall |<--IPSec VPN--->| Virtual Private GW | |**

**| | Customer Gateway (CGW)| | | | or Transit GW | |**

**| +-----------------------+ | | +---------+---------+ |**

**| Internal LAN | | | | |**

**+-----------------------------+ | EC2, RDS, etc. |**

**+-------------------------+**

**Answer Summary**

**To extend an on-premises data center to AWS, use Site-to-Site VPN or AWS Direct Connect (or both) to establish secure hybrid connectivity between the on-prem network and your AWS VPC. Configure routing, security, and monitoring to ensure reliable and secure operations.**

**Top of Form**

**Bottom of Form**

**219\* The customer requires a 50-Mbps dedicated and private connection to their VPC. Which**

**AWS Direct Connect provides a dedicated, private network connection from your on-premises data center to AWS.  
It does not use the public internet, giving:**

* **Lower latency**
* **Higher security**
* **More consistent network performance**

**You can choose connection speeds from 50 Mbps up to 100 Gbps, depending on your requirements and AWS location availability.**

**Alternative (for small setups or testing):**

**If a dedicated line isn’t required and secure connectivity over the internet is acceptable, you could use:**

* **AWS Site-to-Site VPN  
  ➜ Encrypted over the internet  
  ➜ Quick to set up  
  ➜ Not dedicated (shared internet bandwidth)**

**Summary:**

| **Requirement** | **Recommended Service** |
| --- | --- |
| **50 Mbps dedicated & private** | **AWS Direct Connect** |
| **Secure but over internet** | **AWS Site-to-Site VPN** |

**220\* AWS product or feature satisfies this requirement?**

That part alone (“AWS product or feature satisfies this requirement”) usually follows a scenario or requirement (for example: *“The customer requires a 50-Mbps dedicated and private connection to their VPC”*).

Once you share the complete sentence or scenario before this line, I can tell you exactly which AWS product or feature satisfies it (e.g., AWS Direct Connect, VPN, CloudFront, etc.).

**221\* Which DNS name can only be resolved within Amazon EC2?**

**Explanation:**

**In Amazon EC2, each instance can have two types of DNS names:**

| **Type** | **Example** | **Resolvable From** | **Description** |
| --- | --- | --- | --- |
| **Public DNS name** | **ec2-203-0-113-25.compute-1.amazonaws.com** | **Internet + within AWS** | **Resolves to the instance’s public IP address. Available only if the instance has a public IP or Elastic IP.** |
| **Private DNS name** | **ip-10-0-0-25.ec2.internal** | **Only within the VPC / EC2 network** | **Resolves to the instance’s private IP address and is not accessible from the Internet.** |

**222\* What does Amazon CloudFormation provide?**

**CloudFormation lets you describe your entire AWS infrastructure (VPCs, EC2s, S3 buckets, RDS, IAM roles, etc.) in a template written in YAML or JSON — and then it automatically provisions those resources for you.**

**🔹 What CloudFormation Provides:**

| **Feature** | **Description** |
| --- | --- |
| **1. Infrastructure as Code (IaC)** | **Define and deploy your AWS resources using code (YAML/JSON templates).** |
| **2. Automated Resource Provisioning** | **Automatically creates, updates, or deletes AWS resources in the correct order.** |
| **3. Consistency and Repeatability** | **Ensures the same environment (like dev, test, prod) can be created reliably multiple times.** |
| **4. Dependency Management** | **Handles resource dependencies (for example, creates VPC before EC2).** |
| **5. Change Sets** | **Lets you preview what changes will be made before applying updates.** |
| **6. Stack Management** | **Groups related resources as a stack — you can create, update, or delete an entire stack at once.** |
| **7. Rollback on Failure** | **If deployment fails, CloudFormation automatically rolls back to the previous stable state.** |
| **8. Integration with Other AWS Services** | **Works with services like EC2, S3, RDS, IAM, Lambda, CloudWatch, etc.** |
| **9. Custom Resources** | **You can extend CloudFormation with Lambda functions to provision non-AWS or custom services.** |

**🔹 Example:**

**A simple CloudFormation YAML template to create an S3 bucket:**

**Resources:**

**MyBucket:**

**Type: AWS::S3::Bucket**

**Properties:**

**BucketName: my-sample-bucket-12345**

**When you deploy this template, CloudFormation automatically:**

* **Creates an S3 bucket named my-sample-bucket-12345**
* **Tracks it as part of a *stack* so you can delete it later with one command.**

**🔹 Real-world Use Cases**

* **Automate full environment setup (VPC, EC2, RDS, ALB)**
* **CI/CD pipelines for infrastructure**
* **Disaster recovery — recreate the same infra in another region**
* **Multi-account, multi-region provisioning**

**🔹 Summary**

**Amazon CloudFormation provides a way to model, provision, and manage AWS infrastructure as code — automatically, consistently, and repeatably.**

**223\* diffrence between the clformation and terraform**

**🧩 Definition**

| **Tool** | **Description** |
| --- | --- |
| **AWS CloudFormation** | **Native AWS service to automate provisioning of AWS infrastructure using YAML/JSON templates.** |
| **Terraform** | **Open-source tool by HashiCorp used to automate infrastructure provisioning across multiple cloud providers (AWS, Azure, GCP, etc.) using HCL (HashiCorp Configuration Language).** |

**⚙️ Key Differences**

| **Feature** | **CloudFormation** | **Terraform** |
| --- | --- | --- |
| **Provider Support** | **AWS only** | **Multi-cloud (AWS, Azure, GCP, etc.)** |
| **Language** | **JSON / YAML** | **HCL (HashiCorp Configuration Language)** |
| **Tool Type** | **AWS-native service** | **Third-party open-source tool** |
| **State Management** | **Managed automatically by AWS** | **Uses a state file (terraform.tfstate) to track infrastructure** |
| **Execution** | **Changes managed by AWS CloudFormation service** | **Terraform CLI applies changes locally via API calls** |
| **Modularity** | **Supports nested stacks** | **Supports modules for reusable infrastructure** |
| **Drift Detection** | **Yes (detects manual changes)** | **Yes (via terraform plan)** |
| **Rollback Support** | **Automatic rollback on failure** | **Manual rollback (you re-apply the previous state)** |
| **Cost** | **Free, pay only for AWS resources used** | **Free (open-source)** |
| **Learning Curve** | **Easier for AWS-only environments** | **Slightly more complex but flexible** |

**🧠 Example Use Cases**

| **Use Case** | **Recommended Tool** |
| --- | --- |
| **Only AWS infrastructure** | **CloudFormation** |
| **Multi-cloud or hybrid setup** | **Terraform** |
| **You want AWS-managed automation** | **CloudFormation** |
| **You want more flexibility, version control, and reuse** | **Terraform** |

**💡 Example: Create an EC2 Instance**

**CloudFormation (YAML):**

**Resources:**

**MyInstance:**

**Type: AWS::EC2::Instance**

**Properties:**

**InstanceType: t2.micro**

**ImageId: ami-0abcd1234efgh5678**

**Terraform (HCL):**

**resource "aws\_instance" "my\_instance" {**

**ami = "ami-0abcd1234efgh5678"**

**instance\_type = "t2.micro"**

**}**

**✅ Summary**

| **Aspect** | **CloudFormation** | **Terraform** |
| --- | --- | --- |
| **Scope** | **AWS-only** | **Multi-cloud** |
| **Language** | **JSON/YAML** | **HCL** |
| **State Handling** | **Managed by AWS** | **Manual (state file)** |
| **Rollback** | **Automatic** | **Manual** |
| **Flexibility** | **Limited to AWS** | **Very flexible and reusable** |

**224\* have you deployed in diffrent regions ? how many availability zone ur using for application? why excatly those azs only**

**✅ Answer:  
Yes, we’ve deployed our application in multiple AWS regions for high availability, disaster recovery, and low latency.  
Our primary region is ap-south-1 (Mumbai), and we have a secondary/DR region in ap-southeast-1 (Singapore).  
We replicate critical data (like S3, RDS snapshots, and AMIs) to the secondary region using Cross-Region Replication (CRR) and AWS Backup.**

**🔹 Q: How many Availability Zones are you using for your application?**

**✅ Answer:  
We are using 2 Availability Zones (AZs) within our primary region — for example, ap-south-1a and ap-south-1b.**

**Each AZ hosts:**

* **1 EC2 instance (App server)**
* **1 RDS instance (multi-AZ setup) for database redundancy**
* **Load Balancer distributes traffic across both AZs**

**This ensures that even if one AZ fails, our application continues running from the other AZ without downtime.**

**🔹 Q: Why exactly those AZs only?**

**✅ Answer:  
We chose those AZs based on:**

1. **High Availability:  
   Using multiple AZs within the same region protects against data center failures.**
2. **Low Latency Communication:  
   AZs within the same region are interconnected with high-speed, low-latency links, ensuring smooth replication and failover.**
3. **AWS Recommendations & Testing:  
   During testing, these AZs showed better network performance and lower latency for our target user base.**
4. **Cost & Resource Availability:  
   Some AZs occasionally have capacity constraints (like specific EC2 instance types not available), so we picked AZs that consistently support our instance types and services.**

