Container Orchestration

Container Orchestration Fundamentals

- **Need for Orchestration:**
  - Scaling applications.
  - Self-healing of workloads.
  - Load balancing.
  - Resource optimization.
- **Kubernetes as an Orchestration Tool:**
  - Managing containerized workloads.
  - Scheduling containers on nodes.
  - Ensuring desired state (via Deployments, ReplicaSets).
- **Core Orchestration Features in Kubernetes:**
  - Automatic scaling (Horizontal Pod Autoscaler).
  - Rolling updates and rollbacks.
  - Service discovery and networking.
  - Persistent storage management.
- **Workload Orchestration Concepts:**
  - Pods as the smallest deployable unit.
  - Running multiple containers in a Pod (e.g., sidecar pattern).

Runtime

1. **What is Container Runtime?**
   - A container runtime is software responsible for running containers, managing their lifecycle (start, stop, execute), and interacting with the operating system to create and manage containers.
2. **Kubernetes and Container Runtime:**
   - Kubernetes interacts with the container runtime through the **Container Runtime Interface (CRI)**.
   - The container runtime is responsible for pulling container images, creating containers, running them, and managing their lifecycle on a node.
3. **Popular Container Runtimes in Kubernetes:**
   - **Docker** (historically the most common, though deprecated in Kubernetes as of v1.20+ in favor of other runtimes).
   - **containerd**: A high-performance container runtime used by Kubernetes, focused on running containers.
   - **CRI-O**: A lightweight container runtime specifically built for Kubernetes, adhering strictly to the Kubernetes Container Runtime Interface (CRI).
   - **runc**: The low-level container runtime that creates and runs containers based on the OCI (Open Container Initiative) standards; often used by containerd and CRI-O.
4. **Container Runtime Interface (CRI):**
   - A Kubernetes API that allows the kubelet to communicate with various container runtimes.
   - Ensures that Kubernetes can support multiple runtimes (e.g., Docker, containerd, CRI-O) by abstracting runtime-specific details.
5. **Runtime Features:**
   - **Container Image Management:** Pulling, caching, and running images.
   - **Container Lifecycle Management:** Starting, stopping, and cleaning up containers.
   - **Namespaces & Cgroups:** Providing isolation for containers (ensuring they have their own process space, network, etc.).
6. **Runtime in Kubernetes Workflow:**
   - **Kubelet** requests container runtimes to start or stop containers on a node.
   - **PodSpec** in Kubernetes specifies what containers to run; the runtime manages the execution.
7. **Runtime Security Considerations:**

- Using **runtime security tools** to scan container images and ensure compliance with security standards.
- Enforcing security policies through runtime, such as restricting container privileges (user IDs, rootless containers).

8. **Transition from Docker to containerd/CRI-O:**
   - As of Kubernetes 1.20+, **Docker** is no longer the default container runtime.
   - Docker is replaced with **containerd** or **CRI-O** for better integration with Kubernetes.

In the **KCNA exam**, understanding how Kubernetes interacts with **container runtimes** and the different **runtime options** is essential, particularly focusing on how Kubernetes uses the **Container Runtime Interface (CRI)** to communicate with container runtimes like **containerd** or **CRI-O**.

Security

1. Container Security:

- **Image Scanning:**
  - Ensuring container images are free from vulnerabilities before deployment.
  - Tools like **Clair**, **Trivy**, or **Anchore** can scan for known vulnerabilities in container images.
- **Immutable Images:**
  - Using read-only images and containers to prevent modification at runtime.

2. Pod Security:

- **Pod Security Policies (PSP)** (deprecated in Kubernetes 1.21, but still relevant in older versions):
  - Define security controls for Pods (e.g., allow or disallow privileged containers, restrict running as root, enforce read-only file systems).
  - Controls what actions can be performed on Pods, such as running containers as root or accessing sensitive host files.
- **Pod Security Standards (PSS):**
  - New set of security controls for Pods replacing PSP, with three levels: **Privileged**, **Baseline**, and **Restricted**.
- **Admission Controllers:**
  - The **PodSecurityAdmission** (PSA) admission controller enforces the Pod Security Standards (PSS).
  - You can configure it at the namespace level with annotations that specify which PSS level (Privileged, Baseline, Restricted) is required.

3. Role-Based Access Control (RBAC):

- **Roles and RoleBindings:**
  - Define who can access Kubernetes resources and what operations they can perform.
  - Granular permissions on resources like Pods, Services, ConfigMaps, and more.
- **ClusterRoles and ClusterRoleBindings:**

- Used for setting permissions across the entire Kubernetes cluster.

