Kubernetes Interview Questions

Write down question along with answers on kubernetes for 5 years experienced devops engineer. Include scenario basd questions as well.

Here's a set of Kubernetes interview questions with answers tailored for a 5-years-experienced DevOps engineer. These include both theoretical and scenario-based questions:

Basic Questions

1. **What is Kubernetes, and why is it used?**
   Answer: Kubernetes is an open-source container orchestration platform that automates deploying, scaling, and managing containerized applications. It provides features like self-healing, automated rollouts and rollbacks, service discovery, and load balancing.
2. **What are Kubernetes Pods?**
   Answer: A Pod is the smallest deployable unit in Kubernetes. It encapsulates one or more containers, storage resources, a unique network IP, and specifications for how the containers should run.
3. **What are namespaces in Kubernetes?**
   Answer: Namespaces are used for logical partitioning of resources in a Kubernetes cluster. They help organize objects (e.g., Pods, Services) and are commonly used for environments (e.g., dev, staging, prod).

Intermediate Questions

4. **How does the Kubernetes scheduler work?**
   Answer: The scheduler identifies the best node to place a Pod based on resource requirements, constraints, and policies (e.g., affinity, taints, and tolerations). It ensures efficient resource utilization and balances workloads.
5. **What is a Deployment in Kubernetes?**
   Answer: A Deployment manages the desired state of an application. It automates the creation, scaling, and updates of Pods. Deployments can roll out updates incrementally or roll back to a previous version if needed.
6. **What is the role of etcd in a Kubernetes cluster?**
   Answer: `etcd` is a distributed key-value store used by Kubernetes to store cluster state and configuration data. It ensures consistency across the cluster.

Scenario-Based Questions

7. **Scenario**: A Pod is stuck in `Pending` state. How would you troubleshoot?
   Answer:
   - Check events using `kubectl describe pod <pod-name>`.
   - Verify if there are sufficient resources available on the nodes.
   - Check if the scheduler is working properly.
   - Review taints and tolerations, node affinity, and selector mismatches.
8. **Scenario**: Your application update caused downtime. How would you implement a strategy to prevent this in the future?
   Answer:
   - Use `RollingUpdate` in Deployments to gradually update Pods.
   - Leverage readiness probes to ensure Pods are ready before accepting traffic.
   - Perform canary deployments to test changes on a subset of users.
   - Use blue-green deployments for seamless transitions between versions.
9. **Scenario**: A node in the cluster is not scheduling any Pods. What steps would you take?
   Answer:
   - Verify if the node is in a `Ready` state using `kubectl get nodes`.
   - Check for taints using `kubectl describe node <node-name>`.
   - Confirm that there are no resource constraints (e.g., CPU, memory) or network issues.
   - Inspect kubelet logs for errors.

Advanced Questions

10. **What is a StatefulSet, and when would you use it?**
    Answer: StatefulSet manages stateful applications by ensuring ordered deployment, scaling, and updates. Use it for applications like databases or distributed systems where each Pod requires a unique identity.
11. **How do you secure a Kubernetes cluster?**
    Answer:
    - Enable Role-Based Access Control (RBAC).
    - Use namespaces to isolate resources.
    - Enable network policies to control traffic between Pods.
    - Ensure secrets are encrypted at rest.
    - Use Pod security policies or OPA Gatekeeper to enforce security rules.
12. **Explain the concept of Ingress in Kubernetes.**
    Answer: Ingress is an API object that provides HTTP and HTTPS routing to services in a cluster. It allows exposing multiple services under a single IP and enables functionalities like SSL termination and path-based routing.

Deep Scenario-Based Questions

13. **Scenario**: A Deployment is not scaling as expected, despite CPU utilization exceeding 80%. How do you resolve this?
    Answer:
    - Verify the Horizontal Pod Autoscaler (HPA) configuration using `kubectl describe hpa`.
    - Ensure metrics-server is running and providing resource metrics.

- Confirm resource requests and limits are defined for Pods.
- Check for API server throttling or network issues.

