**The Final Interview Devops Guide**

**Networking**

The OSI Model

Application Layer (Layer 7) - Interfaces directly with the end user applications and provides services for application layer protocols such as HTTP, HTTPS, SMTP.

Presentation Layer (Layer 6) - Translates data between the application and the network, handles data format translation, encryption and compression

Session Layer (Layer 5) - Manages sessions or connections between applications. establishing, maintaining and terminating sessions, controlling dialog between two devices.

Transport Layer (Layer 4) - Ensures reliable or fast delivery of data between applications. It segments data into packets and manages error and flow control.

A header is being added, the header contains the source/dest ports as well as sequencing number and acknowledgement number (TCP). creating segments and datagrams (UDP)

Network Layer (Layer 3) - Handles routing of data packets between devices across different networks. It uses logical addressing (IP) to determine the best path for data transmission. Routers operate at this layer, Directing packets to their destination based on IP. A header is created containing source/dest IP as well as TTL and packet size.

Data Link Layer (Layer 2) - Manages node to node data transfer and error detection/correction. It formats packets into frames. The frame’s header contains control information and source/dest MAC address to identify the devices.

Physical Layer (Layer 1) - Deals with the physical connection between devices. Including cables and switches. Also the transmission of raw bits.

| **Layer** | **Data Unit Name** | **Headers/Trailers** |
| --- | --- | --- |
| Transport (4) | Segment / Datagram | Source/Dest Ports, SYN-ACK |
| Network (3) | Packet | Source/Dest IPs, TTL, Protocol |
| Data Link (2) | Frame | MAC address, EtherType, FCS |
| Physical (1) | Bits | Transmitted as signals and bits |

Packets

Packets are small, manageable units of data that facilitate the transmission of larger data files across networks. By breaking data into packets, networks can efficiently route and deliver information, improving speed and reliability, each packet composed of those three:

1. Header: Critical for guiding packets to their destinations and contains essential metadata such as source/dest IP, a unique sequence number for order and protocol information.
2. Payload: The actual data that is being transmitted and its size.
3. Trailer: an optional port of the packet that often includes error checking such as a Frame Check Sequence (FCS) or checksum

Sockets

A socket is a software endpoint that enables communication between applications over the internet. It serves as an interface for sending and receiving data between devices. Each socket states its own IP and port and also what IP and port it listens to.

DNS

Computers, unlike humans communicate with each other by numbers such as IP Addresses, humans on the other hand are accustomed to use names. In order to break this communication gap, networking engineers developed DNS. DNS resolves domain names into IP Addresses. Here how it works:

1. OS to Resolver - If the local machine cannot find the IP in its own cached memory it will query the Resolver server hosted by your internet provider.
2. Resolver to Root DNS Server - If the resolver cannot find the IP in its own cached memory, it will send a query to the root DNS Server. The DNS Server is the tip of the DNS hierarchy. There are 13 of them strategically placed over the world owned by 12 organisations. The root server will direct the resolver to the Top Level Domain Server for its \*\*.\*\*.com/\*\*.\*\*.org & etc.
3. Resolver to the TLD Server - Each TLD Server manages different TLD names such as .org/.com . The TLD does not know the IP address and will direct the Resolver to the Authoritative Name Server
4. Resolver to authoritative Name Server - The NS responsible for knowing everything about the domain name including its IP address, The NS is the final authority, the NS will return the resolver with the IP address, the resolver will tell your local machine the IP address and will retrieve the web page.
5. Resolver Caching - The resolver will cache the IP so it won’t need to go through all these steps again

All DNS Records

**A**: Maps a domain to an IPv4 address.

**AAAA**: Maps a domain to an IPv6 address.

**CNAME**: Creates an alias for a domain.

**MX**: Specifies mail servers for a domain.

**NS**: Indicates authoritative name servers.

**PTR**: Maps an IP address to a domain name (reverse lookup).

**SOA**: Provides information about the DNS zone and authority.

**SRV**: Specifies services available on a domain.

**TXT**: Holds arbitrary text for verification and policy purposes.

**SPF**: Prevents email spoofing by specifying allowed mail servers.

**DKIM**: Allows for email signing to verify authenticity.

**CAA**: Authorizes which certificate authorities can issue SSL/TLS certificates for a domain.

**Virtual Private Network (VPN)**

is a secure connection method that allows users or devices to access a private network over a public network (like the internet). It creates an encrypted "tunnel" that ensures data traveling between the user's device and the private network is protected from interception.

Protocols

TCP

A connection-oriented protocol that ensures reliable data transmission Before data transfer, a connection is established using a three-way handshake (SYN, SYN-ACK, ACK):

SYN: initiates the connection and sets the client’s initial sequencing number.

SYN-ACK confirms the connection request and provides the server’s inital sequencing number by adding 1 to the client’s sequencing number

ACK: the client respond to the server with an ack packet that completes the setup

TCP also ensures packets arrive in order and handles retransmission if packets are lost.

* Segments: TCP splits large data into segments , each segment has the sequence number (SYN) and the acknowledgement numbers (ACK) as well as the actual data itself

UDP

Does not establish a connection between the sender and receiver and doesn't guarantee data delivery making it faster than tcp due to not having segments but rather having datagrams that don't contain any SYN or ACK numbers. UDP sacrifices reliability for speed. It is used by streaming services and gaming

HTTP

The foundation of data communication on the World Wide Web (www). It is used to transfer hypertext documents( like HTML pages) and etc. Each connection is serverless, and could be mitigated with cookies. Requests include methods like GET/POST/DELETE/PUT, paths, headers and body and responses include status code and requested data.

HTTPS

Similar to HTTP with an added layer of security. It establishes a source connection with SSL/TLS Handshake that authenticates the server and generates a session key for encryption.

SSH

A protocol used for secure remote administration and file transfer, providing a secure channel over an unsecured network. A connection made on port 22, a key exchange is made to generate a temporary session after that the server authenticates the client's private key with its public key.

ICMP

A lightweight protocol used to convey error messages and perform diagnostics, vital for managing and troubleshooting IP networks. It powers tools like ping and traceroute, helping network administrators assess connectivity and locate issues. However, ICMP itself carries no data and can be a target for misuse, requiring careful monitoring and filtering in secured environments. Layer 3

FTP/SFTP

FTP is a standard protocol for transferring files between a client and a server over a network. It is one of the earliest and simplest methods for file sharing but lacks modern security features. On port 20/21

SFTP is a secure version of FTP that uses SSH (Secure Shell) to encrypt data during file transfers. It ensures confidentiality, integrity, and authentication. On port 2SMTP

a protocol used to send emails between mail servers and from clients to mail servers for outgoing mail. It operates on port 25 (or 587/465 for secure communication) and ensures the delivery of messages to the recipient's server, which can then be accessed via protocols like IMAP or POP3.

### **DHCP (Dynamic Host Configuration Protocol)**

DHCP automates the assignment of IP addresses and network configuration to devices in a network. When a device joins a network, DHCP provides it with an IP address, subnet mask, gateway, and DNS server details.

* **How It Works:** The process involves four steps (DORA):
  + **Discover:** The device broadcasts a request for network configuration.
  + **Offer:** The DHCP server responds with an IP address offer.
  + **Request:** The device requests the offered IP address.
  + **Acknowledge:** The server confirms the assignment.
* **Use Case:** Simplifies network administration by dynamically managing IPs in large networks, avoiding manual configuration.

**Private IP Addresses**

Ranges of Private IPs:

* **Class A: 10.0.0.0 – 10.255.255.255 (large networks)**
* **Class B: 172.16.0.0 – 172.31.255.255 (medium-sized networks)**
* **Class C: 192.168.0.0 – 192.168.255.255 (small networks)**

**Networks Commnads**

1. Ping

* **Purpose**: Sends ICMP Echo Request packets to a host and waits for a reply. It measures round-trip time (latency) and reports packet loss.
* **Usage Example**: ping google.com

1. Traceroute (Linux) / Tracert (Windows):

* **Purpose:** Tracks the path data packets take from source to destination and measures the delay at each hop.
* **Example Use Case:** Troubleshooting slow connections by identifying bottlenecks in the route to a server. traceroute google.com

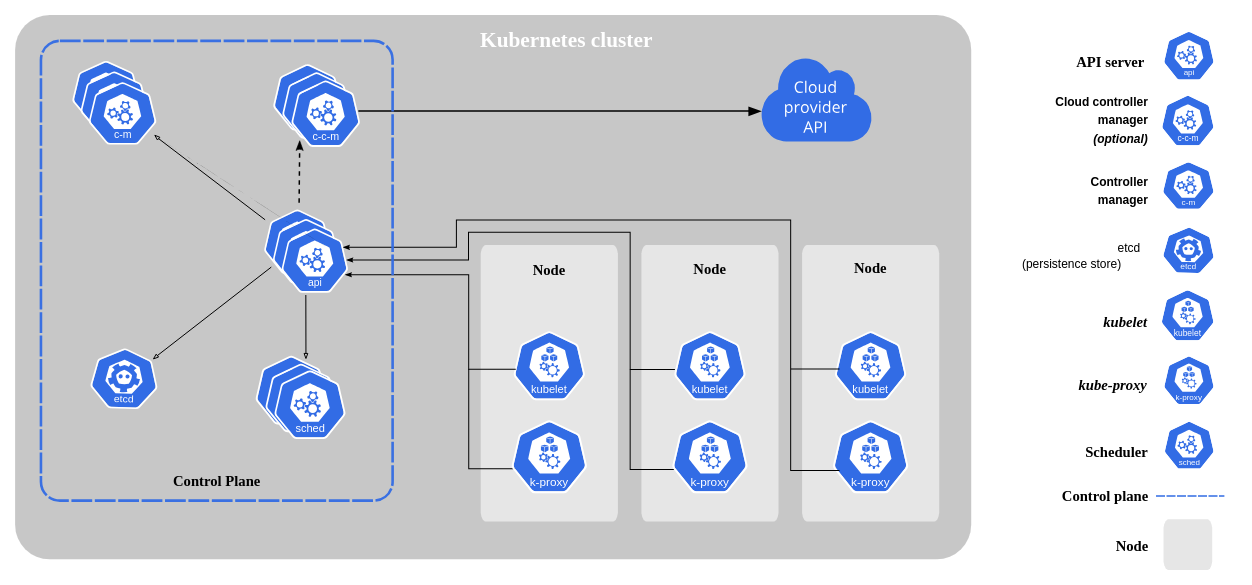
1. Netstat

* **Purpose:** Displays a wide range of network statistics, including open ports, active connections, and routing tables.
* **Example Use Case:** Use netstat -an to see all active connections and the ports being used or netstat -tunlp to find exactly which ports are already in use.

1. Tcpdump: Purpose: To capture and analyze network traffic. Usage Example: tcpdump -i eth0

**Kubernetes**

The Kubernetes architecture is built from two main nodes, the worker node and the control plane. Each node is built off different components and they are responsible for different things in the cluster..



### **Kubernetes Architecture**

#### **Worker Node Components**

1. **kubelet**
   * **Core Functionality:** Manages the lifecycle of containers and ensures they are running in their desired state.
   * **Responsibilities:**
     + Pod and container management, including health checks and reporting their status to the API server.
     + Node status and resource monitoring. Reports metrics like CPU and memory usage to the control plane.
     + Interacts with the container runtime to manage containers (e.g., pulling images, starting/stopping containers).
     + Invokes the CNI (Container Network Interface) plugin for pod networking, ensuring proper network configuration.
     + Handles storage and Persistent Volume (PV) assignments for pods.
2. **kube-proxy**
   * **Core Functionality:** Manages networking rules for services and enables communication between pods and services.
   * **Responsibilities:**
     + Configures networking rules (via iptables or ipvs) to distribute traffic among pods backing a service.
     + Supports service discovery by mapping service IPs to pod IPs.
     + Implements round-robin traffic distribution and optimizes for local traffic where possible.
     + Listens to service changes from the API server and dynamically updates rules accordingly.
3. **Container Runtime (CRI)**
   * **Core Functionality:** Responsible for running containers on the worker node. Examples include Docker, containerd, and CRI-O.
   * **Responsibilities:**
     + Pulls, starts, and stops container images.
     + Deletes containers when no longer needed.
     + Provides an abstraction layer between Kubernetes and underlying container technologies.