4. Service Accounts and Identity Management:

- **Service Accounts:**
  - Kubernetes uses service accounts to grant applications running in Pods permissions to access Kubernetes API resources.
- **Identity and Access Management (IAM):**
  - Integration with external identity providers (e.g., AWS IAM, Azure AD) to control access to Kubernetes resources.

5. Network Security:

- **Network Policies:**
  - Define rules for controlling the communication between Pods (e.g., allow traffic only from specific Pods, block access from certain networks).
  - Ensure secure communication within the cluster.
- **TLS Encryption:**
  - Encrypting communication between services and between Pods using **Transport Layer Security (TLS)**.
- **Service Mesh (e.g., Istio):**
  - Provides secure communication between microservices by enforcing mTLS, along with fine-grained access control.

6. Secrets Management:

- **Kubernetes Secrets:**
  - Store sensitive information like API keys, tokens, passwords in Kubernetes resources.
  - Secrets can be mounted as volumes or injected as environment variables into containers.
- **Encryption at Rest:**
  - Ensuring that Secrets and sensitive data are encrypted when stored in etcd.
- **Third-Party Tools for Secret Management:**
  - Integration with tools like **HashiCorp Vault**, **SealedSecrets**, or **AWS Secrets Manager** for enhanced security.

7. Container Runtime Security:

- **Security Contexts:**
  - Define security settings for Pods and containers (e.g., setting user IDs, restricting privileges, setting capabilities).
- **RunAsUser & RunAsGroup:**
  - Define the user and group ID under which a container should run, limiting privileges.
- **Privilege Escalation:**
  - Prevent containerized applications from gaining escalated privileges (e.g., running as root).

8. Audit Logging:

- **Audit Logs:**
  - Kubernetes provides audit logs to record access and actions performed on the API server.
  - Helps with monitoring and detecting suspicious or unauthorized access.

9. Supply Chain Security:

- **Secure Supply Chain:**
  - Ensuring security in every step of the container supply chain, from image creation to deployment.
  - Using signed images (e.g., via **Notary** or **Cosign**) to ensure image integrity.

10. Vulnerability Management:

- **Kubernetes Security Contexts:**
  - Controls around running containers with restricted privileges (e.g., no access to the host network or storage).
- **Runtime Security (e.g., Falco):**
  - Tools like **Falco** that monitor container behavior during runtime to detect security incidents like unauthorized network access or privilege escalation.

Networking

Networking in Kubernetes (KCNA Relevant)

Networking is a core component of Kubernetes, enabling communication between Pods, Services, and external resources. Below are the relevant **Networking** topics for **Kubernetes** in the context of the **KCNA** exam:

---

1. Kubernetes Networking Basics:

- **Pod-to-Pod Communication:**
  - Pods can communicate with each other within a Kubernetes cluster using their IP addresses.
  - Kubernetes assigns each Pod a unique IP address, and Pods on different nodes can communicate with each other over the cluster network.
- **Flat Network Model:**
  - Kubernetes assumes that every Pod can communicate with every other Pod in the cluster without NAT (Network Address Translation).

2. Services in Kubernetes:

- **ClusterIP (default):**
  - Exposes a service on a cluster-internal IP address. This type of service is only accessible within the Kubernetes cluster.
- **NodePort:**
  - Exposes a service on a specific port on each Node's IP address. Allows external access to the service through `<NodeIP>:<NodePort>`.
- **LoadBalancer:**

    - Provisioned by cloud providers to expose services externally, typically using an external load balancer (e.g., AWS ELB, GCP Load Balancer).
- **ExternalName:**
    - Maps a service to an external DNS name, allowing Kubernetes to access external services by their DNS names.

3. DNS (Domain Name System):

- **CoreDNS:**
    - Kubernetes uses **CoreDNS** for service discovery. Each Service gets a DNS entry that can be accessed using its name within the cluster.
- **Service Discovery:**
    - Pods can access Services using DNS names (e.g., `my-service.my-namespace.svc.cluster.local`).

4. Network Policies:

- **Network Policies:**
    - Allows you to control the communication between Pods. You can define rules to allow or block traffic between Pods based on labels, IP blocks, or namespaces.
- **Ingress and Egress Rules:**
    - Ingress: Incoming traffic to Pods.
    - Egress: Outgoing traffic from Pods.
- **Pod Security:**
    - Control which Pods can communicate with others, enhancing network isolation and security.