14. **Scenario**: A critical microservice experiences intermittent failures. Logs show connection timeouts. How would you debug?
    Answer:
    - Check service configuration and DNS resolution using `kubectl exec` and `nslookup`.
    - Verify network policies and firewall rules.
    - Use `kubectl logs` and `kubectl describe pod` for error details.
    - Use tools like `kubectl port-forward` or service mesh observability features to trace requests.
15. **Scenario**: You need to migrate a workload from one cluster to another without downtime. How would you approach it?
    Answer:
    - Use Kubernetes tools like Velero to back up and restore resources.
    - Configure cross-cluster load balancing (e.g., DNS updates).
    - Gradually migrate traffic using canary or blue-green strategies.
    - Use shared storage or replicate persistent data if needed.

Problem-Solving Scenarios

16. **Scenario**: PersistentVolumes are not binding to PersistentVolumeClaims. How do you fix this?
    Answer:
    - Verify if the storage class matches between the PV and PVC.
    - Check if the PV is in the `Available` state and has sufficient capacity.
    - Confirm the access modes (e.g., ReadWriteOnce).
    - Inspect events for errors using `kubectl describe`.
17. **Scenario**: A Pod keeps restarting. How would you debug?
    Answer:
    - Inspect logs using `kubectl logs <pod-name>`.
    - Check liveness and readiness probe configurations.
    - Verify resource limits (CPU, memory) for potential OOM kills.
    - Use `kubectl describe pod` to check for crash loops or events.

This mix of questions should help evaluate both technical skills and problem-solving abilities.

pod vs container

Pod vs Container in Kubernetes

Aspect	Pod	Container
Definition	A Pod is the smallest deployable unit in Kubernetes. It can encapsulate one or more containers.	A container is an isolated runtime environment for running applications (e.g., Docker container).

Aspect	Pod	Container
Purpose	Provides a higher abstraction over containers, adding networking, storage, and configuration.	Runs a single application or service with its own file system and dependencies.
Lifecycle Management	Kubernetes directly manages Pods, not individual containers.	Containers are managed within the Pod by Kubernetes.
Networking	Pods share the same network namespace, IP address, and port space.	Containers inside a Pod communicate with each other via `` `localhost` ``.
Storage	Pods can mount shared storage volumes accessible by all containers in the Pod.	Containers within a Pod access shared storage provided by the Pod's definition.
Use Case	Use Pods to group tightly coupled containers that need to share resources.	Use containers to encapsulate specific application functionality.
Example	A Pod running a web app container and a sidecar logging container.	A single Nginx container running in isolation.

Key Points

1. A Pod **contains one or more containers** and acts as a wrapper that manages the containers' lifecycle, networking, and storage.
2. Containers **cannot exist independently in Kubernetes**; they must always run inside a Pod.
3. When multiple containers run in a Pod, they share:
    - **Networking**: All containers can communicate over `` `localhost` `` and share the same IP.
    - **Storage**: Shared volumes defined at the Pod level.

Analogy

- A **Pod** is like a **house**, and the **containers** are the **rooms** within it.
    - The **house (Pod)** provides shared resources like a yard (network) and storage (volumes).
    - Each **room (container)** has a specific purpose but operates within the house.

This abstraction helps Kubernetes handle orchestration, scaling, and management more effectively!

Explain statefulset usecase with real-time industry example

StatefulSet Use Case with Real-Time Industry Example

What is a StatefulSet?

A **StatefulSet** is a Kubernetes controller designed to manage stateful applications. Unlike a Deployment, it ensures:

- Ordered deployment and scaling of Pods.
- Stable and persistent network identities for each Pod.
- Persistent storage tied to each Pod, ensuring data remains intact even if the Pod is rescheduled.

Key Features of StatefulSet

1. **Stable Network Identity**: Each Pod gets a unique hostname (e.g., `pod-name-0`, `pod-name-1`).
2. **Ordered Operations**: Pods are created, deleted, or updated in sequence.
3. **Persistent Storage**: Each Pod has its own PersistentVolume that isn't shared.

Real-Time Industry Example: Distributed Databases

Use Case: Running a Cassandra Cluster with StatefulSet

- **Industry Context**: Cassandra is a distributed NoSQL database widely used for applications requiring high availability, scalability, and fault tolerance (e.g., IoT platforms, e-commerce, banking systems).
- **Challenge**: Distributed databases require:
  - Stable hostnames for node-to-node communication.
  - Data persistence even after Pod restarts.
  - Proper ordering during scaling (adding/removing nodes).
- **Solution**: Use a StatefulSet.