#### **Control Plane Components**

1. **API Server**
   * **Core Functionality:** Serves as the gateway for all interactions with the cluster. It is the single source of truth for the cluster’s state.
   * **Responsibilities:**
     + Exposes RESTful APIs for all cluster operations.
     + Authenticates and validates incoming requests.
     + Acts as the primary communication hub between internal components (e.g., scheduler, controller manager) and worker nodes.
2. **etcd**
   * **Core Functionality:** A highly available, distributed key-value store that stores all cluster data.
   * **Responsibilities:**
     + Stores the desired state of the cluster (e.g., pods, deployments, services).
     + Persists operational data, ensuring fault tolerance and recovery.
     + Acts as the data backend for the API server.
     + Ensures consistency across all cluster components through leader elections and consensus mechanisms.
3. **Controller Manager**
   * **Core Functionality:** Ensures the cluster’s actual state aligns with the desired state defined in the configuration.
   * **Responsibilities:**
     + Manages controllers for core Kubernetes objects, such as ReplicaSets, Endpoints, and Nodes.
     + Monitors the cluster for discrepancies and reconciles changes.
     + Includes various specialized controllers, such as:
       - **Node Controller:** Monitors node health.
       - **ReplicaSet Controller:** Ensures the desired number of pod replicas are running.
       - **Endpoint Controller:** Manages endpoints for services.
4. **Scheduler**
   * **Core Functionality:** Determines the optimal worker node for running a new pod.
   * **Responsibilities:**
     + Evaluates worker nodes based on resource requirements (CPU, memory, etc.), taints, tolerations, and affinities.
     + Prioritizes nodes to optimize workload distribution.
     + Assigns pods to nodes by updating their status in the API server.

### **Step-by-Step Process of Creating a New ReplicaSet in the Control Plane**

1. **User Submits the ReplicaSet Definition:**
   * A user creates and submits a **ReplicaSet** definition (usually in YAML or JSON format) to the Kubernetes API server, typically using kubectl apply -f replicaset.yaml.
   * This definition specifies the desired state, such as the number of replicas, the pod template (with container specifications), labels, and selectors.
2. **API Server Receives the Request:**
   * The **API server** receives the ReplicaSet definition and validates the provided configuration.
   * It checks the YAML or JSON for correctness (e.g., syntax, required fields) and ensures that the configuration adheres to Kubernetes API schema.
3. **API Server Updates etcd:**
   * After validation, the **API server** stores the **desired state** of the ReplicaSet in **etcd** (the cluster's key-value store).
   * **etcd** holds the current state of the cluster, and this state will be continually updated as the cluster operates. The desired state (as defined by the user) is now persisted in **etcd** via the API server.
4. **Controller Manager Detects the New ReplicaSet:**
   * The **Controller Manager** (specifically, the ReplicaSet Controller) monitors the state of the API server for changes in ReplicaSet objects.
   * It periodically checks the API server for new or updated ReplicaSets and compares the desired state with the actual state. If the desired state is different from the actual state, the controller takes corrective actions to reconcile them.
   * In this case, the ReplicaSet controller sees that a new ReplicaSet has been created and checks if the specified number of replicas is already running.
5. **ReplicaSet Controller Acts:**
   * The **ReplicaSet Controller** sees that the desired number of pods (replicas) is specified but that they aren't yet created or running.
   * It then instructs the **Scheduler** to place the required number of pods onto appropriate worker nodes, considering resource availability, node affinity, and taints/tolerations (if specified in the ReplicaSet definition).
6. **Scheduler Selects a Worker Node:**
   * The **Scheduler** receives the request from the ReplicaSet controller to schedule the pods.
   * It selects appropriate worker nodes based on the pod’s resource requests (CPU, memory), node affinity, and other scheduling policies.
   * The Scheduler updates the **API server** with the status of which node the pod will be scheduled on, which is also stored in **etcd**.
7. **Controller Manager Reconciles the State:**
   * Once the Scheduler assigns the pods to nodes, the **Controller Manager** continues to monitor the ReplicaSet’s desired state.
   * It ensures that the correct number of replicas is maintained. If any pod fails or is terminated, the Controller Manager creates a new pod to maintain the desired count of replicas.

8. **Kubelet's Role in Pod Initialization and Container Setup**

1. Retrieve Pod Specification: The kubelet continuously communicates with the API server to fetch the pod specification, which includes details like container images, environment variables, resource limits, and volumes.
2. Prepare the Node: It interacts with the container runtime (e.g., Docker, containerd) to pull the specified container images from a registry (if not already cached) and prepares them for execution.
3. Create Containers: The kubelet instructs the container runtime to create containers based on the pod specification, allocating the necessary resources (CPU, memory, etc.) for them to run.
4. Set Up Networking: The kubelet triggers the CNI plugin to configure networking, assigning an IP address to the pod and setting up the necessary routes for communication with other pods and services in the cluster.

**9. Kube-proxy Configures Service Communication:**

* The kube-proxy on the worker node is responsible for handling networking and load balancing between pods and services.
* It ensures that traffic directed to a service is correctly routed to the right pod(s) by setting up iptables rules or IPVS (depending on the mode).
* The kube-proxy monitors the state of the pod and updates routing rules dynamically as pods come online or go offline, ensuring efficient routing for service discovery.

**Kubernetes Objects**

**Deployment -**  in Kubernetes manages the rollout and scaling of applications, ensuring that a specified number of pod replicas are running and up-to-date. It supports rolling updates, allowing you to deploy new versions of applications without downtime, by gradually replacing old pods with new ones. However, mounting the same PV to many pods won’t be possible and even recreating new PVs for each pod won’t be possible. We will use statefulset for that

**Statefulset -** Differently from deployment, in statefulsets not all pods are identical, they are different from each other by the next properties:

1. For database replicas not all replicas can both execute read and write queries to the database, instead only one can do both (master) while the rest can only read (slaves).
2. Each pod has its own replica of the storage.
3. If a new pod enters the statefulset it will first replicate the data from the nearest pod and then sync with the master.
4. Each pod has its own state. It is all stored in the PV. When a pod die and get replaced the state from the PV will be
5. Each pod get its own name

To perceive an app’s state we will use statefulsets. It will have a persistentVolumeClaim that will automatically create a new PV directed by the Storage Class. It provides each pod a unique and persistent ID. The data is first replicated from the closet pod’s PV and after that it will synchronize with the master pod. The pods will be created in the following order pod-0, pod-1 and deleted by the opposite order.

**Daemonset -**written very similarly to a replicaset, used to create a node mandatory cluster wide deployments. It ensures an automatic pod management and deployment. Use cases will be monitoring and logging agents such as fluentd and prometheus exporter and network services

**Job** - process that will run on a pod but will be executed just once. Allows parallelism for pods to run and also to set a number of pods to run successfully before a job is done

**CronJob** - Similar to a job, but runs whenever it is scheduled.

**ResourceQuata** - limits the amount of resources per namespace. It can contain a max amount of pods or a guaranteed amount of CPU and RAM and also a limit

**HorizontalAutoscaling** - Automatically scales the number of pods in a deployment based on CPU, RAM and other metrics. Has a minimum and maximum amount of pods it can scale to.

**Service -** enables communication between pods and the exposure of applications within and outside the cluster. Kubernetes provides **automatic DNS-based service discovery**, assigning each service a unique DNS name (e.g., <service-name>.<namespace>.svc.cluster.local). This DNS name allows pods to connect without relying on hardcoded IP addresses, ensuring flexibility and resilience. Services manage **endpoints**, representing the IPs of the pods they route traffic to, and dynamically update these as pods are added or removed, using selectors (based on pod labels) to determine pod membership.

There are several **types of services** in Kubernetes:

* **ClusterIP**: The default type, exposing the service on an internal cluster IP accessible only from within the cluster.
* **NodePort**: Exposes the service on each node’s IP at a static port, enabling external traffic access through the node’s IP and specified port.
* **LoadBalancer**: Integrates with cloud providers to create an external load balancer that routes traffic to the service, providing a single externally accessible IP address.

**Headless Services**: A specialized type of service that doesn't assign a cluster IP. Instead, it directly provides the pod IPs to clients, often used with stateful applications requiring direct access to individual pod instances. This is achieved by setting clusterIP: None in the service specification.

**Ingress** - a Kubernetes resource that defines rules for routing external HTTP(S) traffic to services within a cluster. It acts as a bridge between external clients and internal services, enabling access based on specific routing rules such as:

* **Host-based routing**: Directs traffic to different services based on the requested hostname (e.g., app.example.com vs. api.example.com).
* **Path-based routing**: Routes traffic to services based on the request URL path (e.g., /app to one service and /api to another).

Ingress can also handle:

* **SSL/TLS termination**: Offloads HTTPS encryption from backend services, simplifying their configuration.
* **Load balancing**: Balances incoming traffic across multiple pods behind a service, ensuring high availability and efficient resource utilization.
* **Custom headers or annotations**: Facilitates advanced configurations like rate limiting, authentication, and more.

**Ingress Controller -**An ingress resource is simply a set of rules; it doesn't work alone. An **Ingress Controller** is required to interpret those rules and manage external traffic accordingly.

* The ingress controller is a **pod running within the cluster** that watches for changes to ingress resources.
* It programs an underlying HTTP(S) load balancer or proxy (like **NGINX**, **HAProxy**, or **Traefik**) to enforce the rules defined in the ingress resource.
* The controller ensures that traffic from external clients is directed to the appropriate service based on the ingress configuration.

**PersistentVolume (PV)** - A storage resource in the cluster that exists independently of pods. It is a physical storage resource, such as an NFS share, a disk on a cloud provider, or a local disk.

**PersistentVolumeClaim (PVC)** - A request for storage made by a pod. PVCs specify the storage requirements (size, access modes, etc.) and bind to a PV that meets those requirements.

**StorageClass (SC)** - Defines the dynamic provisioning of PVs. It specifies the storage backend (e.g., AWS EBS, GCP PD) and parameters like performance or replication settings. When a PVC requests storage and specifies an SC, Kubernetes automatically provisions a PV based on the SC configuration.

**Pod Configuration and Lifecycle Management**

**Node Affinity -**  Node affinity allows you to control where pods are scheduled based on the node’s labels. It's a flexible way to specify which nodes are suitable for your pods. There are two types:

* requiredDuringSchedulingIgnoredDuringExecution: The pod will only run on nodes that match the node affinity rule.
* preferredDuringSchedulingIgnoredDuringExecution: The scheduler will prefer nodes with certain labels, but it can still schedule the pod on nodes that don't match if necessary.

**Taints and Tolerations** - Taints are applied to nodes to repel certain pods from being scheduled on them. Tolerations are applied to pods to allow them to be scheduled on tainted nodes. For example, a node might be tainted with a "NoSchedule" taint, meaning no pod will be scheduled on it unless the pod has a toleration for that taint.

**LivenessProbe** - A crucial feature to ensure a pod is alive and responsive. If a container goes into a hung or unresponsive state, the livenessProbe will detect it and restart the pod

**ReadinessProbe** - Determine if a container is prepared to handle traffic. This probe helps to avoid sending requests to pods/containers that are still initializing or temporarily unavailable

**Resource requests and limits** - Define both the minimum resources a container needs and also the maximum it can use (CPU, RAM etc. )

**Prometheus**

### **Why Prometheus is Essential**

* **Modern Infrastructure Complexity:** Managing distributed servers, containerized applications, and interconnected services is challenging. Prometheus helps ensure smooth operation and reduces downtime.
* **Problem Diagnosis:** Prometheus provides insights into the entire infrastructure, enabling quick identification of issues such as memory overloads, disk space exhaustion, or network traffic spikes.
* **Proactive Monitoring:** It alerts maintainers of potential issues before they escalate, such as memory nearing capacity or storage limits in Elasticsearch.

### **Prometheus Architecture**

* **Core Components:**
  + **Time-Series Database:** Stores metrics like CPU usage or application errors.
  + **Data Retrieval Worker:** Pulls metrics from targets and stores them.
  + **Web Server/API:** Allows querying metrics and integrates with visualization tools like Grafana.
* **Targets and Metrics:** Prometheus monitors specific resources (targets) such as servers, databases, or applications. Metrics are units of monitoring, like CPU usage or request rates.