5. Ingress and Egress Controllers:

- **Ingress Controller:**
    - Manages HTTP/HTTPS traffic into the cluster. It routes traffic based on domain name, paths, or other rules defined in the Ingress resource.
    - Popular Ingress controllers: **NGINX Ingress**, **Traefik**, **HAProxy**.
- **Egress Controllers:**

- Manage outbound traffic from the cluster to external services. Ensures control and security of traffic leaving the cluster.

6. CNI (Container Network Interface):

- **CNI Plugins:**
  - Kubernetes uses CNI plugins to manage networking for containers. Popular CNI plugins include **Flannel**, **Calico**, **Weave**, and **Cilium**.
- **Networking Model:**
  - The CNI ensures that Pods on different nodes can communicate using an overlay network or other networking strategies.
- **Network Overlay:**
  - Virtual networks that enable Pod-to-Pod communication across different physical machines or nodes.

7. Load Balancing:

- **Service Load Balancing:**
  - Kubernetes Services can automatically distribute traffic to Pods based on service type (e.g., ClusterIP, NodePort, LoadBalancer).
- **Ingress Load Balancing:**
  - Ingress Controllers handle the distribution of HTTP(S) traffic across multiple Pods, supporting features like SSL termination, routing, etc.

8. Network Security:

- **mTLS (Mutual TLS) with Service Mesh:**
  - Service meshes like **Istio** can be used to enforce mTLS for secure communication between microservices.
- **Network Isolation:**
  - Using Network Policies to isolate services and restrict communication between Pods.
  - Restrict which services can access certain Pods based on labels and namespaces.

9. External Connectivity:

- **Outbound Networking:**
  - Pods can access external services outside the cluster, managed through **egress rules** and **NAT** configurations.
- **External IPs:**
  - Assigning external IP addresses to services (e.g., using LoadBalancer services or NodePort for external access).

10. Troubleshooting Networking Issues:

- **kubectl commands** like `kubectl get pods -o wide`, `kubectl describe pod <pod-name>`, `kubectl logs <pod-name>`, and `kubectl exec` to troubleshoot Pod networking issues.
- **Network Diagnostics Tools** like **ping**, **traceroute**, and **curl** to test connectivity between Pods, Services, and external endpoints.

Service Mesh

Service Mesh in Kubernetes (KCNA Relevant)

A **Service Mesh** is a dedicated infrastructure layer that controls and manages the communication between microservices in a Kubernetes cluster. It provides features like traffic management, security, monitoring, and observability, all without requiring changes to application code.

Here are the key topics related to **Service Mesh** for the **Kubernetes and Cloud Native Associate (KCNA)** certification:

---

1. What is a Service Mesh?

- **Definition:** A service mesh is a network of microservices that work together to handle inter-service communications in a cloud-native application. It manages how services communicate with each other, providing features like load balancing, traffic routing, service discovery, observability, and security.
- **Decoupling Networking and Application Logic:**
  - Offloads network-related tasks (e.g., traffic management, security, monitoring) from the application code into a separate infrastructure layer.

2. Key Features of a Service Mesh:

- **Traffic Management:**
  - Routing, load balancing, and failure recovery between microservices.
  - Fine-grained traffic control (e.g., canary deployments, A/B testing, blue/green deployments).
- **Security:**
  - **mTLS (Mutual TLS):** Automatically encrypts and secures communication between microservices.
  - **Authentication and Authorization:** Ensures only authorized services can communicate with each other.

- **Observability:**
  - Monitoring and logging of microservices communication.
  - Distributed tracing and metrics collection (e.g., with Prometheus, Jaeger).
- **Service Discovery:**
  - Services in a mesh can discover each other through the service registry managed by the service mesh.
- **Fault Injection & Resilience:**
  - Control the behavior of services in the event of failure (e.g., retries, timeouts, circuit breaking).

3. Popular Service Mesh Implementations:

- **Istio:**
  - One of the most popular and widely used service meshes in Kubernetes.
  - Provides advanced traffic management, security, monitoring, and policy enforcement.
- **Linkerd:**
  - A lightweight, simpler service mesh compared to Istio, focusing on simplicity and ease of use.
- **Consul:**
  - A service mesh by HashiCorp that integrates with Kubernetes to manage service discovery, traffic routing, and security.
- **Kuma:**
  - A service mesh designed to be simple and flexible, built on top of Envoy proxies.

4. Service Mesh Architecture:

- **Data Plane:**
  - Consists of proxies deployed alongside services (often sidecars). These proxies handle the actual communication and enforcement of policies.
  - Example: **Envoy proxy** is commonly used as the sidecar proxy in service meshes.
- **Control Plane:**
  - Manages and configures the data plane, defining the rules and policies (e.g., routing, security) that the proxies will enforce.
  - Example: Istio's **Istiod** or Linkerd’s **control plane**.