**✅ Example Summary Answer:**

**“Yes, our application is deployed in multiple AWS regions for redundancy — primarily in ap-south-1 (Mumbai) and DR in ap-southeast-1 (Singapore).  
Within Mumbai, we use 2 Availability Zones — ap-south-1a and ap-south-1b — to ensure high availability and fault tolerance.  
We selected these AZs based on performance, latency, and AWS’s recommendation for even distribution across the region.”**

**225\* distastive recovery in aws ?**

**Definition**

**Disaster Recovery (DR) in AWS refers to the strategy and process of restoring systems and data after an outage, failure, or disaster (like data loss, region failure, or natural disaster).**

**It ensures business continuity — minimizing downtime (RTO) and data loss (RPO).**

**Key Terms**

| **Term** | **Meaning** |
| --- | --- |
| **RTO (Recovery Time Objective)** | **Maximum time the system can be down before it must be restored.** |
| **RPO (Recovery Point Objective)** | **Maximum acceptable amount of data loss measured in time (e.g., last 5 mins of data).** |

**AWS Disaster Recovery Strategies**

**AWS defines 4 main DR patterns, depending on cost vs recovery speed:**

| **Strategy** | **Description** | **RTO** | **RPO** | **Cost** |
| --- | --- | --- | --- | --- |
| **1. Backup & Restore** | **Store backups (EBS snapshots, RDS snapshots, S3) and restore after disaster.** | **High (hours)** | **High** | **💲 Low** |
| **2. Pilot Light** | **Keep core components (like databases) running in AWS, start other servers only during disaster.** | **Medium** | **Low** | **💲💲 Medium** |
| **3. Warm Standby** | **Run a scaled-down version of full production in AWS; scale up during disaster.** | **Low** | **Low** | **💲💲💲 Higher** |
| **4. Multi-Site / Active-Active** | **Full production runs in both AWS and on-prem or multiple regions; traffic load-balanced.** | **Very Low** | **Very Low** | **💲💲💲💲 High** |

**Example Setup**

**Pilot Light Example**

* **Keep database (RDS) continuously replicated to AWS.**
* **Keep minimal EC2s with app code in stopped state.**
* **During disaster → start EC2s, scale Auto Scaling Group, and route traffic via Route 53 failover.**

**AWS Services Commonly Used for DR**

| **Service** | **Purpose** |
| --- | --- |
| **S3 + Glacier** | **Backup and archival storage** |
| **EBS Snapshots** | **Volume-level backups** |
| **RDS Read Replica / Cross-Region Replica** | **Database DR** |
| **Route 53** | **DNS-based failover routing** |
| **CloudFormation / Terraform** | **Infrastructure recovery automation** |
| **AWS Backup** | **Centralized backup management** |
| **CloudEndure Disaster Recovery (AWS Elastic DR)** | **Automated, near real-time replication and recovery** |

**Example Architecture (Warm Standby)**

**On-Prem Data Center AWS Cloud**

**+----------------------+ +------------------------+**

**| App Servers (Full) | ---> | Scaled-down EC2 |**

**| Database Primary | ---> | RDS Replica |**

**| DNS Failover via | <--- | Route 53 Health Check |**

**+----------------------+ +------------------------+**

**When on-prem fails → Route 53 redirects traffic → AWS EC2 auto-scales to full capacity.**

**Answer Summary**

**Disaster Recovery in AWS is about preparing for failures and restoring workloads using AWS infrastructure and services.  
AWS offers four main DR strategies — Backup & Restore, Pilot Light, Warm Standby, and Multi-site (Active-Active) — depending on cost and recovery requirements.**

**226\* how to protect the aws through disaster**

**🧩 1. Understand Disaster Recovery (DR)**

**Disaster Recovery means having a plan and setup to recover your applications and data after a disaster — ensuring business continuity.**

**⚙️ 2. Key AWS Services for DR**

| **Purpose** | **AWS Service** | **Description** |
| --- | --- | --- |
| **Backup & Restore** | **AWS Backup / S3 / EBS Snapshots / RDS Snapshots** | **Automatically back up data regularly.** |
| **Multi-Region Replication** | **S3 Cross-Region Replication, RDS Read Replica, DynamoDB Global Tables** | **Keep copies of data in another region.** |
| **Infrastructure as Code** | **AWS CloudFormation / Terraform** | **Recreate your environment quickly.** |
| **Networking Redundancy** | **Route 53** | **DNS failover between regions.** |
| **Compute Failover** | **Auto Scaling / Elastic Load Balancer / Multi-AZ setup** | **Keep instances running even if one AZ fails.** |
| **Monitoring & Alerts** | **CloudWatch / CloudTrail** | **Detect and respond to issues early.** |

**🧠 3. DR Strategies on AWS**

| **Strategy** | **Description** | **Recovery Time (RTO)** | **Cost** |
| --- | --- | --- | --- |
| **Backup & Restore** | **Only data is backed up; restore after disaster** | **Hours–Days** | **💰 Low** |
| **Pilot Light** | **Minimal environment running (like DB), rest spun up on disaster** | **Minutes–Hours** | **💰💰 Medium** |
| **Warm Standby** | **Scaled-down full environment running; scale up during disaster** | **Minutes** | **💰💰💰 Medium–High** |
| **Multi-Site (Active-Active)** | **Fully replicated systems in two regions; traffic load balanced** | **Seconds** | **💰💰💰💰 High** |

**🛡️ 4. Example Setup**

**Let’s say you host an e-commerce app:**

* **Run primary workload in ap-south-1 (Mumbai).**
* **Enable S3 cross-region replication to us-east-1.**
* **Use RDS Multi-AZ for DB failover.**
* **Use Route 53 failover routing to switch traffic automatically.**
* **Keep EBS snapshots and AMIs for EC2 recovery.**

**✅ 5. Best Practices**

* **Store backups in another region or account.**
* **Regularly test DR drills.**
* **Automate recovery using CloudFormation.**
* **Maintain IAM least-privilege to avoid accidental deletions.**
* **Use AWS Organizations & Backup Policies for compliance.**

**227\* if you have diffrent aws account how you will do the deployment? multi region deployment?**

**🧩 Scenario**

**You have:**

* **Multiple AWS accounts (e.g., Dev, Test, Prod, or different teams).**
* **You also want multi-region deployment (e.g., us-east-1 + ap-south-1).**

**You need a deployment strategy that can handle both.**

**🚀 Step-by-Step Approach**

**1. Centralized CI/CD (using AWS CodePipeline or Jenkins)**

**Use a central CI/CD account or service to deploy to multiple target accounts and regions.**

**Tools:**

* **AWS CodePipeline + CodeBuild + CodeDeploy**
* **AWS CodeArtifact (for shared dependencies)**
* **Jenkins, GitHub Actions, or GitLab CI/CD**

**How it works:**

1. **Source code lives in one place (GitHub / CodeCommit).**
2. **CodePipeline in the main (e.g., tools) account builds the artifact.**
3. **Use cross-account IAM roles in target accounts:**
   * **Each target account has a deployment role.**
   * **The central pipeline assumes that role using STS:AssumeRole to deploy.**
4. **Deploy application stacks using:**
   * **AWS CloudFormation StackSets, or**
   * **Terraform Cloud / AWS CDK with cross-account roles.**

**2. Multi-Region Deployment**

**You can configure the same pipeline to deploy in multiple regions:**

**Example:**

* **Build once in us-east-1.**
* **Deploy the same artifact to:**
  + **us-east-1**
  + **ap-south-1**
  + **eu-west-1, etc.**

**You can do this using:**

* **Multiple deploy stages in CodePipeline.**
* **CloudFormation StackSets (supports multi-region).**
* **S3 artifact replication (to replicate build artifacts across regions).**

**3. Security and Access**

* **Create an IAM role in each target account with required permissions (e.g., CloudFormation deploy, ECS update, etc.).**
* **Allow the CI/CD account to assume that role using trust policies.**
* **Use AWS Organizations and Service Control Policies (SCPs) for governance.**

**4. Optionally Use AWS Organizations**

**For large-scale management:**

* **Use AWS Control Tower or AWS Organizations to manage multiple accounts.**
* **Apply consistent IAM roles and guardrails.**

**✅ Summary**

| **Requirement** | **Solution** |
| --- | --- |
| **Deploy across different AWS accounts** | **Use cross-account IAM roles + centralized CI/CD (CodePipeline/Jenkins)** |
| **Deploy across multiple regions** | **Use multi-region stages in the pipeline or CloudFormation StackSets** |
| **Manage consistency and security** | **Use AWS Organizations / Control Tower** |

**228\* how using route 53 route the traffoc to diffrent regions**

**🌍 Goal**

**You have users across the globe, and you want them to be directed to the nearest or most appropriate AWS region — for better performance, latency, or disaster recovery.**

**You can do this using Route 53 routing policies.**

**🔹 Route 53 Overview**

**Amazon Route 53 is a DNS (Domain Name System) service.  
It translates your domain name (like www.myshop.com) into an IP address of your web servers or load balancers.**