### **Metrics Collection**

* Prometheus uses an HTTP **pull mechanism** to gather metrics from endpoints (e.g., /metrics). Targets must expose this endpoint in a Prometheus-compatible format.
* **Exporters**: For services without native Prometheus endpoints, exporters convert data into a format Prometheus understands. Examples include:
  + **Node Exporter:** For Linux server metrics.
  + **MySQL Exporter:** For monitoring MySQL databases.
* **Client Libraries:** Developers can use Prometheus libraries (e.g., for Java or Node.js) to expose custom application metrics.

### **Advantages of the Pull Mechanism**

Unlike push-based systems like CloudWatch or New Relic, Prometheus's pull-based approach reduces network traffic and avoids creating bottlenecks in large-scale microservice architectures.

### **Pull Mechanism in Prometheus**

Prometheus uses a **pull-based approach** to collect metrics. This means that instead of services pushing data to Prometheus, Prometheus actively queries specific endpoints on monitored targets to fetch metrics.

* **Endpoint Requirement:** Each target (e.g., server, application) must expose metrics at an HTTP endpoint, typically /metrics.
* **Advantages of Pull Mechanism:**
  + **Reduced Overhead:** Prometheus decides when and how often to scrape metrics, optimizing resource usage.
  + **Isolation:** If one target fails, Prometheus can continue scraping other targets without interruption.
  + **Customizable Scrape Intervals:** Different scrape frequencies can be set per target, allowing flexibility in monitoring critical vs. non-critical components.

### **Key Use Cases**

* **Preventing Resource Exhaustion:** Monitors memory, storage, and network usage to alert maintainers before critical thresholds are reached.
* **Debugging Chain Failures:** Tracks cascading issues across interconnected services, such as database failures leading to authentication errors.
* **Custom Metrics Monitoring:** Allows teams to collect metrics tailored to their applications, enhancing visibility and control.

Prometheus’s design for scalability, extensibility, and proactive monitoring makes it a vital tool in modern DevOps workflows. Its integration with exporters and visualization tools further extends its utility across diverse infrastructure setups.

**AWS**

**EC2**

### **EC2 Launch Types**

1. **On-Demand Instances**:
   * For short, unpredictable workloads.
   * No upfront payment, highest cost, no long-term commitment.
2. **Reserved Instances**:
   * For long-term, predictable workloads.
   * Discounts up to 72% based on reservation period (1 or 3 years) and payment options (No Upfront, Partial Upfront, All Upfront).
   * Can have regional or zonal scope.
3. **Convertible Reserved Instances**:
   * For long workloads requiring flexibility in instance attributes (type, family, OS, etc.).
   * Discounts up to 66%.
4. **Savings Plans**:
   * Commitment to a fixed usage amount ($/hour) for 1 or 3 years.
   * Flexible across size, OS, and tenancy but locked to a family and region.
   * Discounts up to 72%.
5. **Spot Instances**:
   * For short-term, cost-sensitive workloads.
   * Up to 90% discount but instances can be interrupted.
   * Ideal for fault-tolerant tasks like batch jobs or data analysis.
6. **Dedicated Instances**:
   * No hardware sharing with other customers.
   * Useful for compliance needs but shares hardware within the same account.
7. **Dedicated Hosts**:
   * Full control of physical servers.
   * Suitable for licensing needs or strict compliance requirements.
8. **Capacity Reservations**:
   * Reserve capacity in a specific AZ for a duration.
   * Charged at On-Demand rates even if not used; no discounts unless combined with Reserved Instances or Savings Plans.

**Placement Groups**

Placement Groups allow you to define how EC2 instances are placed on AWS infrastructure to optimize performance, fault tolerance, or scalability. There are **three strategies**:

### **1. Cluster Placement Group**

* **Setup**: Instances are grouped together within a single **Availability Zone (AZ)**.
* **Features**:
  + **High performance**: Low latency, high throughput, and up to **10 Gbps bandwidth** between instances (with enhanced networking).
  + **Risk**: A failure in the AZ impacts all instances in the group.
* **Use Cases**:
  + **Big data processing** that needs to finish quickly.
  + Applications requiring **extremely low latency** and **high network throughput**.

### **2. Spread Placement Group**

* **Setup**: Instances are placed on **different hardware** across multiple AZs (one instance per host).
* **Features**:
  + Reduces failure risk by isolating instances.
  + Limited to **7 instances per AZ per placement group**.
* **Use Cases**:
  + **Critical applications** where instance failures must not affect one another.
  + Scenarios requiring **high availability** and **fault isolation**.

### **3. Partition Placement Group**

* **Setup**: Instances are spread across multiple **partitions** (groups of hardware racks) within an AZ or across multiple AZs.
* **Features**:
  + **Partitions** isolate failures; instances in one partition do not share the same rack as another partition.
  + Supports **up to 7 partitions per AZ** and **hundreds of instances**.
  + Partition information is available via the **AWS metadata service**.
* **Use Cases**:
  + **Big data applications** that can be partition-aware (e.g., **Hadoop, Cassandra, Kafka**).
  + Workloads requiring massive scalability with fault isolation.

**EC2 Hibernate**

EC2 Hibernate preserves the **in-memory state (RAM)** of an instance, enabling faster restarts without reinitializing applications or warming caches. During hibernation, the RAM is dumped to the **root EBS volume**, which must be **encrypted** and large enough to store the RAM contents.

**EC2 Instance Storage**

**EBS**

EBS (Elastic Block Store) is a network-attached storage service for EC2 instances that provides persistent, independent data storage. Acting like a network USB drive, EBS volumes can be detached and reattached to instances within the same AZ. They are AZ-specific, support snapshots for cross-AZ or cross-region replication, and allow multiple volumes per instance. With configurable performance, capacity, and the "Delete on Termination" attribute, EBS is ideal for scenarios requiring flexible, high-performance, and durable storage.

**EBS Snapshots**

EBS Snapshots are backups of your EBS volumes at a specific point in time. These can be created without detaching the volume from an EC2 instance, though detaching is recommended. Snapshots can be copied across Availability Zones (AZs) or Regions.

* **Snapshot Archive**: Moving snapshots to an archive tier reduces costs by up to 75%, but restores take 24 to 72 hours.
* **Recycle Bin**: Deleted snapshots are placed in a Recycle Bin, allowing recovery within a retention period (1 day to 1 year) to prevent accidental deletion.
* **Fast Snapshot Restore**: Speeds up the initialization of large snapshots for quick volume use but comes at a higher cost.

**EC2 Instance Store**

EC2 Instance Store is a high-performance storage option attached directly to physical servers hosting EC2 Instances. It provides better I/O performance and throughput compared to network-attached storage like EBS, making it ideal for workloads requiring high disk performance.

* **Ephemeral Storage**: Data is lost if the EC2 Instance is stopped or terminated.
* **Use Case**: Best for temporary data like caches, buffers, or scratch data—not suitable for long-term storage.
* **Risk of Data Loss**: If the underlying physical server fails, data loss occurs. Responsibility for backup and replication lies with the user.
* **Performance**: EC2 Instance Store can offer IOPS up to 3.3 million, far exceeding EBS performance (e.g., 32,000 IOPS for a BP2 EBS volume).

**EBS Volume Type**

EBS volumes come in six types, each suited for different workloads. These can be broadly grouped into SSD and HDD categories, offering various price points, performance characteristics, and use cases.

### **SSD Volumes:**

* **gp2 (General Purpose SSD)**: Balanced cost and performance for a wide range of workloads. Can be used for boot volumes, virtual desktops, and development environments.
* **gp3**: The newer version of gp2, providing up to 16,000 IOPS and 1,000 MB/s throughput. IOPS and throughput are independent, unlike gp2, where they are linked.
* **io1 & io2 (Provisioned IOPS SSD)**: Designed for mission-critical applications requiring consistent IOPS, like databases. io1 supports up to 64,000 IOPS, while io2 Block Express can provide up to 256,000 IOPS with sub-millisecond latency.

### **HDD Volumes:**

* **st1 (Throughput Optimized HDD)**: Low-cost volume for frequently accessed, throughput-intensive workloads like big data and log processing, supporting up to 500 MB/s throughput and 500 IOPS.
* **sc1 (Cold HDD)**: The lowest-cost volume for infrequently accessed data, offering up to 250 MB/s throughput and 250 IOPS.

**EFS**

Amazon **Elastic File System (EFS)** is a managed network file system that provides scalable, high-availability storage for EC2 instances. It can be mounted on instances in different Availability Zones (AZs) and is ideal for workloads that require shared access across multiple EC2 instances. It is pay-per-use and eliminates the need for capacity planning.

### **Key Features:**

* **Scalable and Managed**: Automatically scales to petabyte levels and provides high availability.
* **Linux-Only Compatibility**: EFS supports only Linux-based AMIs, not Windows.
* **Encryption**: Encryption at rest is supported through AWS KMS.
* **Flexible Access**: Can be mounted by multiple EC2 instances across different AZs for shared access.

### **Performance:**

* **General Purpose Mode**: Default mode suitable for latency-sensitive workloads like web servers and content management systems.
* **Max I/O Mode**: Ideal for applications needing high throughput and parallel processing, such as big data and media processing.

### **Throughput Modes:**

* **Bursting**: Throughput increases with storage size, suitable for variable workloads.
* **Provisioned**: Allows you to set throughput independently of storage size.
* **Elastic**: Automatically adjusts throughput based on workload needs, ideal for unpredictable workloads.

### **Storage Classes:**

* **Standard**: For frequently accessed files.
* **EFS-IA (Infrequent Access)**: Lower cost for infrequently accessed data, with retrieval charges.
* **Archive**: Cheapest storage option for rarely accessed data.
* **Lifecycle Management**: Moves data automatically between tiers based on access patterns, optimizing costs.

**High Availability and Scalability**

### **Load Balancing**

A load balancer is a service that distributes incoming traffic across multiple backend EC2 instances to improve application performance and availability. It provides a single point of access for users, who are unaware of the backend servers handling their requests.

#### **Key Benefits:**

* **Load Distribution:** Distributes traffic among multiple EC2 instances, improving performance as user volume increases.
* **Failure Handling:** Automatically redirects traffic from unhealthy instances using health checks.
* **SSL Termination:** Manages HTTPS connections and offloads SSL encryption.
* **High Availability:** Supports high availability across multiple zones.
* **Traffic Segmentation:** Separates public and private traffic in the cloud.

#### **Types of AWS Load Balancers:**

* **Classic Load Balancer (CLB):** Older version (2009) supporting HTTP, HTTPS, TCP, SSL, and security protocols. AWS no longer recommends using it.
* **Application Load Balancer (ALB):** Introduced in 2016, supports HTTP, HTTPS, and WebSocket protocols. Ideal for routing requests based on URL or host.
* **Network Load Balancer (NLB):** Introduced in 2017, supports TCP, TLS, and UDP protocols. Best for extreme performance and static IPs.
* **Gateway Load Balancer (GWLB):** Introduced in 2020, operates at the network layer for security appliances and services.

#### **Health Checks:**

* Health checks are performed to ensure that only healthy EC2 instances receive traffic. If an instance fails the check (e.g., no response or status code other than 200), the load balancer will stop routing traffic to it.

#### **Security:**

* Load balancers allow connections over HTTP/HTTPS, while EC2 instances are configured to accept traffic only from the load balancer using security group settings. This enhances security by ensuring traffic is routed correctly.

**Application Load Balancer (ALB):**

A **Layer 7** load balancer designed to handle **HTTP/HTTPS** traffic. It provides advanced routing features such as **path-based** (e.g., /users, /posts) and **host-based** (e.g., example.com) routing. It supports **WebSockets**, **HTTP/2**, and **redirects** (e.g., HTTP to HTTPS). Best for **microservices** and **containerized applications**, it integrates seamlessly with **ECS**, **Lambda functions**, and **EC2 instances**. The ALB is ideal for applications requiring intelligent routing and handling dynamic traffic.