Scenario Details:

- A company is managing a **global e-commerce platform** and uses Cassandra as the database to handle:
  - **Product catalog** across regions.
  - **User sessions** for a seamless shopping experience.
  - **Order management** with low-latency requirements.

How StatefulSet Helps:

1. **Stable Network Identity**:
   - Cassandra nodes need unique identifiers for cluster communication (e.g., `cassandra-0`, `cassandra-1`).
   - StatefulSet ensures each node gets a stable hostname.
2. **Persistent Storage**:
   - Each Cassandra node stores part of the database.
   - PersistentVolumes ensure that data on `cassandra-0` isn't lost, even if the Pod is restarted or moved to another node.
3. **Ordered Scaling**:
   - When adding new nodes, they must join the cluster in order and replicate data correctly.
   - StatefulSet ensures that Pods are created sequentially (e.g., `cassandra-3` only starts after `cassandra-2` is ready).
4. **Rolling Updates**:
   - During updates, StatefulSet ensures one node is updated at a time to avoid disrupting the entire cluster.

Kubernetes YAML Example:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      containers:
      - name: cassandra
        image: cassandra:3.11
        ports:
        - containerPort: 9042
        volumeMounts:
        - name: cassandra-data
          mountPath: /var/lib/cassandra
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Benefits Observed:

- **High Availability**: Cassandra ensures data replication across nodes, providing fault tolerance.
- **Data Persistence**: Pods retain their data across restarts.
- **Scalability**: Additional nodes can be added sequentially without impacting existing operations.
- **Seamless Updates**: Rolling updates ensure minimal downtime.

Other Real-Time Use Cases of StatefulSet

1. **Kafka Cluster**: StatefulSet is used for message brokers where each broker has unique data and identity.
2. **Zookeeper**: Manages distributed coordination systems requiring stable identities and order.
3. **Elasticsearch**: StatefulSet is ideal for search engines to handle data shards persistently.
4. **Redis Master-Slave Setup**: StatefulSet ensures stability for master and replica configurations.

Conclusion

StatefulSet is essential for deploying and managing stateful applications like databases or distributed systems where stability, persistence, and ordered operations are critical. For industries like **finance, retail, IoT, and SaaS**, it ensures robust application availability and reliability.

what is SSL termination?

What is SSL Termination?

SSL Termination refers to the process of decrypting Secure Sockets Layer (SSL)/TLS traffic at a server (often a load balancer or reverse proxy) and forwarding the unencrypted (plain HTTP) traffic to the backend servers. It simplifies encryption management by centralizing SSL/TLS operations at a single point.

How SSL Termination Works:

1. A client (e.g., a web browser) sends an encrypted HTTPS request to a server.
2. The server at the termination point decrypts the traffic using its SSL/TLS certificate and private key.
3. The decrypted request is forwarded to the backend servers over HTTP.
4. Responses from the backend are sent back to the termination point, where they are re-encrypted (if necessary) and sent to the client.

Common Locations for SSL Termination:

1. **Load Balancers (e.g., Nginx, HAProxy, AWS ELB)**:
   - SSL termination occurs at the load balancer, which forwards plain HTTP traffic to backend servers.
2. **Reverse Proxies (e.g., Nginx, Traefik)**:
   - These decrypt SSL traffic and route requests to backend services.
3. **API Gateways**:
   - SSL termination is performed for incoming API requests before forwarding to microservices.

Why Use SSL Termination?

1. **Performance Optimization**:
   - Decrypting SSL/TLS traffic is computationally expensive. By offloading this to a single termination point, backend servers can focus on processing requests.
2. **Simplified Certificate Management**:
   - SSL certificates are managed at a single location, reducing administrative overhead.
3. **Support for Legacy Applications**:
   - If backend applications don't support HTTPS, SSL termination can secure client communications while forwarding plain HTTP internally.
4. **Centralized Monitoring and Logging**:
   - All encrypted traffic is decrypted in one place, making it easier to inspect and monitor.

Real-Time Industry Example of SSL Termination

Scenario: A large e-commerce platform serving customers globally needs to secure customer transactions and optimize backend performance.

- **Solution**: An SSL-terminating load balancer (e.g., AWS Elastic Load Balancer or Nginx) is placed in front of backend servers.
  - The load balancer decrypts HTTPS requests (e.g., customer checkout data) at the edge.
  - Traffic is forwarded as HTTP to backend services for order processing.
  - This reduces the computational load on backend servers while ensuring customer data is encrypted during transmission.