**You can configure different routing policies to decide *which endpoint* (region) should get the traffic.**

**🔸 Common Routing Policies for Multi-Region Routing**

| **Routing Policy** | **Use Case** | **How it routes traffic** |
| --- | --- | --- |
| **1. Latency-based routing** | **Send users to the region with lowest latency** | **Route 53 measures latency between AWS regions and user’s DNS resolver. Example: users in India → Mumbai region, users in US → N. Virginia region** |
| **2. Geolocation routing** | **Route based on user’s location (country/continent/state)** | **Example: users from Europe → eu-west-1 region, users from Asia → ap-south-1 region** |
| **3. Geoproximity routing *(requires Route 53 Traffic Flow)*** | **Route based on geographic proximity of AWS regions or resources and allows bias adjustment** | **Example: shift 20% more traffic toward Singapore region manually** |
| **4. Failover routing** | **For disaster recovery — route to primary region unless it fails** | **Primary → us-east-1, Secondary → us-west-2 (only used if health check fails)** |
| **5. Weighted routing** | **Distribute traffic across regions by percentage** | **Example: 70% to us-east-1, 30% to ap-south-1 (good for testing or migration)** |

**🔹 Example: Latency-based Multi-Region Setup**

**Scenario**

**You have two application stacks:**

* **us-east-1 → for North America users**
* **ap-south-1 (Mumbai) → for Asia users**

**Steps**

1. **Deploy your app in both regions (EC2, ALB, or CloudFront).**
2. **Each region has a Load Balancer DNS name, for example:**
   * **app-us-east-1.elb.amazonaws.com**
   * **app-ap-south-1.elb.amazonaws.com**
3. **In Route 53, create a hosted zone for your domain (e.g., myapp.com).**
4. **Create two records of type A or Alias:**
   * **Record 1 → Alias to us-east-1 ALB → Latency routing policy → Region: US East (N. Virginia)**
   * **Record 2 → Alias to ap-south-1 ALB → Latency routing policy → Region: Asia Pacific (Mumbai)**
5. **(Optional) Add health checks so that if one region fails, Route 53 routes all traffic to the healthy one.**

**How it works**

* **When a user in the US queries myapp.com, Route 53 routes them to the US (N. Virginia) ALB — lowest latency.**
* **When a user in India queries the same domain, Route 53 routes them to the Mumbai ALB.**
* **If Mumbai region fails, traffic automatically shifts to US region (if health check is configured).**

**🔹 Diagram**

**User (India) ──► Route 53 ──► ap-south-1 ALB (Mumbai)**

**User (USA) ──► Route 53 ──► us-east-1 ALB (Virginia)**

**│**

**Health checks ensure**

**automatic failover**

**🔹 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| **Latency-based routing** | **Best performance for users (lowest latency)** |
| **Geolocation routing** | **Route based on user's geographic origin** |
| **Weighted routing** | **Split traffic (for testing or gradual rollout)** |
| **Failover routing** | **High availability and disaster recovery** |

**✅ In short:**

**Using Route 53, you can route traffic to different AWS regions using latency, geolocation, geoproximity, weighted, or failover routing policies — ensuring global performance and resilience.**

**229\* how your created vpc , how many type of subnets u have in compnay?**

**✅ Answer:**

**In our company, I created a custom VPC using the AWS Management Console (or Terraform/CloudFormation depending on automation).**

**While creating the VPC, I specified:**

* **A CIDR block (for example 10.0.0.0/16)**
* **Then created subnets in multiple Availability Zones for high availability.**

**🏗️ Subnet Types in our Company**

**We generally have two main types of subnets:**

| **Type** | **Purpose** | **Accessibility** |
| --- | --- | --- |
| **Public Subnet** | **Used for resources that need internet access, such as ALB (Application Load Balancer), Bastion Host, or NAT Gateway** | **Connected to Internet Gateway (IGW)** |
| **Private Subnet** | **Used for internal resources such as Application servers, Databases (RDS), ECS tasks, or Jenkins master** | **Routes go through NAT Gateway for outbound internet only** |

**Sometimes companies also use a third type:**

| **Type** | **Purpose** |
| --- | --- |
| **Database Subnet (Isolated)** | **No internet access at all — used for RDS or data-sensitive systems** |

**🧩 VPC Setup Example**

* **VPC CIDR: 10.0.0.0/16**
* **Public Subnets: 10.0.1.0/24, 10.0.2.0/24**
* **Private Subnets: 10.0.3.0/24, 10.0.4.0/24**
* **Internet Gateway: attached to VPC**
* **NAT Gateway: in public subnet for private instances to reach the internet**
* **Route Tables:**
  + **Public → route to IGW**
  + **Private → route to NAT GW**

**🗣️ Short Interview Summary:**

**I created the VPC with a /16 CIDR block and divided it into multiple subnets — public, private, and database subnets across two Availability Zones. Public subnets host ALB/NAT, and private subnets host application and database servers for better security and isolation.**

**230\* how will acess the database from the public subnet**

**You have a database (like RDS) and you want to access it from a public subnet (for example, from an EC2 in a public subnet or your local system).**

**✅ Case 1: Access from EC2 in Public Subnet**

**If your DB is in a private subnet, and you want to connect to it via EC2 in a public subnet, follow these steps:**

**Step 1: Setup the VPC properly**

* **Your public subnet must have an Internet Gateway (IGW) attached.**
* **Your private subnet (where DB is) should not have direct Internet access.**

**Step 2: Launch EC2 in Public Subnet**

* **Launch a bastion/ jump EC2 instance in the public subnet.**
* **Assign a public IP or Elastic IP to it.**
* **Ensure it has SSH (port 22) open for your IP in the security group.**

**Step 3: Access the Database via Bastion Host**

* **Connect to the public EC2 via SSH:**
* **ssh -i mykey.pem ec2-user@<public-ec2-ip>**
* **From that EC2, connect to your RDS database:**
* **mysql -h <rds-endpoint> -u <dbuser> -p**

**This is the most secure and recommended way — you don’t expose the database publicly.**

**✅ Case 2: Access Database Directly (Publicly Accessible)**

**If you must access the DB directly from the internet (not recommended for production):**

**Step 1: In RDS → Modify DB instance**

* **Set Publicly Accessible = Yes.**

**Step 2: Ensure the RDS subnet is in a public subnet (i.e., has a route to Internet Gateway).**

**Step 3: Update RDS Security Group**

* **Allow inbound traffic on the DB port (e.g., 3306 for MySQL) from your public IP or a specific EC2 security group.**

**Step 4: Use the RDS endpoint**

**Connect from your system:**

**mysql -h <rds-endpoint> -u <dbuser> -p**

**⚠️ Security Best Practice**

* **Never keep a production DB public.**
* **Always use a bastion host (jump box) in a public subnet to access a private DB.**
* **Restrict access using security groups and NACLs.**
* **Use SSL/TLS for secure DB connections.**

**✅ Example Interview Answer:**

**“We never expose our database directly to the public.  
Our database is hosted in a private subnet.  
To access it, we use a bastion host or EC2 in a public subnet, connect via SSH, and from there we can access the database using its private endpoint.  
This approach keeps the database secure while still allowing controlled access.”**

**231\* what are the nacl rules u have allowed for your application?**

**1. Purpose of NACL**

**A Network Access Control List (NACL) is a stateless firewall at the subnet level in a VPC.  
It controls inbound and outbound traffic using allow/deny rules based on IP, protocol, and port range.**

**2. Typical NACL Rules for a Web Application**

| **Rule Type** | **Rule #** | **Protocol** | **Port Range** | **Source/Destination** | **Action** | **Description** |
| --- | --- | --- | --- | --- | --- | --- |
| **Inbound** | **100** | **TCP** | **80** | **0.0.0.0/0** | **ALLOW** | **Allow HTTP traffic** |
| **Inbound** | **110** | **TCP** | **443** | **0.0.0.0/0** | **ALLOW** | **Allow HTTPS traffic** |
| **Inbound** | **120** | **TCP** | **22** | **Your IP (e.g., 203.x.x.x/32)** | **ALLOW** | **Allow SSH from admin IP only** |
| **Inbound** | **130** | **TCP** | **1024–65535** | **0.0.0.0/0** | **ALLOW** | **Allow ephemeral ports for responses** |
| **Inbound** | **\*** | **All** | **All** | **All** | **DENY** | **Deny everything else by default** |

| **Rule Type** | **Rule #** | **Protocol** | **Port Range** | **Destination** | **Action** | **Description** |
| --- | --- | --- | --- | --- | --- | --- |
| **Outbound** | **100** | **TCP** | **80** | **0.0.0.0/0** | **ALLOW** | **Allow outbound web traffic** |
| **Outbound** | **110** | **TCP** | **443** | **0.0.0.0/0** | **ALLOW** | **Allow outbound HTTPS** |
| **Outbound** | **120** | **TCP** | **1024–65535** | **0.0.0.0/0** | **ALLOW** | **Allow return traffic** |
| **Outbound** | **\*** | **All** | **All** | **All** | **DENY** | **Deny all other traffic** |