5. Service Mesh and Kubernetes:

- **Integration with Kubernetes:**
  - Service meshes integrate directly with Kubernetes and take advantage of Kubernetes resources like Pods, Services, and Namespaces.
  - The mesh is often deployed as a set of controllers and sidecar proxies in Kubernetes clusters.
- **Sidecar Pattern:**
  - The service mesh relies on the **sidecar pattern**, where a proxy (such as Envoy) runs alongside each microservice to handle traffic management, security, and monitoring.

6. Benefits of Using a Service Mesh:

- **Simplified Service-to-Service Communication:**
  - Eliminates the need for manual configuration of networking features (like load balancing, retries, and circuit breaking) in each service.
- **Security:**
  - Enforces strong encryption (mTLS) for inter-service communication.
  - Provides centralized control for enforcing authentication and authorization policies.
- **Traffic Control:**
  - Provides fine-grained control over the traffic between microservices (e.g., routing, retries, circuit breaking, fault injection).
- **Observability:**
  - Provides deep insights into service communication, latency, error rates, and overall health of services.
  - Enables distributed tracing to track requests across multiple services.

7. Use Cases for Service Mesh:

- **Microservices Communication:**
  - A service mesh is ideal for managing complex microservice architectures, where multiple services need to communicate securely and reliably.
- **Traffic Management:**
  - Advanced traffic routing, blue/green deployments, and canary releases.
- **Service Discovery and Load Balancing:**
  - Helps microservices discover each other and load balance traffic efficiently.
- **Security and Compliance:**

- Ensures all communications are encrypted (mTLS) and enforces strict access control policies.

8. Challenges with Service Mesh:

- **Complexity:**
  - Service meshes can introduce additional complexity into the cluster due to their need for sidecar proxies and control plane components.
- **Overhead:**
  - Running sidecar proxies and maintaining a control plane adds overhead to the cluster.
- **Learning Curve:**
  - Kubernetes operators and developers may need to learn additional concepts related to service mesh configurations, policies, and monitoring.

Storage

Storage in Kubernetes (KCNA Relevant)

In Kubernetes, storage plays a crucial role in providing persistent storage solutions for applications running in containers. Unlike containers, which are ephemeral and can be destroyed and recreated, storage needs to persist across Pod restarts. Kubernetes provides a powerful system for managing and abstracting storage resources, allowing you to manage stateful applications effectively.

Here are the key **Storage** topics relevant to **Kubernetes** and the **KCNA exam**:

---

1. Persistent Storage in Kubernetes:

- **Persistent Volumes (PVs):**
  - A **Persistent Volume (PV)** is a piece of storage in the Kubernetes cluster that has been provisioned by an administrator or dynamically by a storage class.
  - **PV** is an abstraction of storage resources (e.g., network-attached storage, cloud storage).
  - PVs are independent of the lifecycle of Pods and are not tied to any specific Pod.
- **Persistent Volume Claims (PVCs):**
  - A **Persistent Volume Claim (PVC)** is a request for storage by a user or a Pod. It specifies the amount of storage and the access modes.
  - A PVC is used to claim a PV, and Kubernetes binds the claim to an available PV.
- **Storage Classes:**
  - A **StorageClass** defines a type of storage and how it should be provisioned (e.g., speed, replication, type of disk).
  - It allows dynamic provisioning of storage resources when a PVC is created.
  - **Examples:** `standard`, `fast`, `ssd`, or cloud-specific classes (e.g., `aws-ebs`).
- **Dynamic Provisioning:**
  - When a PVC is created, Kubernetes can automatically provision the corresponding PV based on the storage class.

    - If no PV exists that satisfies the PVC’s request, Kubernetes dynamically provisions a PV.
- **Access Modes:**
    - **ReadWriteOnce (RWO):** The volume can be mounted as read-write by a single node.
    - **ReadOnlyMany (ROX):** The volume can be mounted as read-only by many nodes.
    - **ReadWriteMany (RWX):** The volume can be mounted as read-write by many nodes.