**Network Load Balancer (NLB):**A **Layer 4** load balancer optimized for **TCP** and **UDP** traffic. It is designed for **high performance** and **ultra-low latency**, capable of handling millions of requests per second. NLB allows the use of **static IPs**, with one per Availability Zone, and supports **Elastic IPs** for fixed, public IP addresses. Ideal for applications requiring **extreme performance**, **static IPs**, or **TCP/UDP traffic**. It supports target groups consisting of **EC2 instances** or **private IP addresses**, including on-premises servers.

**Gateway Load Balancer (GWLB):**A **Layer 3** load balancer primarily used for routing traffic to 3rd party **virtual appliances** like **firewalls**, **IDS/IPS**, and other network security services. It enables deep integration with security solutions and supports traffic inspection at the **network layer**. Ideal for organizations needing high-level control over network traffic while integrating security appliances into their architecture. GWLB simplifies managing complex, network-level traffic flows.

### **Elastic Load Balancer Sticky Sessions**

Sticky sessions (session affinity) in Elastic Load Balancers ensure that a client's requests are consistently routed to the same backend instance for the duration of the session. This is achieved using cookies:

* **ALB:** Uses **application-based cookies** for stickiness. These can be managed by the ALB or the application itself. Useful for session-specific applications requiring client affinity.
* **CLB:** Supports **duration-based stickiness**, controlled by the load balancer. The stickiness duration is configurable, but the backend instance may change if it becomes unhealthy.
* **NLB:** Does **not support sticky sessions** because it operates at Layer 4 and focuses on TCP/UDP traffic without session-level management.

**Cross-zone**

Cross-zone load balancing evenly distributes traffic across all instances, regardless of availability zone. It’s enabled by default for ALB without inter-AZ charges, while NLB and GWLB require enabling and incur charges.

**SSL/TLS certificates**

SSL/TLS certificates enable encrypted in-transit communication, ensuring secure data transfer between clients and servers. TLS is the modern standard, although commonly referred to as SSL. Public SSL certificates, issued by certificate authorities (e.g., Digicert, Let's Encrypt), are attached to load balancers for HTTPS connections. Load balancers perform SSL termination, decrypting traffic at the load balancer while backend communication may use HTTP within the secure VPC.

**Server Name Indication (SNI)** enables hosting multiple SSL certificates on a single load balancer by letting clients specify the hostname during the initial SSL handshake. SNI is supported by ALB, NLB, and CloudFront but not by CLB. With SNI, the load balancer routes traffic based on the hostname and loads the corresponding SSL certificate.

**Connection Draining**

Connection Draining (CLB) or Deregistration Delay (ALB/NLB) allows active requests to complete before an EC2 instance is deregistered. New requests are redirected to other instances, while existing ones finish during a configurable period (1-3,600 seconds, default 300). Setting it to 0 disables the feature.

**Auto Scaling Groups**

An Auto Scaling Group (ASG) automatically adjusts the number of EC2 instances to handle changes in load. It scales out by adding instances during increased demand and scales in by removing instances when demand decreases, ensuring efficiency and cost optimization. ASGs define minimum, desired, and maximum instance capacities and integrate with load balancers to distribute traffic and handle health checks. Instances deemed unhealthy are replaced automatically. ASGs use launch templates to define instance configurations and can integrate with CloudWatch alarms for dynamic scaling based on metrics like CPU usage.

### **Scaling Policies for Auto Scaling Groups (ASGs)**

* **Dynamic Scaling**
  + **Target Tracking Scaling:** Simplifies scaling by maintaining a specific metric (e.g., CPU utilization) at a target value (e.g., 40%). Automatically scales in or out to meet the target.
  + **Simple/Step Scaling:** Uses predefined CloudWatch alarms to add or remove capacity based on specific thresholds.
* **Scheduled Scaling**
  + Anticipates scaling needs based on known usage patterns, such as increasing minimum capacity at specific times (e.g., Fridays at 5 PM).
* **Predictive Scaling**
  + Forecasts load patterns using historical data and schedules scaling actions accordingly, ideal for cyclical or predictable traffic.

### **Key Metrics for Scaling**

* **CPU Utilization:** Useful for applications that rely on computation, scaling up when instances are overutilized.
* **RequestCountPerTarget:** Ensures optimal performance by scaling based on the number of requests handled per target.
* **Network In/Out:** Suitable for network-bound applications with significant uploads or downloads.
* **Custom Metrics:** Application-specific metrics pushed to CloudWatch for tailored scaling.

### **Scaling Cooldown**

* **Cooldown Period:** A default 300-second pause after scaling actions to stabilize metrics and reasing
* **Optimization:** Using pre-configured AMIs can reduce instance startup times, enabling faster scaling and shorter cooldowns.

### **AWS RDS, Aurora and ElasticCache**

### **AWS RDS**

AWS Relational Database Service (RDS) is a fully managed service for deploying and maintaining SQL-based databases like PostgreSQL, MySQL, MariaDB, Oracle, Microsoft SQL Server, and AWS Aurora. It automates tasks such as provisioning, patching, backups, and monitoring, ensuring high availability and scalability without requiring direct server access.

### **Key Features**

* **Automation**: Handles backups, point-in-time restores, and maintenance.
* **Scaling**: Supports vertical scaling (instance size) and horizontal scaling (read replicas).
* **Disaster Recovery**: Multi-AZ deployments for failover support.
* **Storage Auto Scaling**: Automatically expands storage when free space drops below 10%, with safeguards to avoid overgrowth.

### **RDS Read Replicas vs. Multi-AZ**

**Read Replicas**

* **Purpose**: Scale read operations by creating up to 15 replicas of a primary database.
* **Replication**: Asynchronous, meaning replicas may lag and data is eventually consistent.
* **Use Case**: Offload read-heavy operations like analytics or reporting to replicas to avoid overloading the primary database.
* **Deployment**: Can be within the same AZ, across AZs, or cross-region. Cross-region incurs replication fees, but intra-region does not.
* **Promotion**: Read replicas can be promoted to standalone databases for write operations if needed.
* **Limitations**: Only supports SELECT queries (no INSERT, UPDATE, or DELETE).

**Multi-AZ**

* **Purpose**: Enhance disaster recovery by maintaining a standby database in a separate AZ.
* **Replication**: Synchronous, ensuring full consistency between primary and standby.
* **Use Case**: Ensures high availability during failures (e.g., AZ outages, instance issues) with automatic failover.
* **Failover**: Uses a single DNS name, automatically redirecting traffic to the standby when the primary fails.
* **Limitations**: Standby is for failover only and cannot handle reads or writes.

**Amazon Aurora**

Aurora is a proprietary, cloud-optimized relational database service by AWS, compatible with MySQL and PostgreSQL. It offers significantly enhanced performance, with 5x MySQL and 3x PostgreSQL performance on RDS. Aurora automatically manages storage scaling, high availability, and fault tolerance, making it ideal for high-demand applications. Works with DNS endpoints such as Read and Write endpoints

* **Replica Auto Scaling**Automatically scales read replicas based on demand, adding replicas during high CPU usage to distribute traffic through the Reader Endpoint and reduce load.
* **Custom Endpoints**Allows grouping specific replicas for targeted workloads, such as analytics, using custom endpoints instead of the Reader Endpoint for optimized query handling.
* **Aurora Serverless**Manages database instantiation and scaling dynamically based on usage, ideal for unpredictable workloads, with per-second billing and no capacity planning required.
* **Global Aurora**Enables low-latency reads and disaster recovery with cross-region replication in under one second and failover recovery in under a minute, supporting global workloads.
* **Aurora Machine Learning Integration**Integrates with AWS ML services like SageMaker and Comprehend, enabling predictions (e.g., fraud detection, recommendations) directly via SQL without ML expertise.

**RDS and Aurora Security**

**Data Encryption at Rest**RDS and Aurora support KMS-based encryption for volumes. Encryption must be enabled at database creation, as unencrypted databases cannot have encrypted read replicas. To encrypt an existing unencrypted database, take a snapshot, then restore it as an encrypted database.

**In-Flight Encryption**Data between clients and databases is encrypted using TLS. AWS provides root certificates for secure client connections.

**Database Authentication**Databases support classic username/password authentication and IAM-based authentication, allowing seamless integration with AWS services like EC2 instances using IAM roles.

### **Amazon RDS Proxy** Amazon RDS Proxy is a fully managed database proxy service that enables applications to pool and share connections to RDS databases, improving efficiency by reducing the load on database resources like CPU and RAM. Instead of connecting directly to the database, applications connect to the proxy, which manages and optimizes database connections, reducing connection timeouts and improving scalability.

### **Key Features**:

### **Connection Pooling**: RDS Proxy manages connections, reducing the strain on the database.

### **Auto-scaling and High Availability**: It is serverless, auto-scaling, and highly available across multiple availability zones (AZs).

### **Failover Improvement**: Reduces failover time by up to 66% during database instance failovers (RDS and Aurora).

### **No Code Changes**: Applications don’t need code changes; they connect to the RDS Proxy instead of the database directly.

### **IAM Authentication**: Enforces IAM-based authentication and securely stores credentials in AWS Secrets Manager.

### **VPC Security**: The proxy is only accessible within the VPC for enhanced security.

### **Lambda Support**: Helps manage large numbers of Lambda function connections by pooling them, preventing connection overload.

### **Amazon ElastiCache**

Amazon ElastiCache is a fully managed service for in-memory caching with high performance and low latency. It supports **Redis** and **Memcached** to reduce database load by caching frequent queries and improving application performance. ElastiCache can help applications become stateless by storing session data in the cache. It requires application code changes to interact with the cache, either before or after querying the database.

### **ElastiCache Architecture**

* **Cache Hit:** When data is found in the cache, it's returned without querying the database.
* **Cache Miss:** If data isn’t in the cache, it's fetched from the database and then stored in the cache for future requests.
* **Session Management:** User session data can be stored in ElastiCache to maintain statelessness in applications. If the user is redirected to another application instance, the session data is retrieved from the cache.

### **Redis vs Memcached**

* **Redis**
  + Supports **multi-AZ** with **auto-failover** and **read replicas** for high availability.
  + Offers **data durability** with AOF persistence and backup/restore features.
  + Good for use cases like leaderboards due to support for **sets** and **sorted sets**.
  + **Replication** provides high availability and data resilience.
* **Memcached**
  + **Sharding**: Data is split across multiple nodes.
  + **No high availability or replication** unless using the serverless version.
  + **Multi-threaded architecture** can improve performance.
  + **Backup/restore** available in the serverless version only.
  + Suitable for use cases where simplicity and performance are needed.

**ElastiCache Security**

* **Redis Security**: ElastiCache supports IAM authentication for Redis, providing API-level security. For Redis clusters, **Redis AUTH** allows setting passwords and tokens for added security. It also supports **SSL in-flight encryption** and can be protected by **security groups**.
* **Memcached Security**: Memcached uses **SASL-based authentication**, an advanced mechanism for access control.

**Data Loading Patterns**

1. **Lazy Loading**: Data is cached only on cache misses, and can become stale.
2. **Write Through**: Data is added/updated in the cache every time it’s written to the database, ensuring no stale data.
3. **Session Store**: ElastiCache can be used to store session data, with expiration based on **Time to Live (TTL)**.

**AWS S3**

**S3 and S3 Website**: S3 is a scalable object storage service that can host static websites by enabling public read access to an S3 bucket.

**S3 Bucket Policies**: JSON-based policies define fine-grained access controls to manage permissions at the bucket level.

**S3 Versioning**: Enables object version tracking to recover deleted or overwritten files.

**S3 Replication**: Automatically replicates objects between buckets (cross-region or same-region) to enhance data durability and compliance.

**S3 Storage Classes**: Offers tiered storage (e.g., Standard, Intelligent-Tiering, Glacier) to optimize costs for varying access patterns.

**S3 Lifecycle Rules and Analytics**: Automates object transitions between storage classes based on usage patterns, with analytics aiding decision-making.

**S3 Requester Pays**: Shifts data download costs to the requester instead of the bucket owner.

**S3 Event Notifications**: Triggers services like Lambda, SQS, or SNS on object events (e.g., upload, delete).

**S3 Performance**: Optimized for low latency and high throughput, supporting parallelism and multipart uploads for large files.

**S3 Batch Operations**: Enables large-scale management tasks (e.g., object tagging, ACL updates) across multiple objects.