**3. Key Points**

* **NACLs are stateless, so return traffic must be explicitly allowed.**
* **Lower rule numbers have higher priority.**
* **Default NACL allows all inbound/outbound traffic.**
* **Always restrict ports to only necessary ones (principle of least privilege).**

**4. Example Use Case**

**For a 3-tier web application:**

* **Public subnet (Web tier): Allow ports 80, 443, 22 inbound.**
* **Private subnet (App/DB tier):**
  + **App → DB: Allow port 3306 (MySQL) or 5432 (PostgreSQL).**
  + **Web → App: Allow port 8080 (if using that for internal communication).**

**Short Interview Answer Example:**

**“For my application, I configured NACL rules to allow inbound ports 80 and 443 for HTTP/HTTPS traffic, port 22 only from admin IP for SSH, and ephemeral ports (1024–65535) for return traffic. All other ports are denied by default. Outbound rules allow HTTP/HTTPS and ephemeral ports for response traffic.”**

**232\* what are the ports you have used for accesing data base for your application?**

**💡 Common Database Ports in AWS**

| **Database Type** | **Default Port** | **AWS Service Example** | **Notes** |
| --- | --- | --- | --- |
| **MySQL / MariaDB** | **3306** | **Amazon RDS for MySQL / Aurora MySQL** | **Most common for web apps (PHP, Java, Python)** |
| **PostgreSQL** | **5432** | **Amazon RDS for PostgreSQL / Aurora PostgreSQL** | **Used by Django, Spring Boot, etc.** |
| **Oracle** | **1521** | **Amazon RDS for Oracle** | **Enterprise DBs** |
| **Microsoft SQL Server** | **1433** | **Amazon RDS for SQL Server** | **Used for .NET apps** |
| **MongoDB** | **27017** | **Self-managed on EC2 / DocumentDB** | **NoSQL DB** |
| **Redis** | **6379** | **Amazon ElastiCache** | **In-memory cache** |
| **Cassandra** | **9042** | **Self-managed / Keyspaces** | **Columnar NoSQL** |
| **DynamoDB** | **—** | **Fully managed (no port, uses API)** | **Access via AWS SDK/API** |

**🛡️ Security Tip**

**When accessing your database:**

* **Open the DB port only to your application servers’ security group.**
* **Never open DB ports to 0.0.0.0/0 (public access).**
* **Use IAM authentication (where supported, e.g., RDS MySQL/PostgreSQL).**
* **Encrypt connections with SSL/TLS.**

**✅ Example Answer (Interview Style):**

**“For my application, we used Amazon RDS for MySQL, which listens on port 3306.  
The port is open only to the application EC2 instances’ security group — not to the public.  
We also used SSL for secure DB connections.”**

**233\* how access the database like mysql , how you will define the port for database? how you will do that**

**🧩 1. Accessing the Database (MySQL)**

**You can host MySQL on:**

* **Amazon RDS (MySQL engine)**
* **Amazon EC2 (self-managed MySQL)**

**You access it using a MySQL client or an application through:**

**mysql -h <endpoint> -u <username> -p**

**Where:**

* **-h → Hostname (RDS endpoint or EC2 private IP)**
* **-u → Username**
* **-p → Prompts for password**

**🔧 2. Defining the Port**

**By default, MySQL uses port 3306.**

**You can define or change the port in two ways depending on where you host it:**

**🟢 If using Amazon RDS (MySQL):**

* **When creating the DB instance:**
  + **Go to RDS Console → Create database → Additional configuration → Port**
  + **Default: 3306, but you can change it (e.g., 3307).**
* **Example:**
* **Endpoint: mydb.abcxyz.us-east-1.rds.amazonaws.com**
* **Port: 3306**
* **In the Security Group attached to RDS, open Inbound rule:**
* **Type: MySQL/Aurora**
* **Protocol: TCP**
* **Port range: 3306**
* **Source: App Server Security Group / IP**

**🟠 If using EC2 (self-hosted MySQL):**

* **Edit MySQL config file: /etc/mysql/mysql.conf.d/mysqld.cnf**
* **[mysqld]**
* **port = 3306**
* **bind-address = 0.0.0.0**
* **Restart MySQL:**
* **sudo systemctl restart mysql**
* **In EC2 Security Group, add an inbound rule:**
* **Type: MySQL/Aurora**
* **Protocol: TCP**
* **Port: 3306**
* **Source: <Private IP or Security Group of app servers>**

**🔐 3. Secure Access**

* **Never open port 3306 to 0.0.0.0/0 (public).  
  Use:**
  + **Private subnets for DB.**
  + **App servers in the same VPC/private subnet.**
  + **Security group rules allowing only specific sources.**

**✅ Summary**

| **Step** | **Action** | **Example** |
| --- | --- | --- |
| **1** | **Create MySQL (RDS or EC2)** | **RDS → MySQL engine** |
| **2** | **Define port** | **Default 3306 or custom** |
| **3** | **Allow in Security Group** | **Inbound rule TCP 3306** |
| **4** | **Connect** | **mysql -h endpoint -u admin -p** |
| **5** | **Secure** | **Use private subnets + SG access only from app tier** |

**234\* where will you define the port?**

**1️⃣ Application Level (inside your app or server)**

* **This is where your code or application framework listens on a port.**
* **Example:**
  + **In Node.js → app.listen(8080)**
  + **In Tomcat/JBoss → default HTTP port is 8080**
  + **In Nginx/Apache → defined in config files (listen 80;)**
* **⚙️ Purpose: Tells your app *which port* to listen for connections on inside the instance or container.**

**2️⃣ Operating System / Instance Level**

* **You must open the same port in your EC2 instance security group and OS firewall.**
* **Example:**
  + **Security group: inbound rule → TCP → port 8080 → source: ALB or 0.0.0.0/0 (for public)**
  + **OS firewall (iptables, firewalld): allow incoming connections on that port.**
* **⚙️ Purpose: Allows traffic to reach your application at that port.**

**3️⃣ Load Balancer Level (if using ALB / NLB / ELB)**

* **In AWS Application Load Balancer (ALB):**
  + **Listener port: defines what port ALB listens on (e.g., HTTP :80, HTTPS :443)**
  + **Target group port: defines what port the backend instance (target) listens on (e.g., :8080)**
* **Example:**
* **Listener (ALB) → Port 80**
* **↓**
* **Target Group → Port 8080**
* **↓**
* **EC2 App → Port 8080**
* **⚙️ Purpose: Maps incoming traffic to backend app port.**

**4️⃣ Container Level (if using ECS / Docker / Kubernetes)**

* **Container port: Port inside the container where app runs (e.g., 8080)**
* **Host port / Target port: Port on the host or service mapping**
* **Example (Docker):**
* **docker run -p 80:8080 myapp**

**→ Maps host port 80 to container port 8080**

* **⚙️ Purpose: Controls how external users or services reach containerized apps.**

**✅ Summary Table**

| **Layer** | **Where Port is Defined** | **Example** | **Purpose** |
| --- | --- | --- | --- |
| **Application** | **In code/config** | **app.listen(8080)** | **App listens on that port** |
| **EC2 / Firewall** | **Security Group / OS firewall** | **Inbound rule for 8080** | **Allow inbound traffic** |
| **Load Balancer** | **Listener & Target Group** | **Listener: 80 → Target: 8080** | **Route client to backend** |
| **Container** | **Dockerfile / ECS task** | **EXPOSE 8080, -p 80:8080** | **Map host to container port** |

**✅ In short:**

**You define the port at multiple layers — in your application, security group, and optionally in load balancer or container configuration, depending on your architecture.**

**235\* how you will protect your system so that no one can acess the system?**

**✅ Answer (Interview Style):**

**To protect my system so that only authorized users can access it, I implement multiple layers of security controls — at network, OS, and application levels.**

**🔒 1. Network-Level Security**

* **Use Security Groups & NACLs (Network ACLs):**
  + **Allow only required ports (e.g., 22 for SSH, 443 for HTTPS)**
  + **Deny all inbound traffic by default**
* **Restrict SSH access:**
  + **Only from my specific IP address or bastion host**
  + **Disable direct SSH to production servers**

**🧱 2. System-Level Security**

* **Strong authentication:**
  + **Use key pairs (SSH keys) instead of passwords**
  + **Disable root login over SSH (PermitRootLogin no)**
* **OS hardening:**
  + **Regularly update and patch the system**
  + **Remove unnecessary packages and services**
  + **Enable firewalls (ufw / iptables)**

**🔑 3. Identity and Access Management**

* **Grant least privilege access using IAM roles and policies**
* **Use MFA (Multi-Factor Authentication) for all users**
* **Rotate access keys and passwords regularly**
* **Disable or delete unused IAM users**

**🧰 4. Monitoring and Logging**

* **Enable CloudTrail and CloudWatch to track who accessed what**
* **Set up alarms for unauthorized access attempts**
* **Use GuardDuty or Security Hub for continuous threat detection**