SSL Termination vs SSL Passthrough

Aspect	SSL Termination	SSL Passthrough
Decryption Point	Traffic is decrypted at the termination point (e.g., load balancer).	Traffic is passed through fully encrypted to backend servers.
Backend Communication	Backend servers receive unencrypted HTTP traffic.	Backend servers handle SSL decryption themselves.
Use Case	Simplifies SSL management and improves performance.	Used when backend servers require end-to-end encryption.

Drawbacks of SSL Termination

1. **Security Concerns**:
   - Plain HTTP traffic between the termination point and backend servers could be intercepted unless within a secure, private network.
2. **Compliance Issues**:
   - Some regulations (e.g., HIPAA, PCI-DSS) may require end-to-end encryption, which SSL termination alone doesn't provide.

Mitigation for Security Risks

- Use **SSL/TLS re-encryption**: Re-encrypt traffic after termination before forwarding to the backend.
- Deploy SSL termination points within a **secure, private network**.

Conclusion

SSL termination is widely used for performance optimization, centralized certificate management, and simplifying HTTPS traffic handling in architectures like microservices, load-balanced applications, and high-traffic websites. However, it's important to evaluate security and compliance requirements when implementing it.

HPA vs VPA in k8s

HPA (Horizontal Pod Autoscaler) vs VPA (Vertical Pod Autoscaler) in Kubernetes

Aspect	HPA (Horizontal Pod Autoscaler)	VPA (Vertical Pod Autoscaler)
Definition	Automatically scales the number of Pods in a deployment or replica set based on resource usage.	Automatically adjusts resource requests and limits (CPU/memory) for individual Pods.
Scaling Type	Scales **horizontally** by increasing or decreasing the number of Pods.	Scales **vertically** by increasing or decreasing resource allocation (CPU/Memory) for existing Pods.
Focus Area	Handles traffic spikes by distributing the workload across more Pods.	Optimizes resource allocation for individual Pods to improve efficiency and reduce wastage.
Metrics Used	Typically uses CPU and memory usage metrics, or custom metrics via the Kubernetes Metrics API.	Observes historical usage patterns and adjusts resource requests accordingly.
Use Case	High traffic workloads where scaling the number of instances improves performance.	Applications where resource needs fluctuate significantly, or where resource limits are not optimized.
Deployment Requirement	Requires a metrics server and appropriate resource requests/limits in the Pod specs.	Requires the VPA admission controller and may require restarting Pods to apply changes.
Impact on Availability	Scaling is seamless, as new Pods are added or removed dynamically.	Pod restarts might be required to apply updated resource configurations.
Custom Metrics Support	Supports custom metrics using the Custom Metrics API.	Does not directly rely on custom metrics; uses historical data to adjust resources.
Examples	- Scaling a web application to handle more user traffic.	- Adjusting memory for a database service based on fluctuating query loads.

When to Use HPA?

- You want to handle variable workloads by **increasing/decreasing the number of Pods**.
- Your application is **stateless**, making horizontal scaling straightforward.
- Example: Scaling a web server or REST API under high traffic loads.

When to Use VPA?

- You want to ensure Pods are efficiently allocated **just enough CPU and memory** to function well without over-provisioning.
- Useful for **stateful applications** (e.g., databases) or workloads with predictable resource usage.
- Example: Optimizing the resource allocation of a MySQL database.

Can HPA and VPA Work Together?

Yes, **HPA and VPA can be used together** but require careful configuration to avoid conflicts:

- **VPA** adjusts the resource requests and limits for individual Pods.
- **HPA** uses those updated resource requests to decide how many Pods to scale.

However, **VPA is often used in "recommendation-only" mode** when combined with HPA to avoid frequent Pod restarts.

Example Scenarios

1. **HPA Example**: Scaling a front-end application:
   - A shopping website's frontend experiences spikes in traffic during a sale.
   - HPA adds more Pods to handle the load as CPU utilization increases.
2. **VPA Example**: Optimizing a database workload:
   - A backend database requires more memory during peak hours and less at other times.
   - VPA adjusts memory allocation dynamically to prevent out-of-memory errors and reduce waste.

