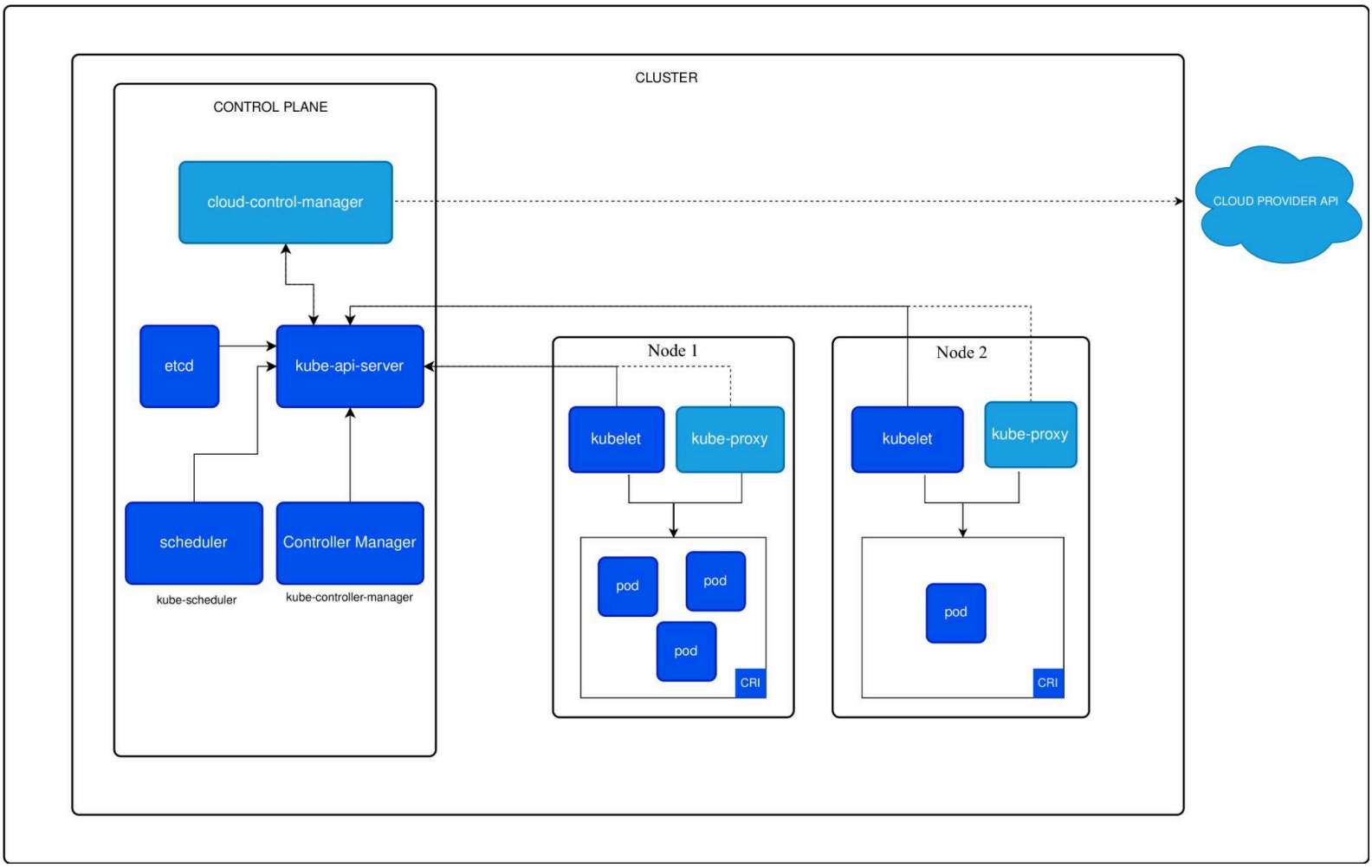
Kubernetes & Cloud Native Study Guide



Control Plane Components

Manage the overall state of the cluster:

kube-apiserver

The core component server that exposes the Kubernetes HTTP API

etcd

Consistent and highly-available key value store for all API server data In Kubernetes, **etcd** is the "source of truth," storing all cluster state and configuration data, including objects, secrets, and configuration settings.

kube-scheduler

Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node.

kube-controller-manager

Runs controllers to implement Kubernetes API behavior.

cloud-controller-manager (optional)

Integrates with the underlying cloud provider(s).

Node Components

Run on every node, maintaining running pods and providing the Kubernetes runtime environment:

kubelet

Ensures that Pods are running, including their containers, primarily responsible for ensuring that the containers are running as defined in the Pod specifications.

kube-proxy (optional)

Responsible for routing traffic for services and managing IP rules. Maintains network rules on nodes to implement Services.