**🧠 5. Encryption & Backup**

* **Encrypt data in-transit (HTTPS/SSH) and at-rest (KMS, EBS encryption)**
* **Maintain secure, regular backups with limited access**

**🗣️ Short Interview Summary:**

**I secure the system by restricting network access using security groups and firewalls, using key-based authentication, applying IAM least privilege, enabling MFA, patching regularly, and monitoring with CloudTrail and GuardDuty.**

**236\* how do ypu check how much memory is consumed once you configure the autoscaling ? where will be logs stored**

**🔹 1️⃣ How to check how much memory is consumed after Auto Scaling configuration**

**By default, CloudWatch automatically monitors:**

* **CPU Utilization**
* **Network In/Out**
* **Disk Read/Write**
* **Status Check**

**👉 But memory utilization is NOT monitored by default.  
You must configure it manually.**

**✅ Steps to Monitor Memory Usage in Auto Scaling EC2 Instances**

**Step 1: Install CloudWatch Agent**

**You need to install the CloudWatch Agent on your EC2 instances (the AMI used in the Auto Scaling group).**

**Example (Amazon Linux):**

**sudo yum install amazon-cloudwatch-agent -y**

**Step 2: Configure the Agent**

**Create or edit the agent configuration file /opt/aws/amazon-cloudwatch-agent/bin/config.json**

**Example snippet:**

**{**

**"metrics": {**

**"metrics\_collected": {**

**"mem": {**

**"measurement": ["mem\_used\_percent"],**

**"metrics\_collection\_interval": 60**

**}**

**}**

**}**

**}**

**Step 3: Start the Agent**

**sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \**

**-a start**

**Step 4: View Metrics in CloudWatch**

**Go to:  
AWS Console → CloudWatch → Metrics → All metrics → CWAgent → InstanceId → mem\_used\_percent**

**Here you can see memory utilization graphs per instance in your Auto Scaling group.**

**✅ For Auto Scaling Groups**

**You can:**

* **Aggregate metrics using CloudWatch Dashboards**
* **Set Alarms to trigger scaling policies, e.g.:**
  + **Scale out if average CPU > 70%**
  + **Scale in if average memory < 30%**

**🔹 2️⃣ Where will logs be stored?**

**Logs depend on what kind of logs you mean — here’s the breakdown:**

**🧩 a. Application Logs (from EC2 instances)**

**You can:**

* **Store locally on EC2 (e.g., /var/log/yourapp.log)**
* **Or send them to:**
  + **CloudWatch Logs (recommended)**
  + **S3 bucket (for archival)**

**To push logs to CloudWatch:**

**sudo vim /opt/aws/amazon-cloudwatch-agent/bin/config.json**

**Add this under logs section:**

**"logs": {**

**"logs\_collected": {**

**"files": {**

**"collect\_list": [**

**{**

**"file\_path": "/var/log/messages",**

**"log\_group\_name": "my-ec2-logs",**

**"log\_stream\_name": "{instance\_id}"**

**}**

**]**

**}**

**}**

**}**

**Then restart the agent — logs will appear in CloudWatch → Logs → Log Groups.**

**🧩 b. Auto Scaling Logs**

**Auto Scaling activities (like instance launch/terminate) are stored in:**

* **CloudWatch → Logs → Log Groups**
* **CloudWatch → Events / Alarms**
* **Auto Scaling → Activity History**

**Each activity entry shows:**

* **Time**
* **Status (e.g. "Launching a new EC2 instance")**
* **Cause (e.g. "Alarm triggered: CPU > 70%")**

**🧩 c. System Logs**

**Each EC2 also stores logs locally:**

**/var/log/messages**

**/var/log/cloud-init.log**

**/var/log/amazon-cloudwatch-agent.log**

**✅ Example Interview Answer**

**“After configuring Auto Scaling, I monitor memory usage by installing and configuring the CloudWatch agent on the EC2 AMI used by the Auto Scaling group.  
This sends memory metrics (like mem\_used\_percent) to CloudWatch where I can create dashboards or alarms.**

**Application and system logs are forwarded to CloudWatch Logs for centralized monitoring, while Auto Scaling activities can be viewed in the Auto Scaling console under ‘Activity History’ or in CloudWatch Events.”**

**237\* how ur acess the api gateway ? diffrence between load balancer and api gateway**

**✅ Answer:**

**You can access AWS API Gateway using its public endpoint URL or via a custom domain name.**

**When you deploy an API, API Gateway provides an endpoint like:**

**https://abcd1234.execute-api.ap-south-1.amazonaws.com/prod**

**You can then:**

* **Access it directly via browser or HTTP client (like Postman or curl).**
* **Integrate it with applications (mobile/web/IoT).**
* **Secure it using:**
  + **IAM authentication**
  + **Cognito user pools**
  + **API keys**
  + **Lambda authorizers (JWT or custom)**
* **Optionally, attach a custom domain name (e.g., api.myapp.com) through Route 53.**

**Example Access Flow**

**Client → HTTPS Request → API Gateway → Lambda / EC2 / Fargate / DynamoDB → Response → Client**

**Question 2: Difference between API Gateway and Load Balancer**

| **Feature** | **API Gateway** | **Load Balancer (ALB/NLB)** |
| --- | --- | --- |
| **Purpose** | **Manages and exposes APIs (REST, HTTP, WebSocket)** | **Distributes traffic across multiple targets (EC2, containers)** |
| **Layer (OSI)** | **Layer 7 (Application Layer)** | **ALB → Layer 7, NLB → Layer 4** |
| **Protocol Support** | **HTTP, HTTPS, WebSocket** | **ALB → HTTP/HTTPS; NLB → TCP/UDP** |
| **Authentication** | **Supports IAM, Cognito, JWT, API Keys** | **No built-in user auth; only security groups, certificates** |
| **Rate Limiting & Throttling** | **✅ Yes** | **❌ No** |
| **Caching** | **✅ Integrated with CloudFront** | **❌ Not supported** |
| **Integration Targets** | **Lambda, EC2, ECS, Step Functions, DynamoDB** | **EC2, ECS, Lambda (via target group)** |
| **Custom Domain** | **✅ Supported** | **✅ Supported** |
| **Monitoring** | **CloudWatch metrics for APIs** | **CloudWatch metrics for target health & requests** |
| **Use Case** | **Public API layer (REST/GraphQL)** | **Distribute load for web/app servers** |

**✅ In short:**

**API Gateway is for managing and securing API endpoints with fine-grained control (authentication, throttling, caching).  
Load Balancer is for distributing network traffic to backend servers for scalability and availability.**

**Example Answer (Interview Style):**

**“I access the API Gateway using its HTTPS endpoint or a custom domain name configured in Route 53. API Gateway acts as an entry point for my APIs and can connect to Lambda, EC2, or other services.  
The main difference is that API Gateway is used for managing APIs with features like authentication, throttling, and caching, while a Load Balancer is used for distributing traffic across multiple instances for scalability and fault tolerance.”**

**238\* how do we pass the credentials for api gateway? how do we pass for web browser?**

**How to pass credentials to API Gateway — and how a browser should send them**

**Good question — there are several secure ways to authenticate/authorize calls to Amazon API Gateway depending on your use case (server-to-server vs browser). Below I’ll summarize the common methods, when to use each, security best practices, and short examples you can copy.**

**1) Common authentication methods for API Gateway**

* **Cognito User Pools (JWT) — Browser-friendly. Clients sign in and receive JWT id\_token / access\_token. Send token in Authorization: Bearer <token>. Use API Gateway *Cognito authorizer* or a Lambda authorizer to validate.**
* **Lambda Authorizer (custom) — Your own logic (validate JWTs, check DB, call OIDC provider, etc.). Client passes credentials (usually in Authorization header) and Lambda returns allow/deny.**
* **AWS\_IAM (SigV4) — Uses AWS credentials (access key + secret) and request is signed with Signature Version 4. Good for server-to-server or browser when using temporary AWS credentials (from Cognito Identity Pools or STS).**
* **API Key + Usage Plan — Lightweight identification (not a secure auth mechanism by itself). Often used together with other auth; not recommended as sole security for sensitive APIs.**
* **OpenID Connect / OAuth 2.0 (OIDC) — Validate standard OAuth tokens (JWT) with API Gateway OIDC or custom authorizer.**
* **Mutual TLS (mTLS) — Certificate-based client auth (for high-security B2B APIs).**
* **Presigned URLs — For limited-time access to specific resources (S3 etc.) — token embedded in URL (avoid for broad API auth).**

**2) What to use from a web browser (recommended)**

* **Preferred: Use Cognito User Pools (or another OIDC provider). Browser performs user sign-in (hosted UI or SDK), receives a JWT, then includes it in requests:**
  + **Authorization: Bearer <access\_token> (or id\_token depending on your policy)**
  + **API Gateway validates the token with a Cognito authorizer or Lambda authorizer.**
* **If you need AWS\_IAM access from browser: Acquire temporary AWS credentials (via Cognito Identity Pool or STS) and sign requests with SigV4 (SDKs like AWS Amplify or AWS SDK handle signing).**
* **Do not embed long-lived AWS access keys in the browser.**