Conclusion

- **HPA** is ideal for dynamically adjusting the number of Pods based on workload demands.
- **VPA** optimizes resource allocation for individual Pods, improving cost efficiency and reliability. For the best results, use them together but with proper configuration to balance horizontal and vertical scaling needs.

what is Velero how to use with exmaples

What is Velero?

Velero is an open-source tool used to **back up, restore, and migrate Kubernetes cluster resources and persistent volumes (PVs)**. It helps protect critical data and provides disaster recovery and migration capabilities for Kubernetes environments.

Key Features of Velero

1. **Backups**:
   - Back up Kubernetes cluster resources and Persistent Volumes.
   - Backup can be stored in object storage (e.g., S3, Azure Blob, GCS, MinIO).
2. **Restores**:
   - Restore cluster resources and Persistent Volumes from backups.
3. **Cluster Migration**:
   - Move resources and Persistent Volumes between clusters.
4. **Scheduled Backups**:
   - Automate backups with schedules.
5. **Selective Backups**:
   - Backup specific namespaces, resources, or labels.
6. **Disaster Recovery**:
   - Restore entire clusters in case of failures.

Velero Components

1. **Server**: Runs in the cluster and handles backup/restore operations.
2. **CLI**: A command-line tool for interacting with Velero.
3. **Backup Storage**: Object storage where backups are saved (e.g., AWS S3, Azure Blob, Google Cloud Storage).

How to Use Velero

1. Installation

Install Velero on your Kubernetes cluster using a compatible storage provider.

- **Example: Installing Velero with AWS S3 as a Backup Storage**

1. Install the CLI:

```
bash


  wget https://github.com/vmware-tanzu/velero/releases/latest/download/velero-linux-amd6
  4.tar.gz
  tar -xvf velero-linux-amd64.tar.gz
  sudo mv velero /usr/local/bin/
```

2. Set up AWS credentials: Create an IAM user with access to an S3 bucket and export the credentials:

```
bash


  export AWS_ACCESS_KEY_ID=<your-access-key>
  export AWS_SECRET_ACCESS_KEY=<your-secret-key>
```

3. Install Velero in the cluster:

```
bash


  velero install \
    --provider aws \
    --plugins velero/velero-plugin-for-aws:v1.8.0 \
    --bucket <bucket-name> \
    --secret-file ./credentials-velero \
    --backup-location-config region=<aws-region> \
    --snapshot-location-config region=<aws-region>
```

4. Verify installation:

```
bash


  kubectl get pods -n velero
```

---

2. Backing Up Resources

- **Backup all resources in the cluster**:

```
bash

  velero backup create cluster-backup
```

- **Backup a specific namespace**:

```
bash

  velero backup create my-namespace-backup --include-namespaces=my-namespace
```

- **Backup with specific labels**:

```
bash

  velero backup create label-backup --selector app=my-app
```

- **List existing backups**:

```
bash

  velero backup get
```

---

3. Restoring Resources

- **Restore a backup**:

```
bash

  velero restore create --from-backup cluster-backup
```

- **Restore a specific namespace**:

```
bash

  velero restore create namespace-restore --from-backup cluster-backup --include-namespac
  es=my-namespace
```

- **List restores**:

```bash
velero restore get
```

---

4. Scheduling Backups

- Schedule a daily backup of all resources:

```bash
velero create schedule daily-backup --schedule="0 2 * * *"
```

Real-World Example: Disaster Recovery

- **Scenario**: A production cluster has critical applications running, and an accidental deletion of resources or a hardware failure occurs.
- **Solution**:
  1. Use Velero to back up resources and persistent volumes daily to an S3 bucket.
  2. In case of a failure:
     - Spin up a new cluster.
     - Install Velero in the new cluster.
     - Restore resources and volumes using the latest backup.

Best Practices

1. **Storage Configuration**:
   - Ensure your object storage (e.g., S3) is secure and accessible.
2. **Regular Backups**:
   - Use scheduled backups to ensure data protection.
3. **Test Restores**:
   - Periodically test the restore process to verify data integrity.
4. **Namespace Segmentation**:
   - Use namespace-specific backups for targeted restores.
5. **Encryption**:
   - Encrypt backup data for security compliance.

Conclusion

Velero is a powerful tool for **backup, restore, and disaster recovery** in Kubernetes. Its ability to handle both resources and Persistent Volumes makes it a must-have for production-grade clusters, ensuring business continuity and operational reliability.