**S3 Storage Lens**: Provides detailed storage usage insights and recommendations for cost optimization.

**S3 Encryption and Default Encryption**: Protects data at rest with server-side encryption (SSE-S3, SSE-KMS) or client-side encryption, with default encryption enforcing these settings.

**S3 CORS**: Configures cross-origin resource sharing for secure access from external domains.

**S3 MFA Delete**: Adds extra security by requiring multi-factor authentication to delete objects or change versioning settings.

**S3 Access Logs**: Captures detailed logs of bucket access for monitoring and auditing.

**S3 Pre-Signed URLs**: Generates temporary, signed URLs to securely grant time-limited access to private objects.

**S3 Glacier Vault Lock and Object Lock**: Glacier Vault Lock enforces compliance for archived data, while Object Lock protects objects against deletion.

**S3 Access Points**: Simplifies data access with unique endpoints and policies for shared buckets.

**S3 Object Lambda**: Transforms objects dynamically during retrieval, enabling custom data processing without modifying the source.

**Amazon Route 53**

**Amazon Route 53** is a scalable DNS (Domain Name System) service that routes end-user requests to appropriate resources like EC2 instances, load balancers, or S3 buckets. It also supports domain registration, health checks, and DNS failover.

**Routing Policies**Routing policies control how Route 53 responds to DNS queries, allowing for flexible traffic management. The key routing policies are:

* **Simple Routing**: Routes traffic to a single resource, without any complex rules.
* **Weighted Routing**: Routes traffic to multiple resources based on assigned weights. Useful for load balancing.
* **Latency Routing**: Routes traffic to the resource with the lowest latency, improving user experience.
* **Failover Routing**: Routes traffic to a backup resource if the primary resource is unhealthy, ensuring high availability.
* **Geolocation Routing**: Routes traffic based on the user's geographic location, providing region-specific content.
* **Geoproximity Routing**: Routes traffic based on the geographic proximity to resources, with optional biasing.
* **Multivalue Answer Routing**: Routes multiple IP addresses for a single DNS query, improving availability and fault tolerance.

**AWS CloudFront and Global Accelerator**

**Amazon CloudFront**

Amazon CloudFront is a Content Delivery Network (CDN) that caches content at global edge locations, improving performance by delivering content with low latency. By serving content from the nearest edge location, CloudFront reduces load times and ensures fast delivery of both static and dynamic content, enhancing user experience.

* **Geo Restriction**: CloudFront allows restricting content access based on geographic location, enabling or blocking access from specific countries.
* **Price Classes**: CloudFront offers different price classes (100, 200, and All) to manage the cost by selecting specific edge locations, with Class 100 being the least expensive.
* **Cache Invalidation**: You can invalidate cached content to ensure users always get the latest version of data.

**AWS Global Accelerator**

AWS Global Accelerator improves the performance and availability of your global applications by leveraging AWS's internal global network. It routes traffic using Anycast IP, sending users to the nearest AWS edge location, reducing latency and enhancing connection stability.

* **Anycast IP**: Global Accelerator assigns two Anycast IPs to your application, ensuring that user traffic is routed to the nearest edge location. From there, the traffic travels through AWS's private network to the application, bypassing public internet latency and reducing hops.
* **Supports Various Resources**: It works with EC2 instances, Elastic IPs, Application Load Balancers (ALB), and Network Load Balancers (NLB), offering both public and private configurations.
* **Performance**: Global Accelerator optimizes latency by routing traffic through the lowest-latency AWS edge location and supports fast regional failover, ensuring consistent performance. In case of failure, automated failover occurs in under a minute.
* **Security**: It provides DDoS protection through AWS Shield, and only two external IPs need to be whitelisted by clients, simplifying security management.
* **Health Checks**: Continuous health checks monitor application performance, enabling fast failover if an endpoint is unhealthy.

**Global Accelerator vs. CloudFront**

* **CloudFront**: Primarily used for content delivery, improving performance for both static (e.g., images) and dynamic content (e.g., APIs). It caches content at edge locations.
* **Global Accelerator**: Improves performance for applications over TCP/UDP protocols. It doesn’t cache content but focuses on reducing latency and providing static global IP addresses, ideal for use cases like gaming, IoT, VoIP, or applications requiring deterministic failover.

**Messaging and Decoupling Applications**

**Amazon Simple Queue Service (SQS**

Amazon Simple Queue Service (SQS) is a fully managed queuing service that enables decoupling of application components. It allows producers to send messages into queues, which are processed by consumers. Multiple producers can send messages to a queue, while multiple consumers can process them in parallel.

* **Producers & Consumers**: Producers send messages (up to 256 KB) to the queue using the SendMessage API. Consumers poll for messages and process them. After processing, messages are deleted from the queue with DeleteMessage to avoid duplication.
* **Queue Types & Throughput**: SQS provides unlimited throughput and message retention (4 days default, 14 days max). It's designed for high-volume, low-latency message processing, with at least once delivery and best-effort ordering.
* **Scaling**: To scale, add more consumers. Auto Scaling groups can adjust based on queue length metrics (ApproximateNumberOfMessages), ensuring efficient message processing even during traffic surges.
* **Use Cases**: SQS is commonly used to decouple microservices, manage workflows, and handle background tasks like video processing. For example, an order processing system can use SQS to queue orders for later processing by another service.
* **Security**: SQS supports encryption in transit (HTTPS) and at rest (KMS). IAM and SQS access policies control who can access queues and manage permissions, including cross-account access.

SQS is essential for building scalable, decoupled systems where components can work independently. It’s often integrated with EC2 instances, Lambda functions, and Auto Scaling for flexible, high-throughput workloads.

* **Message Visibility Timeout**: This setting prevents other consumers from processing the same message while a consumer is working on it. After a message is received, it becomes "invisible" to other consumers for the duration of the visibility timeout. If the consumer fails to process and delete the message within the timeout, it becomes visible again for another consumer.
* **Long Polling**: By default, SQS uses short polling (polling for messages with immediate responses), but long polling reduces unnecessary API calls. With long polling, consumers wait up to 20 seconds for new messages to arrive, reducing costs and improving efficiency by minimizing empty responses.
* **FIFO Queues**: FIFO (First-In-First-Out) queues guarantee that messages are processed in the exact order they are sent. Unlike standard queues, FIFO queues also ensure that messages are delivered exactly once and in the correct sequence. They are ideal for scenarios where message order is critical, such as financial transactions.

**SQS with Auto Scaling and Common Patterns**

* **Auto Scaling**: EC2 instances in an Auto Scaling group poll an SQS queue. CloudWatch monitors queue length, triggering scaling actions to add more EC2 instances when the queue length exceeds a set threshold.
* **Buffering Database Writes**: SQS buffers transaction requests before writing to databases (RDS, Aurora, DynamoDB), preventing data loss during high traffic. Messages are processed by an Auto Scaling group and deleted once written to the database.
* **Decoupling Application Tiers**: SQS decouples the front-end and back-end by queuing requests, allowing the back-end to process them at scale, ensuring efficient handling of spikes in demand.

### **Amazon SNS Overview and Key Patterns**

* **SNS (Simple Notification Service)** enables the **Pub/Sub pattern**, where a message is published to a topic, and many subscribers (e.g., SQS, Lambda, emails) can receive the message. It's scalable, supporting up to 12 million subscriptions per topic.
* **Subscribers**: Can include SQS queues, Lambda functions, HTTP endpoints, mobile notifications (via platform endpoints), emails, and more.
* **Use Cases**: SNS is widely used for decoupling services, handling notifications, and sending messages from AWS services like CloudWatch, Auto Scaling, or S3 to SNS topics.

### 

### 

### **SNS + SQS Fan-Out Pattern**

* **Fan-Out**: Instead of sending messages directly to multiple SQS queues, you send one message to an SNS topic. Multiple SQS queues can subscribe to this topic and receive the message.
* **Benefits**: Provides decoupling, message persistence, retry capabilities, and easy scaling by adding more SQS queues as subscribers. Commonly used with S3 events and other AWS service notifications.
* **Cross-Region**: SNS can deliver messages across regions to SQS queues if access is allowed.

### **SNS FIFO and Message Filtering**

* **SNS FIFO**: Supports first-in-first-out (FIFO) message ordering with deduplication and limited throughput. Used when maintaining order in messages is crucial, like with SQS FIFO queues as subscribers.
* **Message Filtering**: Allows filtering messages at the subscriber level. For example, an SQS queue can be set to only receive specific messages, such as "Placed" orders, using a JSON policy.
* **Key Integration Example**: S3 events sent to an SNS topic can fan out to multiple SQS queues or trigger Lambda functions, offering flexibility in handling different types of events (e.g., placed, canceled orders).

### **Security and Access Control**

* **Encryption**: SNS provides encryption in-flight and at-rest using KMS keys. Clients can also encrypt messages before sending them.
* **IAM Policies**: Used for controlling access to SNS resources, similar to S3 bucket policies. This allows you to manage cross-account access and permissions for services to send messages to SNS topics.

**Kinesis Data Streams**

Kinesis Data Streams enables real-time data streaming by organizing data into shards for scalable ingestion and processing. Producers send data as records, and consumers like Lambda or Firehose process it. It supports retention, encryption, key-based ordering, and flexible capacity modes for real-time analytics.

**Structure and Shards**: Streams are made up of shards, which determine ingestion and consumption capacity. Shards must be provisioned beforehand. Each shard supports up to 1 MB/s or 1,000 messages per second for producers and 2 MB/s for consumers.

**Producers**: Send data to streams using the AWS SDK, Kinesis Producer Library (KPL), or Kinesis Agent. Records consist of a **partition key** (determining shard assignment) and a **data blob** (up to 1 MB).

**Consumers**: Consume data via SDK, Kinesis Client Library (KCL), AWS Lambda, Kinesis Data Firehose, or Kinesis Data Analytics. Enhanced fan-out mode enables 2 MB/s per shard per consumer.

**Retention and Data Properties**: Data is immutable and can be retained from 1 to 365 days, allowing replay. Key-based ordering is achieved using partition keys.

**Capacity Modes**:

* **Provisioned Mode**: Manually scaled shards, with predictable costs.
* **On-Demand Mode**: Automatically scales based on throughput, suitable for variable workloads.

**Security**: IAM policies control access, with encryption at rest (KMS), in transit (HTTPS), and optional client-side encryption. VPC endpoints allow private access. CloudTrail monitors API calls.

**Kinesis Data Firehose**

Amazon Kinesis Data Firehose is a fully managed, serverless service designed to reliably capture, transform, and deliver streaming data to specified destinations with minimal effort. It supports various producers, including applications, SDKs, Kinesis Data Streams, and CloudWatch logs/events, and can deliver data to destinations such as Amazon S3, Redshift, and OpenSearch. It also integrates with third-party services like Splunk and Datadog or custom HTTP endpoints. Data can optionally be transformed using AWS Lambda before delivery.

### **Kinesis vs. SQS FIFO for Ordering Data**

**Kinesis Data Streams**

* Data is partitioned using a **partition key**, ensuring records with the same key (e.g., truck ID) are always sent to the same shard.
* Ordering is guaranteed at the **shard level**.
* Limited to one consumer per shard; parallelism depends on the number of shards.
* Supports **high throughput** (up to 1 MB/s per shard) and scalable by increasing shards.
* Use when ordering is required for large-scale streaming data with high throughput needs.

**SQS FIFO Queues**

* Ordering is achieved using **message group IDs**, grouping related messages and allowing parallel consumption.
* One **consumer per group ID**; supports up to 300 messages/sec (or 3000 with batching).
* No predefined shards; supports dynamic scaling of consumers based on group IDs.
* Use when multiple consumers are required and ordering within a group is essential.

**Comparison**

* **Kinesis**: Best for high-throughput streaming with shard-level ordering and fixed parallelism.
* **SQS FIFO**: Ideal for workloads requiring dynamic consumer scaling and group-level ordering.

### **Constraints and Microservices**

### **ECS Structure**

Amazon ECS is a container orchestration service that organizes Docker containers into tasks, which run on a cluster of compute resources.

* **Cluster**: Logical grouping of resources (EC2 instances or Fargate tasks).
* **Task Definition**: Blueprint for application tasks, specifying container images, CPU, memory, IAM roles, and networking.
* **Tasks**: Running instances of a task definition, representing individual workloads.
* **Launch Types**: Determines whether tasks run on managed EC2 instances or in a serverless environment (Fargate).