---

2. Types of Storage in Kubernetes:

- **Ephemeral Storage:**
    - Temporary storage that is deleted when the Pod is terminated or deleted (e.g., emptyDir volumes).
    - Used for temporary data during the lifecycle of a Pod.
- **Persistent Storage:**
    - Storage that persists beyond the Pod lifecycle (e.g., databases, logs).
    - This includes PVs and PVCs, which provide durable storage that survives Pod restarts.
- **Volume Plugins (CSI):**
    - Kubernetes uses the **Container Storage Interface (CSI)** to support multiple storage providers (e.g., AWS EBS, GCE PD, NFS, GlusterFS, Ceph, etc.).
    - **CSI drivers** enable Kubernetes to work with external storage systems that support the CSI specification.
- **Common Volume Types:**
    - **emptyDir:** A temporary directory that is created when a Pod starts and is deleted when the Pod terminates.
    - **hostPath:** Mounts a file or directory from the host node’s filesystem into a Pod.
    - **nfs:** Mounts an NFS share into a Pod.
    - **awsElasticBlockStore (EBS):** Persistent block storage for AWS instances.
    - **gcePersistentDisk (GCE PD):** Persistent block storage for Google Cloud Engine.
    - **CephFS:** A shared file system that can be mounted on multiple Pods at once.

---

3. StatefulSets and Storage:

- **StatefulSets:**
  - **StatefulSets** are used for deploying stateful applications that require persistent storage.
  - StatefulSets automatically create and manage **Persistent Volume Claims (PVCs)** for each Pod in the set, ensuring each Pod has its own unique persistent storage.
  - The **PVC** created by a StatefulSet is bound to a **Persistent Volume (PV)** for each Pod, which persists across Pod restarts.
- **Stable Network Identity:**
  - Unlike regular Deployments, StatefulSets provide a stable network identity for each Pod (e.g., `my-app-0`, `my-app-1`), which can be useful when accessing persistent data.

---

4. Accessing Storage:

- **Volume Mounts:**
  - Volumes can be mounted inside Pods to be accessible by containers.
  - The storage provided by the PVs and PVCs can be mounted to Pods as filesystems or raw block devices, depending on the configuration.
- **Example of Mounting a PVC to a Pod:**

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: myapp
    image: myapp:latest
    volumeMounts:
    - mountPath: "/data"
      name: mydata-volume
  volumes:
  - name: mydata-volume
    persistentVolumeClaim:
      claimName: myclaim
```

5. Storage and High Availability:

- **Replication:**
  - Some storage solutions, like **NFS** or **Ceph**, provide replication features for high availability and fault tolerance.
  - Storage solutions can be configured to replicate data across multiple zones or nodes to ensure data redundancy.
- **Backup and Restore:**
  - Backup strategies should be in place for stateful applications, and solutions like **Velero** can be used for backup and restore of Kubernetes resources and persistent volumes.

---

6. Cloud-Native Storage:

- **Cloud Provider Storage Integration:**
  - Cloud platforms like **AWS**, **GCP**, **Azure**, and others offer cloud-native storage solutions that are integrated with Kubernetes.
  - These solutions include **block storage** (e.g., **AWS EBS**, **GCE PD**), **file storage** (e.g., **Azure Files**, **Google Cloud Filestore**), and **object storage** (e.g., **AWS S3**, **Google Cloud Storage**).
- **Cloud Provider-Specific Storage Classes:**
  - Kubernetes can use cloud-specific storage classes for dynamic provisioning of cloud-native volumes (e.g., AWS EBS, Google Persistent Disk).

---

7. Persistent Storage Lifecycle:

- **Binding:**
  - The process of associating a PVC with an available PV. This happens automatically if dynamic provisioning is configured.
- **Reclaim Policy:**
  - The behavior of the PV when the PVC is deleted. Common policies:
    - **Retain:** The PV is not deleted and must be manually cleaned up.

- **Delete:** The PV is deleted automatically when the PVC is deleted.
- **Recycle:** The PV is cleaned and made available for reuse (deprecated in newer versions).

---

8. Troubleshooting Storage Issues:

- **kubectl describe pvc <claim-name>:**
  - Use this command to get detailed information about a PVC and its status.
- **kubectl describe pv <volume-name>:**
  - Use this to check the status and details of a Persistent Volume.
- **Pod logs:**
  - Check the logs of Pods using storage to diagnose mounting or access issues.

In the KCNA Exam:

In the **KCNA exam**, understanding **Kubernetes storage** fundamentals is crucial, especially how to:

- Create and use **Persistent Volumes (PVs)** and **Persistent Volume Claims (PVCs)**.
- Work with **Storage Classes** for dynamic provisioning.
- Handle **StatefulSets** and ensure persistent data in stateful applications.
- Recognize different **volume types** and their use cases.
- Manage **cloud-native storage solutions** and ensure the appropriate use of Kubernetes storage resources.

These topics are important to ensure that applications have the right kind of storage for their needs and are prepared for any potential data recovery, scaling, and performance needs in a Kubernetes