Explain ingress controller, ingress resource and load balancer

Ingress Controller, Ingress Resource, and Load Balancer in Kubernetes

These three components work together to route external traffic to services running inside a Kubernetes cluster. Let's break them down:

---

1. Ingress Controller

Definition:

An **Ingress Controller** is a specialized Kubernetes component responsible for implementing the rules defined in **Ingress Resources**. It acts as a reverse proxy and routes HTTP/S traffic into the cluster based on those rules.

Key Features:

- Manages external access to services inside the cluster using a single endpoint (e.g., a DNS name).
- Supports advanced features like SSL termination, URL path-based routing, and host-based routing.
- Examples of Ingress Controllers:
  - NGINX Ingress Controller
  - Traefik
  - HAProxy
  - AWS ALB Ingress Controller (for AWS environments)

How It Works:

1. The **Ingress Controller** runs as a Pod inside the Kubernetes cluster.
2. It monitors the API server for **Ingress Resources** and dynamically configures itself based on those rules.
3. It processes incoming traffic and directs it to the appropriate backend service.

---

2. Ingress Resource

Definition:

An **Ingress Resource** is a Kubernetes API object that defines **HTTP and HTTPS routing rules** for incoming traffic. It specifies:

- Which **hostnames** and **URLs** to expose.
- Which **services** inside the cluster should handle the traffic.

Key Features:

- Allows host-based and path-based routing.
- Can handle SSL/TLS certificates for secure communication.

- Works with an Ingress Controller to apply the routing rules.

Example of an Ingress Resource:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 80
  tls:
  - hosts:
    - example.com
    secretName: tls-secret
```

What This Does:

1. Routes `http://example.com/app1` traffic to `app1-service`.
2. Routes `http://example.com/app2` traffic to `app2-service`.
3. Secures traffic with an SSL certificate (`tls-secret`).

---

3. Load Balancer

Definition:

A **Load Balancer** is an external or internal resource used to distribute incoming traffic across multiple backend nodes, services, or Pods. It provides high availability and scalability.

Role in Kubernetes:

- In a Kubernetes setup, a **LoadBalancer Service** can be used to expose an application externally. However, **Load Balancers** typically operate at Layer 4 (TCP/UDP), while **Ingress Controllers** work at Layer 7 (HTTP/S).

Example:

- When deploying a Kubernetes cluster on a cloud provider (e.g., AWS, Azure, or GCP), a Load Balancer Service creates a cloud provider-managed load balancer (e.g., AWS ELB or Azure Load Balancer).

Key Features:

- Distributes traffic across cluster nodes.
- Can serve as the entry point for an **Ingress Controller**.

Example LoadBalancer Service:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

How They Work Together

1. **Load Balancer**:
   - Acts as the first point of entry into the cluster.
   - Forwards incoming traffic to the Ingress Controller Pods.
2. **Ingress Controller**:
   - Listens to traffic from the Load Balancer.
   - Processes the traffic based on rules defined in **Ingress Resources**.
3. **Ingress Resource**:
   - Defines routing rules that map external traffic (based on URL paths or hostnames) to Kubernetes services.

Example Workflow

Scenario: A company hosts two applications (`app1` and `app2`) and wants to expose them via a single domain (`example.com`).

1. **Ingress Resource**:
   - Defines rules to route `/app1` traffic to `app1-service` and `/app2` traffic to `app2-service`.
2. **Ingress Controller**:
   - Implements these rules and routes traffic appropriately.
   - Optionally terminates SSL (e.g., decrypt HTTPS traffic).

3. **Load Balancer**:
    - Receives external traffic and forwards it to the Ingress Controller.

Comparison Table

Component	Purpose	Layer
Load Balancer	Distributes incoming traffic to the cluster or Ingress Controller	Layer 4 (TCP/UDP)
Ingress Controller	Implements routing rules and directs traffic to services	Layer 7 (HTTP/HTTPS)
Ingress Resource	Defines HTTP/S routing rules for the Ingress Controller	Layer 7 (HTTP/HTTPS)

Conclusion

- Use a **Load Balancer** for basic traffic distribution or to expose an **Ingress Controller** externally.
- Use an **Ingress Controller** with an **Ingress Resource** for advanced Layer 7 HTTP/S traffic routing within the cluster.
- Together, these components provide a flexible, scalable, and secure way to handle external traffic in Kubernetes.