**3) Security best practices for browser apps**

* **Always use HTTPS.**
* **Do not store long-lived secrets in client code. Use short-lived tokens.**
* **Prefer storing tokens in memory or secure httpOnly cookies (if your architecture supports refresh via secure cookie). Avoid localStorage for sensitive tokens (XSS risk).**
* **Protect endpoints with proper scopes/claims and validate tokens server-side.**
* **Use CORS properly: allow only the required origins and methods.**
* **Avoid passing sensitive tokens in query strings (exposed in logs and referers).**

**4) Quick examples**

**A. Browser fetch with a JWT (Cognito / OIDC)**

**// After user signs in and you have an access\_token**

**const token = "eyJ..."; // access\_token (JWT)**

**fetch("https://api.example.com/v1/orders", {**

**method: "GET",**

**headers: {**

**"Authorization": `Bearer ${token}`,**

**"Content-Type": "application/json"**

**}**

**})**

**.then(res => res.json())**

**.then(data => console.log(data))**

**.catch(err => console.error(err));**

**API Gateway must be configured with a Cognito or JWT/Lambda authorizer to validate the token.**

**B. Server-to-server with SigV4 (curl example via AWS CLI is easier)**

**For servers, SigV4 is common. With AWS CLI you can call API Gateway endpoint protected by AWS\_IAM (CLI handles signing):**

**aws apigateway invoke-rest-api \**

**--rest-api-id <api-id> --resource-id <res-id> --http-method GET \**

**--stage-name prod \**

**--body '{}' output.json**

**Or use an SDK which signs requests automatically when you pass valid AWS credentials.**

**C. Using temporary AWS credentials in the browser (Cognito Identity Pool + AWS SDK)**

1. **User authenticates with Cognito User Pool (or social login) → receive id\_token.**
2. **Exchange id\_token with Cognito Identity Pool to get temporary AWS credentials (IAM role).**
3. **Use AWS SDK (or Amplify) to call AWS services or sign API requests.**

**Example with AWS Amplify:**

**import { Auth, API } from 'aws-amplify';**

**// After sign-in**

**const result = await API.get('MyApiName', '/items', {**

**headers: {} // Amplify uses AWS credentials to sign if API is configured for IAM**

**});**

**5) When to use what — short guide**

* **Public web app where users sign in → Cognito User Pool (JWT) + API Gateway Cognito authorizer.**
* **Browser needs to call AWS services directly (S3, etc.) → Use Cognito Identity Pool for temporary AWS credentials and SigV4 signing (via SDK).**
* **Server-to-server trusted calls → SigV4 or mutual TLS for high security.**
* **Simple throttling/identification → API Key with usage plan (not for sensitive auth).**

**6) Additional operational notes**

* **Use CloudWatch Logs to log authorization failures for debugging.**
* **Use WAF and rate-limiting for public endpoints.**
* **Rotate and expire tokens, and implement refresh-token flows securely.**
* **Test in different clients (curl, Postman, browser) to validate CORS and auth flows.**

**239\*** **how do you check the cloud watch logs through ec2 instance ? how do you check ?**

**🟢 Case 1: If EC2 is sending logs to CloudWatch**

**You can check logs from the EC2 instance itself using the CloudWatch Logs agent (or the newer CloudWatch Unified Agent).**

**1️⃣ Check if the CloudWatch agent is installed and running**

**Run on your EC2:**

**sudo systemctl status amazon-cloudwatch-agent**

**or for the old logs agent:**

**sudo systemctl status awslogs**

**If it’s active → logs are being sent to CloudWatch.**

**2️⃣ View or edit the CloudWatch agent config**

**Config file is usually at:**

**/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json**

**or for older agent:**

**/etc/awslogs/awslogs.conf**

**You can open it:**

**sudo cat /etc/awslogs/awslogs.conf**

**This shows which local log files are mapped to which CloudWatch log groups.**

**Example:**

**[/var/log/messages]**

**file = /var/log/messages**

**log\_group\_name = /ec2/system/logs**

**log\_stream\_name = {instance\_id}**

**3️⃣ To manually verify logs are reaching CloudWatch**

**You can use AWS CLI (from the EC2 instance itself):**

**aws logs describe-log-groups**

**and then:**

**aws logs get-log-events \**

**--log-group-name "/ec2/system/logs" \**

**--log-stream-name "<instance-id>"**

**This will fetch CloudWatch logs right from your EC2 terminal.**

**🔵 Case 2: If you mean checking EC2 logs locally (not in CloudWatch)**

**Then you check the log files directly on EC2:**

**cat /var/log/messages**

**cat /var/log/syslog**

**cat /var/log/cloud-init-output.log**

**cat /var/log/httpd/access\_log**

**But those are local logs, not CloudWatch logs.**

**✅ Summary**

| **Step** | **Purpose** | **Command** |
| --- | --- | --- |
| **1** | **Check agent status** | **sudo systemctl status awslogs** |
| **2** | **Check config file** | **cat /etc/awslogs/awslogs.conf** |
| **3** | **Verify logs using AWS CLI** | **aws logs get-log-events --log-group-name ...** |
| **4** | **Check locally if not using CloudWatch** | **cat /var/log/messages** |

**240\* how to check the logs of application in ec2?**

**🟢 Step-by-Step Answer**

**1️⃣ Connect to the EC2 instance**

**Use SSH to log in:**

**ssh -i my-key.pem ec2-user@<public-ip>**

**2️⃣ Go to the application log directory**

**Most applications store logs under one of these common paths:**

| **Application Type** | **Common Log Location** |
| --- | --- |
| **Web (Apache)** | **/var/log/httpd/ → access\_log, error\_log** |
| **Web (Nginx)** | **/var/log/nginx/ → access.log, error.log** |
| **Tomcat / JBoss / Java App** | **/opt/tomcat/logs/ or /var/log/jboss/** |
| **Node.js** | **Application’s custom log file (e.g., /home/ec2-user/app/logs/app.log)** |
| **System logs** | **/var/log/messages or /var/log/syslog** |
| **Cloud-init logs (startup issues)** | **/var/log/cloud-init-output.log** |

**3️⃣ View the logs using Linux commands**

**📄 To see the last few lines**

**tail -n 50 /var/log/httpd/error\_log**

**🔁 To watch logs live (real-time)**

**tail -f /var/log/httpd/access\_log**

**🔍 To search for errors**

**grep "ERROR" /opt/tomcat/logs/catalina.out**

**📅 To view logs with timestamp filter**

**less /opt/tomcat/logs/catalina.out**

**Use / inside less to search for keywords.**

**4️⃣ If your app writes custom logs**

**Example:  
Your app’s config (like application.properties or logging.conf) may define:**

**logging.file.path=/home/ec2-user/app/logs/**

**Then you can view:**

**tail -f /home/ec2-user/app/logs/app.log**

**5️⃣ (Optional) Send these logs to CloudWatch**

**If you want centralized monitoring:**

* **Install CloudWatch Agent.**
* **Configure /etc/awslogs/awslogs.conf to point to your app logs.**
* **Logs appear in CloudWatch Log Groups.**

**✅ Summary**

| **Task** | **Command / Action** |
| --- | --- |
| **Connect to EC2** | **ssh -i key.pem ec2-user@<ip>** |
| **Check app logs** | **cd /var/log/<appname>** |
| **View last lines** | **tail -n 50 <logfile>** |
| **Watch live logs** | **tail -f <logfile>** |
| **Search for errors** | **grep "ERROR" <logfile>** |
| **Send logs to CloudWatch** | **Configure CloudWatch Agent** |

**241\* if the logs size reaches the maximum capacity how ur going to manage, snapshot and ami are not the solution how you do it**

**Your log files (e.g., /var/log/, app logs, or web server logs) have grown so large that they’re consuming disk space on your instance.  
You can’t just create snapshots or AMIs — you need an operational log management strategy.**

**✅ Goal**

**Prevent log files from filling up the disk, keep recent logs accessible, and archive or delete old logs safely.**

**🔹 Step-by-Step Solutions**

**1️⃣ Log Rotation (Primary Solution)**

**Use logrotate — a Linux utility that automatically:**

* **Rotates (renames) old logs**
* **Compresses them (e.g., .gz)**
* **Deletes or archives older ones**
* **Keeps only recent logs (e.g., last 7 days)**

**Example config file → /etc/logrotate.conf or /etc/logrotate.d/app**

**/var/log/myapp/\*.log {**

**daily**

**rotate 7**

**compress**

**missingok**

**notifempty**

**create 0640 root root**

**postrotate**

**systemctl reload myapp**

**endscript**

**}**

**🟢 Meaning:**

* **Rotate logs daily**
* **Keep 7 days**
* **Compress old logs**
* **Delete older ones**
* **Reload app after rotation**