**ECS**

Amazon Elastic Container Service (ECS) is a fully managed container orchestration service that makes it easy to run, manage, and scale containerized applications on AWS. ECS supports two main launch types:

* **EC2 Launch Type**:
  + ECS tasks are deployed on **EC2 instances** that you provision and maintain.
  + Each instance must run the **ECS Agent**, which registers it with the ECS cluster.
  + Containers are scheduled on EC2 instances, and scaling requires managing instance capacity.
* **Fargate Launch Type**:
  + **Serverless** deployment; no need to manage EC2 instances.
  + Tasks are defined with CPU and memory specifications and run automatically.
  + Simplifies scaling by increasing task count without managing infrastructure.

**IAM Roles**

* **EC2 Instance Profile**:
  + Used by the ECS Agent to make API calls (e.g., ECS, CloudWatch, ECR).
  + Only applies to the **EC2 Launch Type**.
* **ECS Task Role**:
  + Assigned per task, allowing fine-grained permissions for accessing AWS services.
  + Applies to **both EC2 and Fargate** launch types.

**Load Balancer Integration**

* **Application Load Balancer (ALB)**: Recommended for most use cases; compatible with Fargate and supports advanced features.
* **Network Load Balancer (NLB)**: Ideal for high-throughput or AWS PrivateLink use cases.
* **Classic Load Balancer (CLB)**: Legacy option; lacks advanced features and is not Fargate-compatible.

**Data Persistence**

* Use **Amazon EFS** (Elastic File System) for shared, persistent storage across tasks.
  + Compatible with both EC2 and Fargate launch types.
  + Enables multi-AZ shared storage for containerized applications.
  + Combines well with Fargate for a completely serverless solution.

Amazon ECS Service Auto Scaling allows you to automatically adjust the number of ECS tasks in response to demand. You can scale based on three key metrics: **CPU Utilization**, **Memory Utilization**, and **ALB Request Count per Target**. Auto Scaling can be configured in different ways: **Target Tracking**, **Step Scaling**, and **Scheduled Scaling**.

For **EC2 Launch Type**:

* **Auto Scaling Groups (ASG)** can be used to scale EC2 instances based on CPU utilization, adding more instances when needed.
* A more advanced option is the **ECS Cluster Capacity Provider**, which automatically adjusts the ASG capacity when ECS tasks require more resources (CPU or RAM).

For **Fargate Launch Type**, scaling is easier since it's serverless. The need for manual EC2 instance scaling is eliminated, and tasks can be scaled by simply adjusting the desired task count.

### **ECS Solution Architectures**

1. **ECS Tasks Invoked by EventBridge**
   * In this architecture, an **ECS cluster** (backed by Fargate) processes objects uploaded to **S3**.
   * **EventBridge** triggers ECS tasks whenever new objects are added to S3.
   * **ECS task role** enables tasks to access S3 and send results to **DynamoDB**.
   * This creates a **serverless architecture** for processing data using ECS and Fargate.
2. **EventBridge Schedule Triggering ECS Tasks**
   * **EventBridge** schedules ECS tasks to run every hour.
   * ECS tasks can perform tasks such as batch processing files in **S3**.
   * This solution is entirely **serverless** with ECS and Fargate, and can be customized based on specific needs.
3. **ECS and SQS Integration**
   * An **ECS service** with two tasks processes messages from an **SQS queue**.
   * **Auto Scaling** adjusts the number of ECS tasks based on the queue’s load.
   * This architecture scales tasks automatically based on incoming messages.
4. **EventBridge for ECS Lifecycle Monitoring**
   * **EventBridge** can intercept ECS events like tasks starting or stopping.
   * For instance, when a task is stopped, an event is sent to **SNS** to notify administrators.
   * This enables lifecycle monitoring of ECS tasks, providing better event-driven automation.

**ECR**

**Amazon ECR (Elastic Container Registry)** is a fully managed container image registry service that allows you to store, manage, and deploy Docker container images. It integrates seamlessly with **Amazon ECS** and **Amazon EKS**, making it easy to run containerized applications.

**EKS**

Amazon EKS (Elastic Kubernetes Service) is a fully managed service that simplifies running Kubernetes clusters on AWS.

**AWS App Runner**

is a fully managed service designed to simplify the deployment of web applications and APIs at scale. It allows users to deploy applications without needing any knowledge of infrastructure management.

**Serverless**

**AWS Lambda**AWS Lambda is a serverless compute service that allows you to run code without provisioning or managing servers. Functions run on demand, triggered by events, and automatically scale.

* **No Server Management**: No need to worry about provisioning or maintaining servers.
* **Short Execution**: Functions run for up to 15 minutes per invocation.
* **On-Demand Scaling**: Automatically scales based on the number of incoming requests.
* **Pricing**: Pay only for the number of requests and compute time. Free tier includes 1 million requests and 400,000 GB-seconds of compute time per month.
* **Languages Supported**: Supports Node.js, Python, Java, C#, Ruby, PowerShell, and custom runtimes for other languages like Go and Rust.
* **Integration with AWS Services**: Easily integrates with services like API Gateway, Kinesis, S3, CloudWatch, DynamoDB, SNS, SQS, and more.

**Use Cases**

* **Thumbnail Creation**: Automatically generate image thumbnails in response to S3 events.
* **Serverless CRON Jobs**: Use CloudWatch or EventBridge to schedule Lambda functions for periodic tasks.
* **Data Processing**: Lambda can process data streams or files uploaded to S3 or DynamoDB events.

**Lambda Pricing**

* **Requests**: The first 1 million requests are free; subsequent requests cost $0.20 per million.
* **Compute Time**: Charged based on execution time (rounded to the nearest millisecond) and allocated memory, with 400,000 GB-seconds of compute time free each month.

**AWS Lambda Limits**

* **Memory Allocation**: 128 MB to 10 GB, in 64 MB increments. More memory equals more vCPU.
* **Execution Time**: Maximum of 900 seconds (15 minutes). Exceeding this is unsuitable for Lambda.
* **Environment Variables**: Up to 4 KB of environment variables.
* **Temporary Storage**: Up to 10 GB available in the /tmp folder for temporary files.
* **Concurrent Executions**: Default limit of 1,000 concurrent executions. Can be increased by request.
* **Deployment Limits**:
  + Compressed ZIP size: 50 MB
  + Uncompressed size: 250 MB

**Lambda SnapStart**

Lambda SnapStart is a feature that improves the performance of Java-based Lambda functions by up to 10x without extra costs. It works by eliminating the initialization phase, which is usually time-consuming in Java. When SnapStart is enabled, Lambda functions are invoked from a pre-initialized state, reducing startup latency.

**How It Works:**

* When a new Lambda version is published, it undergoes an initialization phase ahead of time.
* A snapshot of the initialized state (memory and disk) is taken and cached for quick access, allowing the function to start much faster.

**Lambda@Edge & CloudFront Functions**

**Customization at the Edge** allows logic execution closer to users via **CloudFront Functions** and **Lambda@Edge**.

* **CloudFront Functions**: Lightweight, high-scale functions (e.g., header manipulation, URL redirects). Supports JavaScript, sub-millisecond execution, and affects viewer requests/responses with a 1ms max execution time.
* **Lambda@Edge**: More complex functions (e.g., user authentication, routing). Supports Node.js/Python, up to 5-10 seconds execution, and impacts viewer, origin requests, and responses. Offers more customization options with CPU/memory adjustment.

RDS and Aurora can invoke Lambda functions in response to database events, allowing for data processing, such as sending emails when new records are inserted. This integration is supported by **RDS for PostgreSQL** and **Aurora MySQL** but requires manual setup within the database, not through the AWS console.

* Ensure proper **network connectivity** between the RDS instance and Lambda (e.g., public internet, NAT gateway, or VPC endpoints).
* Lambda invocation requires correct **IAM permissions** for the RDS instance.
* **RDS event notifications** provide info about the database instance (e.g., creation, start) but not data events.
* For data events, use direct Lambda invocation from RDS; **event notifications** can be sent to **SNS** or **EventBridge** for further processing (e.g., Lambda, SQS).

**DynamoDB**

DynamoDB is a NoSQL database where data is stored in **tables** with a **primary key**. The primary key consists of:

* **Partition Key**: A unique identifier for each item, like a hash of the key. It determines how data is distributed across partitions in DynamoDB.
* **Sort Key (optional)**: An additional key that helps organize data within the same partition. If included, the primary key is composed of both the partition and sort keys, enabling multi-attribute querying.

Data is stored as **items** (rows) with **attributes** (columns), where each item is a key-value pair. Items can have different attributes and can grow over time without a fixed schema.

When querying, DynamoDB uses the **partition key** to locate the partition where the data is stored and the **sort key** (if present) to order or filter the data within that partition. This key-value structure ensures fast and scalable data retrieval.

* **Cloud-native** and **AWS proprietary**; does not require database provisioning.
* **Scalability**: Supports trillions of rows and hundreds of terabytes of data.
* **Performance**: Provides single-digit millisecond latency and is auto-scaling.
* **Security**: Integrated with IAM for security, authorization, and administration.
* **Table Classes**: Offers **standard** (frequent data) and **infrequent access (IA)** classes.
* **Schema flexibility**: Tables are easy to create and modify, allowing dynamic changes to attributes over time.
* **Capacity Modes**:
  + **Provisioned Mode**: You set the read and write capacity upfront. Best for predictable workloads with autoscaling options.
  + **On-Demand Mode**: Scales automatically, ideal for unpredictable workloads. You only pay for actual reads and writes.

**DynamoDB Accelerator (DAX)**DAX is a fully-managed in-memory cache for DynamoDB that reduces read congestion and enhances performance. It provides microsecond latency for cached data without requiring changes to your application logic, making it ideal for caching query results and individual objects.

**Stream Processing**DynamoDB Streams captures changes to your table (create, update, delete) for real-time processing. It integrates with Lambda for immediate actions like sending emails or can be connected to Kinesis for extended retention and processing of large-scale data streams.

**Global Tables**Global Tables replicate DynamoDB tables across multiple regions for low-latency access and active-active replication. They support read and write operations in any region with automatic data synchronization, requiring DynamoDB Streams for replication.

**Time To Live (TTL)**TTL automatically deletes items from your table once they reach an expiration time, ideal for session data or compliance purposes. Items are deleted when their TTL attribute (e.g., ExpTime) surpasses the set expiration time.

**Backups**DynamoDB offers both Point-In-Time Recovery (PITR) for continuous backups and On-Demand Backups for manual, long-term data retention. PITR provides recovery to any point within the last 35 days, while On-Demand Backups do not affect table performance and are retained until deleted.

**S3 Integration**DynamoDB can export tables to Amazon S3 for analytics, audits, or ETL processes, using formats like JSON or ION. It also supports importing data from S3 back into DynamoDB without consuming write capacity, allowing for efficient data migration and transformation.

**API Gateway**API Gateway is a serverless AWS service that enables you to create REST APIs for clients to interact with Lambda functions and other AWS services. It provides many features like authentication, usage plans, API versioning, and real-time streaming (WebSocket).

**Lambda Integration**API Gateway integrates seamlessly with Lambda to expose serverless REST APIs. It proxies client requests to Lambda functions, making it an easy way to set up a fully serverless application without managing infrastructure.

**Security Features**API Gateway offers robust security, including IAM roles for internal applications, Amazon Cognito for external users, and custom authorizers using Lambda. You can also secure APIs with HTTPS using custom domain names and AWS Certificate Manager (ACM).

**Endpoint Types**

* **Edge-Optimized**: Best for global clients, uses CloudFront for low-latency access.
* **Regional**: Suitable for clients within the same region as the API Gateway.
* **Private**: Only accessible within a VPC, using interface VPC endpoints and resource policies.

**Integration with AWS Services**API Gateway can interact with various AWS services, including Kinesis, SQS, and Step Functions. This allows exposing services like Kinesis Data Streams to external clients securely without needing AWS credentials.

**Deployment and Configuration**API Gateway allows deployment in different environments (dev, test, prod) and supports API versioning. It also enables caching, request throttling, and response validation for better API management.