**2️⃣ Offload Logs to Centralized Storage**

**Instead of keeping logs locally, ship them to a central log management system.**

**Options:**

| **Destination** | **Service** | **Description** |
| --- | --- | --- |
| **AWS CloudWatch Logs** | **Use CloudWatch Agent** | **Streams logs in near real-time, view in console** |
| **S3 bucket** | **Use script or AWS CLI** | **Periodically upload old logs to S3** |
| **ELK Stack (Elasticsearch, Logstash, Kibana)** | **Self-managed or AWS OpenSearch** | **For searching and analyzing logs** |
| **Fluentd / Fluent Bit / Filebeat** | **Log forwarders** | **Stream logs to S3, CloudWatch, or OpenSearch** |

**🟢 Benefit: Instance disk stays light; logs are searchable and retained long-term elsewhere.**

**3️⃣ Monitor Disk Space**

**Set up CloudWatch alarms or local cron job to monitor /var/log usage.**

**Example:**

**df -h /var/log**

**du -sh /var/log/\***

**Or use a CloudWatch metric alarm:**

* **Trigger when disk space > 80%**
* **Send notification via SNS**

**4️⃣ Implement Lifecycle Policies**

**If you store logs in S3, configure:**

* **S3 Lifecycle Policy → Move old logs to Glacier / delete after X days.**
* **Keeps costs and storage under control.**

**5️⃣ Application-Level Logging Controls**

**In your app configuration:**

* **Reduce log verbosity (e.g., from DEBUG → INFO)**
* **Use structured logging to reduce redundant data**
* **Implement log rotation within the app (many frameworks support it)**

**6️⃣ Archiving Old Logs (Optional)**

**Periodically move old logs to archive storage:**

**tar -czf /tmp/app-logs-$(date +%F).tar.gz /var/log/myapp/**

**aws s3 cp /tmp/app-logs-$(date +%F).tar.gz s3://myapp-log-archive/**

**🔹 Summary Table**

| **Method** | **Purpose** | **Example** |
| --- | --- | --- |
| **Logrotate** | **Auto delete/compress old logs** | **/etc/logrotate.d/** |
| **CloudWatch / S3** | **Centralized long-term log storage** | **CloudWatch Agent / AWS CLI** |
| **Disk Monitoring** | **Detect high usage early** | **CloudWatch alarms** |
| **Lifecycle Policy** | **Manage retention in S3** | **Move to Glacier/Delete** |
| **App Logging Control** | **Reduce unnecessary logs** | **INFO level logging** |

**✅ In short:**

**You manage oversized logs by rotating, compressing, and offloading them — not by taking snapshots or AMIs.  
Tools like logrotate, CloudWatch Logs, and S3 are the best ways to handle log growth automatically.**

**242\* how do you mount the ebs volume permanently**

**✅ Step-by-Step Process**

**1. Attach the EBS Volume to the EC2 Instance**

* **In AWS Console → EC2 → Volumes**
* **Select the volume → Actions → Attach volume**
* **Choose your instance → note the device name (e.g., /dev/xvdf)**

**2. Connect to the Instance**

**ssh -i mykey.pem ec2-user@<public-ip>**

**3. Check if the volume is detected**

**lsblk**

**You’ll see something like /dev/xvdf (unmounted).**

**4. Create a filesystem on the volume (only once)**

**sudo mkfs -t ext4 /dev/xvdf**

**5. Create a mount point**

**sudo mkdir /data**

**6. Mount the volume temporarily**

**sudo mount /dev/xvdf /data**

**Now it’s mounted — but only until reboot.**

**7. Make it permanent (so it mounts automatically after reboot)**

**Edit the /etc/fstab file.**

**1️⃣ First, get the UUID of the volume:**

**sudo blkid /dev/xvdf**

**Output example:**

**UUID="a1b2c3d4-e5f6-7890-1234-56789abcdef0"**

**2️⃣ Open /etc/fstab:**

**sudo vi /etc/fstab**

**3️⃣ Add this line at the end:**

**UUID=a1b2c3d4-e5f6-7890-1234-56789abcdef0 /data ext4 defaults,nofail 0 2**

**8. Verify it works**

**Unmount and remount using fstab:**

**sudo umount /data**

**sudo mount -a**

**If no errors → it’s correct ✅**

**After reboot, the EBS will mount automatically.**

**🗣️ Short Interview Answer:**

**To mount an EBS volume permanently, I attach it to the EC2 instance, create a filesystem, mount it to a directory, and then add its UUID entry in /etc/fstab so it mounts automatically on every reboot.**

**243\* aws cloundFront**

**Amazon CloudFront is a Content Delivery Network (CDN) service that securely delivers your content (web pages, images, videos, APIs, etc.) to users with low latency and high transfer speed using a global network of edge locations.**

**🔹 How CloudFront Works**

1. **User requests content (e.g., image, video, or website page).**
2. **The request goes to the nearest CloudFront Edge Location.**
3. **If the content is cached there → it’s served immediately (low latency).**
4. **If not cached → CloudFront fetches it from the origin (like S3, EC2, or ALB), caches it at the edge, and serves it to the user.**
5. **Future users nearby get it directly from cache, improving speed and reducing load on your origin.**

**🔹 Key Components**

| **Component** | **Description** |
| --- | --- |
| **Origin** | **The source of your content (S3 bucket, ALB, EC2, or external server).** |
| **Distribution** | **The configuration that defines how CloudFront delivers your content.** |
| **Edge Locations** | **Global AWS data centers that cache your content closer to users.** |
| **Cache Behavior** | **Defines how CloudFront handles requests (methods, caching rules, redirects).** |
| **Origin Access Control (OAC)** | **Restricts direct access to S3; only CloudFront can fetch the files.** |
| **Signed URLs / Cookies** | **Used to restrict access to premium or private content.** |

**🔹 Typical Use Cases**

**✅ Static website acceleration (S3 + CloudFront)  
✅ Dynamic website or API acceleration (EC2 / ALB + CloudFront)  
✅ Video streaming (HLS/DASH)  
✅ Secure content delivery (HTTPS + signed URLs)  
✅ Global load balancing and failover**

**🔹 How to Configure CloudFront**

1. **Go to CloudFront console → Create Distribution**
2. **Choose Origin:**
   * **Example: mybucket.s3.amazonaws.com or your ALB DNS name.**
3. **Set Cache Behavior:**
   * **Redirect HTTP → HTTPS**
   * **Allow GET/HEAD (or POST if API)**
   * **Choose cache and origin request policy**
4. **Set Distribution Settings:**
   * **Alternate domain name (e.g., cdn.example.com)**
   * **Attach ACM SSL certificate**
   * **Choose Price Class (regions to serve from)**
5. **Create Distribution → Wait until status is *Deployed***
6. **Update DNS (Route 53):**
   * **CNAME cdn.example.com → <CloudFrontDomain>.cloudfront.net**

**🔹 CloudFront + S3 Example**

**If you host a static website in S3:**

* **S3 bucket (private) → origin**
* **Enable Origin Access Control (OAC)**
* **Block all public access to S3**
* **CloudFront securely fetches and caches content  
  ✅ Result: Secure, fast global delivery.**

**🔹 Monitoring & Logging**

* **Metrics: Available in CloudWatch (e.g., requests, cache hit rate, 4xx/5xx errors)**
* **Access Logs: Optional → stored in S3 bucket**
* **Invalidations: Manually remove old cached files (index.html, etc.)**

**🔹 Example Interview Answer**

**“CloudFront is AWS’s CDN service that distributes content globally through edge locations to reduce latency and improve performance.  
I configured CloudFront with an S3 origin to deliver static website content securely using HTTPS.  
The bucket is private and accessed only through Origin Access Control (OAC).  
We monitor performance using CloudWatch metrics and store access logs in S3 for audit and analysis.”**

**244\* diff between EKS and ECS**

**1️⃣ Definition**

| **Service** | **Full Form** | **Description** |
| --- | --- | --- |
| **ECS** | **Elastic Container Service** | **AWS’s proprietary container orchestration service — AWS manages everything; no Kubernetes needed.** |
| **EKS** | **Elastic Kubernetes Service** | **AWS’s managed Kubernetes service — runs standard Kubernetes control plane on AWS.** |

**2️⃣ Core Difference Overview**

| **Feature** | **ECS** | **EKS** |
| --- | --- | --- |
| **Orchestration Engine** | **AWS’s own engine** | **Kubernetes** |
| **Vendor Lock-in** | **✅ Yes (AWS-only)** | **❌ No (standard Kubernetes → portable to any cloud or on-prem)** |
| **Control Plane Management** | **Managed by AWS** | **AWS manages control plane (K8s API server, etcd)** |
| **Complexity** | **Simpler to use** | **More complex (K8s concepts: pods, deployments, etc.)** |
| **Pricing** | **No extra cost (pay only for EC2/Fargate)** | **Pay for EKS cluster ($0.10/hr) + worker nodes** |
| **Integration** | **Deeply integrated with AWS services (CloudWatch, IAM, ALB)** | **Integrates with AWS but also supports K8s-native tools (kubectl, Helm)** |
| **Compute Options** | **EC2 or Fargate** | **EC2 or Fargate** |
| **Use Case** | **Pure AWS workloads needing easy container management** | **Multi-cloud or K8s-based workloads needing portability and flexibility** |