**Real-Time and Other Features**API Gateway supports WebSocket protocols for real-time communication, request transformation, SDK generation, and API specifications like Swagger or OpenAPI for easy imports and exports.

**AWS Step Functions**AWS Step Functions is a serverless orchestration service that allows you to build visual workflows to coordinate tasks, commonly involving Lambda functions. It enables the creation of complex workflows with clear sequencing, error handling, and decision-making.

**Features**

* **Task Management**: Supports sequencing, parallel processing, timeouts, and error handling.
* **Integrations**: Works with Lambda, EC2, ECS, API Gateway, SQS, and on-premises servers.
* **Human Approval**: Allows inclusion of manual steps for approvals within workflows.

**Use Cases**Ideal for scenarios like order fulfillment, data processing, and managing complex workflows with visual representations. It simplifies creating and maintaining workflows across various AWS services.

**Amazon Cognito Overview**

Amazon Cognito provides identity and authentication solutions for web and mobile applications, targeting users outside of AWS accounts. It has two main components: **Cognito User Pools** for user authentication and **Cognito Identity Pools** for granting temporary AWS credentials.

### **Cognito User Pools (CUP)**

* A serverless database for managing app users with sign-in functionality.
* Supports features like password reset, email/phone verification, multi-factor authentication, and social login integration (e.g., Facebook, Google).
* Integrates with **API Gateway** and **Application Load Balancer (ALB)** for token-based authentication, verifying users before passing their identity to backend services.
* Shifts authentication responsibility from backend services to API Gateway or ALB, ensuring secure access.

### **Cognito Identity Pools (Federated Identities)**

* Issues temporary AWS credentials for direct access to AWS services like S3 or DynamoDB.
* Supports users authenticated through Cognito User Pools, social identity providers, SAML, or OpenID Connect.
* IAM policies can be customized for fine-grained access control, such as implementing row-level security in DynamoDB.
* Enables guest access with default IAM roles when specific roles are not defined.

### **Key Use Cases**

* Managing external app users and providing identity-based access.
* Fine-grained access control in DynamoDB (e.g., limiting user access to specific rows).
* Secure, direct AWS resource access without API Gateway or ALB.

**AWS Monitoring & Audit**

**Amazon CloudWatch Metrics**

Amazon CloudWatch Metrics enables monitoring of AWS services and custom resources through time-based metrics. Metrics are organized into namespaces and have attributes called dimensions, allowing detailed filtering and analysis.

### **Key Features**

* **Default Metrics:** Automatically provided for AWS services (e.g., CPUUtilization for EC2, bucket size for S3).
* **Custom Metrics:** User-defined metrics, such as memory usage for EC2 instances, for specific monitoring needs.
* **Dimensions:** Attributes like instance ID or environment, supporting up to 30 dimensions per metric.
* **Granularity:** Default data points are collected every 5 minutes; enabling detailed monitoring reduces this to 1 minute.

### **Integrations and Streaming**

* Metrics can be streamed in near real-time to **Amazon Kinesis Data Firehose** for further processing.
  + From Firehose, data can be sent to **Amazon S3** (analyze via Athena), **Amazon Redshift** (data warehousing), or **Amazon OpenSearch** (dashboards and analytics).
* Direct integration with third-party tools like Datadog, Dynatrace, Splunk, and New Relic for advanced analytics.

### **CloudWatch Logs**

CloudWatch Logs is a fully managed service for collecting, storing, and analyzing log data from AWS resources and applications. It enables real-time monitoring, troubleshooting, and long-term log storage with customizable retention policies.

### **Key Components**

CloudWatch Logs operates through **Log Groups** and **Log Streams**.

* **Log Groups**: Named containers representing an application or system.
* **Log Streams**: Specific instances of logs within a group, such as container logs or log files.  
  Retention policies determine how long logs are stored, ranging from 1 day to 10 years or indefinitely. Logs are encrypted by default, with optional KMS-based encryption.

### **Log Sources**

Logs from AWS services and applications can be sent to CloudWatch Logs through various mechanisms:

* Unified Agent (replacing the older Log Agent) collects application and server logs.
* Elastic Beanstalk, ECS, and Lambda send logs from applications, containers, and functions.
* VPC Flow Logs capture network metadata, while API Gateway tracks request details.
* CloudTrail and Route 53 logs events and DNS queries respectively.

### **Querying Logs with Logs Insights**

Logs Insights is a built-in query engine for CloudWatch Logs, allowing historical log data analysis.

* Queries use a custom language to filter, sort, and calculate metrics.
* Results can be visualized, exported, or added to dashboards.
* It supports querying across multiple log groups, even across accounts.

### **Exporting and Streaming Logs**

Logs can be exported in batch mode or streamed in real time:

* **Batch Export**: Send logs to S3 using the CreateExportTask API, with a 12-hour processing time.
* **Real-Time Streaming**: Use subscription filters to deliver logs to destinations:
  + **Kinesis Data Streams**: For downstream analytics.
  + **Kinesis Data Firehose**: Near real-time delivery to S3 or OpenSearch.
  + **Lambda**: For custom real-time processing.

### **Cross-Account Log Aggregation**

Logs can be aggregated across AWS accounts using destinations and subscription filters:

1. Create a subscription filter in the sender account.
2. Define a destination (e.g., Kinesis Stream) in the recipient account.
3. Attach IAM roles and access policies to enable cross-account log streaming.

### **CloudWatch Agents**

CloudWatch Agents are used to collect and send logs and metrics from EC2 instances (or on-premises servers) to CloudWatch. By default, EC2 instances do not send logs to CloudWatch, so you need to configure and start an agent to send the logs you need.

### **Types of CloudWatch Agents**

1. **CloudWatch Logs Agent** (Old version)
   * Sends only logs to CloudWatch Logs.
   * Must be manually installed on EC2 instances or on-premises servers.
2. **CloudWatch Unified Agent** (Newer version)
   * Sends both logs and system-level metrics (e.g., CPU, RAM, disk, network) to CloudWatch.
   * Supports configuration through **SSM Parameter Store** for centralized management.

### 

### 

### 

### **CloudWatch Alarms**

CloudWatch Alarms monitor metrics and trigger actions or notifications based on predefined thresholds. They help automate responses to changes in system performance and can integrate with EC2, Auto Scaling, and SNS for notifications or other actions.

### **Alarm States**

* **OK**: The metric is within the defined threshold.
* **INSUFFICIENT\_DATA**: Not enough data to determine the alarm state.
* **ALARM**: The metric breaches the threshold, triggering the alarm.

### **Key Alarm Configurations**

* **Period**: Time window for evaluating the metric (can range from seconds to minutes).
* **High-resolution metrics**: Supports resolutions as low as 10 seconds.

### **Alarm Actions**

1. **EC2 Actions**: Stop, terminate, reboot, or recover EC2 instances.
2. **Auto Scaling Actions**: Scale out or scale in.
3. **SNS Notifications**: Trigger SNS notifications, which can further invoke Lambda functions for customized responses.

### **Composite Alarms**

* **Purpose**: Monitor the state of multiple alarms, combining them with logical AND/OR conditions.
* **Benefit**: Reduces alarm noise by triggering alerts only when specific combinations of conditions are met.
* **Example**: Combine CPU and network alarms to trigger only when both thresholds are breached simultaneously.

### **EC2 Instance Recovery**

* **Status Checks**: Monitors EC2 instance status (instance, system, and attached EBS health).
* **Recovery**: Trigger recovery actions to move an EC2 instance to another host without losing IP or metadata.
* **SNS Alerts**: Send alerts when an instance recovery occurs.

### **CloudWatch Logs Integration**

* **Metric Filters**: Use CloudWatch Logs metrics (e.g., error occurrences) as triggers for alarms.
* **Testing**: Use the set alarm states CLI command to test alarm configurations and notifications without meeting thresholds.

**Amazon EventBridge**

Amazon EventBridge (formerly CloudWatch Events) is a fully managed event bus service that enables event-driven architectures. It allows you to trigger actions based on events from AWS services, custom applications, or third-party SaaS providers. EventBridge is ideal for automating tasks such as scheduling cron jobs, triggering Lambda functions, or sending notifications.

**Key Features:**

* **Event Sources:** EventBridge integrates with AWS services (e.g., EC2, S3, CloudTrail, CodeBuild) and third-party SaaS applications (e.g., Zendesk, Datadog).
* **Event Buses:**
  + **Default Event Bus**: Receives events from AWS services.
  + **Partner Event Bus**: Receives events from third-party partners.
  + **Custom Event Bus**: Allows you to create your own event bus to send and receive custom events.
* **Event Filtering:** EventBridge lets you filter events, allowing you to only react to specific events (e.g., from a certain S3 bucket).
* **Destinations:** Supports sending events to multiple destinations such as Lambda, SNS, SQS, Step Functions, ECS, and more.
* **Scheduling:** You can set cron-like schedules to trigger events, such as triggering a Lambda function every hour.
* **Event Replay:** Archived events can be replayed for debugging and troubleshooting.
* **Schema Registry:** EventBridge analyzes and infers the schema of events, making it easier to work with events programmatically. It also supports versioning of schemas.
* **Resource-Based Policies:** Allows cross-account event sharing by setting permissions for event buses.

**CloudWatch Insights**

Amazon CloudWatch provides various insights for monitoring and troubleshooting resources effectively. Here's a concise overview:

* **Container Insights**: Collects, aggregates, and visualizes metrics and logs from containers on ECS, EKS, Kubernetes on EC2, and Fargate. Uses a containerized CloudWatch agent for Kubernetes.
* **Lambda Insights**: Offers detailed monitoring for serverless applications on AWS Lambda, including system metrics (CPU, memory, network), cold starts, and worker shutdowns. It runs as a Lambda layer.
* **Contributor Insights**: Analyzes logs to identify top contributors to system performance. Useful for finding heavy network users, bad actors, or problematic URLs. Works with VPC, DNS, and other AWS logs.
* **Application Insights**: Automates dashboards for diagnosing issues in applications running on EC2, .NET, Java, IIS, and more. Integrates with related AWS resources (RDS, S3, ELB, Lambda) and uses SageMaker for insights. Alerts are sent to EventBridge and SSM OpsCenter.

### **Amazon CloudTrail Summary**

Amazon CloudTrail is a service that provides governance, compliance, and auditing for AWS accounts by logging API calls, actions, and events. Here's a concise breakdown:

#### **Key Features:**

* **Enabled by Default**: Automatically logs API calls from AWS Console, SDK, CLI, and other services.
* **Event Retention**: Stores events for 90 days by default, with the option to send logs to Amazon S3 for long-term retention and analysis using Athena.

#### **Event Types:**

1. **Management Events**:
   * Logs actions performed on AWS resources (e.g., creating/deleting subnets or attaching IAM policies).
   * Subdivided into:
     + **Read Events**: Non-destructive actions like listing resources.
     + **Write Events**: Destructive actions like deleting resources.
   * Enabled by default.
2. **Data Events**:
   * Tracks high-volume activities like S3 object actions (e.g., PutObject, GetObject) and Lambda executions (Invoke API).
   * Not logged by default due to volume but can be enabled as needed.
3. **Insights Events**:
   * Detects unusual activity (e.g., service bursts, misconfigurations) by analyzing Management Events.
   * Requires manual enablement and incurs additional cost.
   * Can send anomaly notifications to EventBridge for automation or alerts.

#### **Integration:**

* **CloudWatch Logs**: Use to monitor and analyze CloudTrail events in near real-time.
* **EventBridge**: Automates responses to events or anomalies detected via CloudTrail Insights.
* **Athena**: Enables querying long-term logs stored in S3 for detailed analysis.

### **AWS Config**

AWS Config is a service designed for auditing, compliance, and tracking resource configurations in AWS. It monitors changes, evaluates them against rules, and enables tracking over time.

#### **Key Features:**

* **Configuration Recording**: Tracks configuration changes of AWS resources, providing a history for analysis and troubleshooting.
* **Compliance Checks**: Evaluates resources against AWS Managed or custom rules. Rules can check for compliance in real-time or at regular intervals.
* **Notifications**: Alerts can be sent via SNS or EventBridge for non-compliant resources or changes.

#### **Config Rules:**

* **AWS Managed Rules**: Predefined rules (e.g., checking public access to S3 buckets).
* **Custom Rules**: User-defined rules using Lambda functions (e.g., verifying instance types in specific accounts).
* **Evaluation Triggers**: Rules can trigger evaluations on configuration changes or at scheduled intervals.

#### **Remediation:**

* **Automated Actions**: Use AWS Systems Manager (SSM) Automation Documents for remediation of non-compliance (e.g., deactivating expired IAM keys).
* **Custom Actions**: Automate remediations via Lambda functions for more complex scenarios.
* **Retries**: Supports up to five retries for remediation attempts if non-compliance persists.

#### **Cost Considerations:**

* **Pricing**: Charged per configuration item recorded and per rule evaluation. Costs can increase with large-scale deployments.

#### **Notifications:**

* **Integration**: Use EventBridge for event-driven automation or SNS for filtered notifications sent to admin emails, Slack channels, etc.

#### **Limitations:**

* **Monitoring Only**: Config identifies compliance issues but does not block actions. Security mechanisms (e.g., IAM) must handle preventive controls.

**AWS SSM Parameter Store**

AWS SSM Parameter Store provides secure storage for configuration data and secrets, offering encryption through KMS and version tracking. It is serverless, scalable, and easily integrates with other AWS services.

**Key Features**

* **Encryption**: Supports plaintext and encrypted parameters using KMS.
* **Version Tracking**: Maintains version history of parameters.
* **IAM Integration**: Uses IAM roles to control access to stored parameters.
* **EventBridge Integration**: Sends notifications for certain parameter updates or expirations.
* **CloudFormation Integration**: Can use parameters in CloudFormation templates as input.

**Parameter Organization** Parameters are stored in a hierarchy, enabling easy organization and access control. For example, parameters like DB-URL and DB-password can be stored under environment-specific paths (e.g., Dev, Prod) to control access via IAM policies.

**Public Parameters** AWS provides public parameters for certain use cases, like retrieving the latest AMI for Amazon Linux 2.

### **AWS Secrets Manager**

AWS Secrets Manager is a service designed to securely store and manage secrets, such as database credentials, API keys, and other sensitive information. It offers enhanced features compared to SSM Parameter Store, particularly around secret rotation and automation.

**Key Features**

* **Secret Rotation**: Automatically rotates secrets based on a defined schedule, with Lambda functions used to generate new secrets during each rotation.
* **Encryption**: Secrets are encrypted using AWS KMS.
* **AWS Integration**: Well-integrated with AWS services like RDS (MySQL, PostgreSQL, Aurora) for automatic credential management.
* **Multi-Region Replication**: Secrets can be replicated across multiple regions, ensuring availability and disaster recovery. This allows for seamless application operations in multiple regions.

**Use Cases**

* **Disaster Recover y**: If a primary region fails, the replicated secret in the secondary region can be promoted, ensuring uninterrupted access to secrets.
* **Region-Specific Applications**: Enables the use of the same secret for databases replicated across regions, simplifying multi-region application management.

**AWS WAF**

AWS Web Application Firewall (WAF) is a security service that helps protect web applications from common web exploits and attacks. It allows you to define custom rules to filter and monitor HTTP/HTTPS requests based on conditions like IP addresses, HTTP headers, body, and URI strings.

Key Features:

* **Customizable rules**: Define rules to block, allow, or count requests.
* **Protection**: Safeguard against SQL injection, cross-site scripting (XSS), and other threats.
* **Integration**: Works with Amazon CloudFront, Application Load Balancer, and API Gateway.
* **Managed rules**: Includes pre-configured rules from AWS and third-party vendors.
* **Scalability**: Fully managed and scalable, adjusting to traffic load automatically.
* **Web ACLs (Access Control Lists):** Define rules to allow, block, or monitor requests.
  + Filter by IP sets (up to 10,000 IPs per rule), HTTP headers, URI strings, body, size, or geo-location.
  + Support **rate-based rules** to limit requests per IP (e.g., mitigate DDoS attempts).
* **Scope:**
  + Regional for most resources.
  + Global for CloudFront.
* **Rule Groups:** Collections of reusable rules applied to multiple Web ACLs.

**AWS Shield**

AWS Shield is a managed DDoS protection service that safeguards applications running on AWS. It has two tiers:

* **Shield Standard**: Free for all AWS customers, providing automatic protection against common Layer 3 (network) and Layer 4 (transport) attacks like SYN/UDP floods and reflection attacks.
* **Shield Advanced**: A premium service (~$3,000/month) offering advanced DDoS protection for resources like EC2, ALB, CloudFront, Route 53, and Global Accelerator. It includes:

**AWS Firewall Manager**

is a service that allows you to centrally manage firewall rules across multiple AWS accounts in an organization. It enables the creation of security policies, which can include WAF rules for application protection, Shield Advanced rules for DDoS protection, security groups for EC2 and VPC resources, and Network Firewall policies at the VPC level. These policies are applied automatically to all accounts within the organization.

**AWS Glue**: AWS Glue is a fully managed ETL (Extract, Transform, Load) service that facilitates the movement and transformation of data between data stores. It automatically discovers and categorizes data, creating a centralized data catalog. Glue provides an easy-to-use interface for defining ETL jobs and managing dependencies. The service supports data integration with a wide range of data stores, including S3, Redshift, and RDS. By automating much of the ETL process, AWS Glue reduces manual intervention and helps streamline data preparation for analytics or machine learning workflows.

**AWS EMR (Elastic MapReduce)**: AWS EMR is a managed platform designed for big data processing using frameworks like Apache Hadoop, Spark, and HBase. It allows users to run data analytics, machine learning, and graph processing workloads at scale. EMR automatically provisions and scales clusters based on workload requirements, ensuring cost-efficient use of resources. The service integrates with other AWS offerings like S3, Redshift, and DynamoDB, making it easy to store and analyze large datasets. With pay-as-you-go pricing and built-in tools, EMR simplifies big data processing while offering flexibility for complex analytics and data engineering tasks.

**AWS Inspector**: AWS Inspector is an automated security assessment service that helps identify vulnerabilities and misconfigurations in applications and infrastructure. It analyzes EC2 instances and containerized applications, checking them against a set of best practices and known security vulnerabilities. The service provides detailed findings and recommendations for remediation, helping to improve the security posture of AWS workloads. Inspector integrates with AWS CloudTrail and Amazon CloudWatch for continuous monitoring and alerts. It supports compliance assessments, enabling users to ensure that their applications meet security and regulatory standards.

**AWS GuardDuty**: AWS GuardDuty is a threat detection service that uses machine learning, anomaly detection, and integrated threat intelligence to monitor AWS accounts and workloads for malicious activity. It continuously analyzes AWS CloudTrail, VPC Flow Logs, and DNS logs to identify suspicious behavior, such as unauthorized API calls or unusual network traffic patterns. GuardDuty provides actionable insights into potential threats, with alerts that can be integrated into security automation workflows. The service is easy to enable, requiring minimal configuration, and offers a cost-effective way to improve AWS security by continuously monitoring cloud environments for potential risks.

**AWS Macie**: AWS Macie is a data security and privacy service that uses machine learning to automatically discover, classify, and protect sensitive data, particularly personal identifiable information (PII) stored in Amazon S3. Macie can identify sensitive data types such as credit card numbers, social security numbers, and health information, helping organizations meet compliance requirements like GDPR and CCPA. The service provides detailed reports on data access patterns and can set up alerts for suspicious activity, improving data protection and risk management. Macie’s machine learning capabilities enable it to continuously improve over time, providing more accurate data classifications and identifying potential security risks as they arise.

**AWS Redshift**: AWS Redshift is a fully managed data warehouse service designed for high-performance data analytics. It enables users to run complex queries on large datasets, offering scalability and speed with columnar storage and parallel query execution. Redshift integrates with various AWS data sources, such as S3, DynamoDB, and RDS, as well as third-party tools, to provide a seamless data analytics experience. With features like data compression, automated backups, and the ability to scale up or down based on demand, Redshift helps businesses optimize performance and cost. Redshift Spectrum allows users to run queries directly on data stored in S3, further extending the data processing capabilities of the service.

**AWS Lake Formation**: AWS Lake Formation is a service designed to simplify the process of building, securing, and managing a data lake on AWS. It enables users to securely ingest, catalog, and store vast amounts of data from different sources, including structured, semi-structured, and unstructured data. Lake Formation offers centralized access controls, ensuring that only authorized users can access specific data sets. The service integrates with tools like AWS Glue and Amazon Redshift, making it easy to process and analyze data from the data lake. By providing a secure and scalable framework for managing large volumes of data, Lake Formation accelerates data-driven insights and supports compliance with data governance policies.

**AWS Comprehend**: AWS Comprehend is a natural language processing (NLP) service that analyzes and extracts insights from large amounts of unstructured text. It uses machine learning models to identify key phrases, entities, sentiment, language, and more. Comprehend can automatically detect sentiment in customer reviews, categorize text into topics, and extract named entities like dates or locations, making it useful for applications like content moderation and customer feedback analysis. The service is fully managed, allowing users to focus on extracting actionable insights without needing to build custom NLP models. Additionally, Comprehend can be trained to customize entities and sentiment detection specific to industry needs.

**AWS Backup**: AWS Backup is a fully managed service that automates the backup and restore processes for AWS resources, such as EC2 instances, RDS databases, DynamoDB tables, and S3 buckets. It helps ensure data availability and compliance by providing centralized backup management across AWS environments. The service allows users to define backup plans with scheduled automated backups, retention policies, and lifecycle management rules. AWS Backup also integrates with AWS Organizations for managing backups across multiple accounts and regions. By offering a cost-effective and automated backup solution, AWS Backup enables businesses to ensure data protection while minimizing operational overhead.

**VPC**

**VPC Peering**: VPC Peering connects two Virtual Private Clouds (VPCs) to enable communication using private IP addresses. It allows resources in different VPCs to interact securely without requiring an internet gateway or public IPs. VPC Peering is a one-to-one connection, and the VPCs must be in the same region or in different regions with inter-region peering.

**VPC Endpoints**: VPC Endpoints allow private connections between a VPC and supported AWS services (like S3, DynamoDB) without traversing the public internet. They provide enhanced security by eliminating the need for an internet gateway or NAT devices. VPC endpoints include interface endpoints (for services with elastic network interfaces) and gateway endpoints (for S3 and DynamoDB).

**VPC Flow Logs**: VPC Flow Logs capture detailed records of traffic going to and from network interfaces in a VPC. This service helps with monitoring, troubleshooting, and auditing network traffic. Flow logs provide insights into traffic patterns, security analysis, and compliance verification.

**Site-to-Site VPN**: Site-to-Site VPN creates a secure connection between an on-premises network and an AWS VPC over the internet using IPsec tunnels. This enables hybrid cloud environments, allowing secure data transfer between on-premise data centers and AWS resources.

**Virtual Private Gateway & Customer Gateway**: A **Virtual Private Gateway (VGW)** is the AWS side of a Site-to-Site VPN connection, enabling communication between the VPC and on-premises networks. A **Customer Gateway (CGW)** represents the on-premises device or router that connects to the VGW. Together, they facilitate the secure connection for Site-to-Site VPN.

**Direct Connect & Direct Connect Gateway**: **AWS Direct Connect** establishes a dedicated, low-latency connection between an on-premises network and AWS, bypassing the internet for improved performance and security. **Direct Connect Gateway** allows for connections between AWS Direct Connect and multiple VPCs, enabling hybrid cloud architecture across regions.

**Direct Connect + Site-to-Site VPN**: Combining **Direct Connect** and **Site-to-Site VPN** provides a multi-layered approach to network connectivity. Direct Connect offers a reliable, private link, while Site-to-Site VPN serves as a backup over the internet, ensuring continuous connectivity even if the Direct Connect link goes down.

**Transit Gateway**: **AWS Transit Gateway** enables seamless communication between multiple VPCs, on-premises networks, and other AWS services. It acts as a hub that simplifies network architectures, reducing the need for complex peering arrangements by centralizing routing between connected networks.

**Egress Gateway**: An **Egress Gateway** enables VPCs to route outbound traffic from a private subnet to the internet in a secure and controlled manner. This is typically used to allow internet access from private resources while maintaining strict control over inbound traffic.