**3️⃣ Example Analogy**

**ECS = AWS’s “own container manager” (simpler, AWS-native).  
EKS = Full Kubernetes cluster on AWS (open-source, more flexible, complex setup).**

**4️⃣ Example Use Case**

* **Use ECS if:**
  + **You want simplicity and full AWS integration.**
  + **You’re not using Kubernetes elsewhere.**
  + **You deploy microservices purely in AWS.**
* **Use EKS if:**
  + **You already use Kubernetes on-prem or multi-cloud.**
  + **You need custom K8s tools (Helm charts, operators, etc.).**
  + **You want workload portability.**

**5️⃣ Short Interview Answer Example**

**“ECS is AWS’s native container orchestration service, easy to manage and fully integrated with AWS.  
EKS is AWS’s managed Kubernetes service that gives you full Kubernetes capabilities and portability across environments.  
ECS is simpler; EKS is more flexible but complex.”**

**245\* s3 storage classes and diff bw standard and glacier storage class**

**🧱 Amazon S3 Storage Classes**

**S3 offers multiple storage classes — each optimized for different use cases, durability, and cost.**

| **Storage Class** | **Description / Use Case** | **Availability** | **Retrieval Time** |
| --- | --- | --- | --- |
| **S3 Standard** | **For frequently accessed data (e.g. websites, apps, content delivery).** | **99.99%** | **Immediate** |
| **S3 Standard-IA (Infrequent Access)** | **For infrequently accessed data but needs fast retrieval.** | **99.9%** | **Immediate** |
| **S3 One Zone-IA** | **Like Standard-IA but stored in one AZ only. Cheaper but less durable.** | **99.5%** | **Immediate** |
| **S3 Intelligent-Tiering** | **Automatically moves data between frequent and infrequent tiers based on access pattern.** | **99.9%** | **Immediate** |
| **S3 Glacier Instant Retrieval** | **For archival data that needs occasional, fast retrieval (milliseconds).** | **99.9%** | **Milliseconds** |
| **S3 Glacier Flexible Retrieval *(formerly Glacier)*** | **Low-cost archive storage; retrieval in minutes to hours.** | **99.9%** | **Minutes–hours** |
| **S3 Glacier Deep Archive** | **Lowest cost storage for long-term backup (rarely accessed).** | **99.9%** | **Hours (up to 12)** |

**⚖️ Difference between S3 Standard and S3 Glacier**

| **Feature** | **S3 Standard** | **S3 Glacier (Flexible Retrieval)** |
| --- | --- | --- |
| **Purpose** | **Frequently accessed data** | **Long-term archival storage** |
| **Cost** | **Higher (around $0.023/GB-month)** | **Much lower (around $0.004/GB-month)** |
| **Retrieval Time** | **Immediate (milliseconds)** | **Slow (minutes to hours)** |
| **Durability** | **99.999999999% (11 nines)** | **Same – 11 nines** |
| **Availability** | **99.99%** | **99.9%** |
| **Access Frequency** | **High** | **Very low** |
| **Storage Location** | **Multiple AZs** | **Multiple AZs** |
| **Use Cases** | **Active app data, websites, analytics** | **Backups, compliance archives, logs, old data** |

**💡 Example Use Cases**

* **S3 Standard: Product images, application data, active logs.**
* **S3 Glacier: Old transaction records, audit logs, compliance backups (retain for 7+ years).**

**🛠️ Tip**

**You can use Lifecycle Policies in S3 to automatically move data from:**

**Standard → Standard-IA → Glacier → Glacier Deep Archive  
as data ages and is accessed less frequently.**

**In short:**

* **S3 Standard = Hot storage (active data)**
* **S3 Glacier = Cold storage (archived data, low cost)**

**246\* types of ELB**

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

**Layer: 7 (Application Layer)**

**🔧 Used for:**

* **HTTP and HTTPS traffic**
* **Web applications and APIs**

**🌟 Key Features:**

* **Path-based routing (/api, /images)**
* **Host-based routing (app1.example.com, app2.example.com)**
* **Supports WebSocket and HTTP/2**
* **Can route to multiple target groups**
* **Integrates with AWS WAF**
* **Used with ECS (Fargate / EC2), EKS, or Lambda (via HTTP API)**

**✅ Example:**

**Route /api/\* to backend servers and /static/\* to another set.**

**🔹 2️⃣ Network Load Balancer (NLB)**

**Layer: 4 (Transport Layer)**

**🔧 Used for:**

* **High-performance, low-latency TCP/UDP traffic**
* **When you need to handle millions of requests per second**

**🌟 Key Features:**

* **Works with TCP, UDP, TLS**
* **Preserves client source IP**
* **Static IP support**
* **Ideal for microservices, gaming, or IoT traffic**

**✅ Example:**

**Load balance traffic for MySQL (port 3306) or custom TCP apps.**

**🔹 3️⃣ Gateway Load Balancer (GLB)**

**Layer: 3 (Network Layer)**

**🔧 Used for:**

* **Deploying and scaling network appliances such as:**
  + **Firewalls**
  + **Intrusion detection systems**
  + **Deep packet inspection**

**🌟 Key Features:**

* **Combines transparent network gateway + load balancer**
* **Uses GENEVE protocol (port 6081)**
* **Routes traffic to third-party virtual appliances**

**✅ Example:**

**Inspect traffic through a firewall cluster before reaching your application.**

**🔹 (Older Type) Classic Load Balancer (CLB)**

**⚠️ Legacy — for backward compatibility only.**

**🔧 Used for:**

* **Basic load balancing (HTTP, HTTPS, TCP, SSL)**
* **Operates on both Layer 4 and Layer 7**
* **Being replaced by ALB and NLB**

**✅ Summary Table**

| **Load Balancer Type** | **OSI Layer** | **Protocols** | **Key Use Case** |
| --- | --- | --- | --- |
| **ALB** | **Layer 7** | **HTTP, HTTPS** | **Web apps, APIs, path/host routing** |
| **NLB** | **Layer 4** | **TCP, UDP, TLS** | **High-performance network traffic** |
| **GLB** | **Layer 3** | **IP (GENEVE)** | **Security appliances, firewalls** |
| **CLB (Legacy)** | **Layer 4/7** | **HTTP, HTTPS, TCP, SSL** | **Old-generation load balancer** |

**247\* how will check when you are unable connect to ec2 instance**

**🔹 1. Check the Network Connectivity**

* **Ping test (for Linux instances)  
  → Use the public IP to check if the instance is reachable.**
* **ping <public-ip>**

**If it fails → possible Security Group / NACL / Routing issue.**

**🔹 2. Verify Security Group Rules**

* **Go to EC2 → Security Groups.**
* **Ensure:**
  + **Inbound rule allows SSH (port 22) for Linux or RDP (port 3389) for Windows.**
  + **Source should be your IP (or 0.0.0.0/0 for testing — not recommended for production).**
  + **Outbound rule allows traffic (generally All traffic).**

**🔹 3. Check Network ACLs (NACL)**

* **Go to VPC → Network ACLs.**
* **Ensure both Inbound and Outbound rules allow your traffic:**
  + **Inbound: Allow 22 (SSH) or 3389 (RDP).**
  + **Outbound: Allow ephemeral ports (1024–65535).**

**🔹 4. Check Route Table**

* **Go to VPC → Route Tables.**
* **For a public instance, make sure:**
  + **There’s a route like:  
    0.0.0.0/0 → igw-xxxxxxxx (Internet Gateway)**
* **For private instance, ensure it routes to a NAT Gateway or Bastion Host.**

**🔹 5. Check Elastic IP / Public IP**

* **Verify your instance has a public IPv4 address (if accessing over the internet).**
* **If using a Bastion Host, confirm its public IP is accessible and instance is in same VPC/subnet.**

**🔹 6. Check the Key Pair**

* **Confirm that:**
  + **You are using the correct .pem file (Linux) or .ppk (Windows with PuTTY).**
  + **Permissions on key file are correct:**
  + **chmod 400 my-key.pem**
  + **Connecting syntax:**
  + **ssh -i my-key.pem ec2-user@<public-ip>**

**🔹 7. Instance Status Checks**

* **Go to EC2 → Instances → Status Checks tab.**
  + **Ensure both checks are passed:**
    - **System status check**
    - **Instance status check  
      ❌ If failed → issue with AWS infrastructure or instance configuration.**

**🔹 8. Check OS-level Firewall**

* **If using iptables, firewalld, or Windows Firewall — verify port 22/3389 isn’t blocked inside the OS.**

**🔹 9. Use EC2 Serial Console / Session Manager**

**If you can’t connect even after all checks:**

* **Use EC2 Serial Console (for Linux) to troubleshoot network configs (like wrong SSH settings or network misconfig).**
* **Or AWS Systems Manager → Session Manager to connect without SSH.**

**🔹 10. Check IAM Role / Permissions (if SSM used)**

**If connecting via Session Manager:**

* **Ensure EC2 instance has an IAM role with policy:**
  + **AmazonSSMManagedInstanceCore**
* **Ensure SSM Agent is installed and running.**