Container runtime

Software responsible for running containers.

 This item links to a third-party project or product that is not part of Kubernetes itself. Your cluster may require additional software on each node; for example, you might also run systemd on a Linux node to supervise local components.

Add Ons

DNS

Addon for cluster-wide DNS resolution

Web UI (Dashboard)

Addon for cluster management via a web interface

Container Resource Monitoring

Addon for collecting and storing container metrics

Cluster-level Logging

Addon for saving container logs to a central log store

Open Container Initiative (OCI) Compliance

runC is the native runtime that is compliant with the Open Container Initiative (OCI) standards and serves as the reference implementation for its standards. It ensures container compatibility across various platforms.

Running Stateless Applications in Kubernetes

Deployment is the recommended API object for running scalable and stateless applications in a Kubernetes cluster. It manages the desired state of applications and handles updates and scaling efficiently.

CronJob Execution in Kubernetes

CronJob Controller: In Kubernetes, the CronJob controller is responsible for creating a Job at a scheduled time, which in turn creates a Pod to execute tasks. The Pod runs and completes its assigned tasks according to the schedule.

Kubelet's Role in Kubernetes

Kubelet is an essential Kubernetes component. It runs on each node within the cluster and ensures that containers are running inside Pods as expected.

Kubernetes Service Types

Ingress is not considered a service type. Kubernetes services include ClusterIP, NodePort, LoadBalancer, and ExternalName, which expose applications within or outside of the cluster.

Kubelet Communication Standard

Container Runtime Interface (CRI): The Kubelet uses the CRI to communicate with container runtimes like containerd or CRI-O.

Cgroups Functionality

Cgroups (Control Groups) allow for resource limiting within a system, ensuring that containers do not exceed their allocated CPU, memory, or other resources.

Scaling Applications in the Cloud

Horizontal Scaling is the most common method to scale cloud applications, where more replicas of an application or server are added to handle additional load.

Advantages of Cloud-Native Microservices

Microservices offer greater flexibility and scalability compared to monolithic applications, making it easier to scale and perform updates on specific components without affecting the entire system.

Vertical Scaling

Vertical Scaling involves allocating additional resources (such as memory or CPU) to existing

instances of an application to increase its capacity.

Kubernetes Control Plane Components

kube-proxy is not part of the control plane but operates as a network proxy on each node. The control plane components include the kube-apiserver, kube-scheduler, and etcd.

Factors in Node Scheduling

Kubernetes considers resource requirements when selecting a node to schedule a Pod. Factors like CPU, memory, and affinity rules are taken into account.

etcd Usage in Kubernetes

etcd is the backend object storage for Kubernetes' API, storing all cluster data, including configurations, secrets, and state information. In Kubernetes, **etcd** is the "source of truth," storing all cluster state and configuration data, including objects, secrets, and configuration settings.

Container Runtimes with Kubernetes

Kubernetes supports various container runtimes, such as CRI-O, containerd, and the now-deprecated Dockershim.

Creating Kubernetes Jobs

A CronJob resource is used to create Kubernetes Jobs at scheduled intervals, such as hourly or daily.

Characteristics of StatefulSets

StatefulSets offer ordered, graceful deployment and scaling, often used for stateful applications. They also rely on headless services to maintain stable network identities.

Adding New Resource Types

CustomResourceDefinitions (CRDs) allow users to extend the Kubernetes API by adding new resource types specific to their applications.

Horizontal Scaling

Horizontal Scaling is the process of adding additional replicas of an application or server to handle more traffic or load.

Editing Resources in Kubernetes

kubectl edit is the command used to modify resources directly on the Kubernetes server.

Serverless Computing Model

Serverless is a cloud computing model where infrastructure provisioning and management are abstracted away from the user. Developers focus solely on code.

Running Pods on Specific Nodes

A DaemonSet ensures that a Pod runs on all or some specific nodes, such as for monitoring or logging applications.

Kubernetes Pod Constructs

A Kubernetes Pod wraps one or more containers, making it the smallest deployable unit in Kubernetes.

Smallest Unit in Kubernetes

The Pod is the smallest possible unit in Kubernetes that can run a container.

Getting Documentation with kubectl

kubectl explain provides detailed documentation for a specified resource type directly from the command line.

Cluster Autoscaler

Automatically adjusts the number of nodes in a cluster based on resource demand.

Secrets Protection

Kubernetes stores secrets as Base64-encoded values, ensuring basic confidentiality. The data is not encrypted but is encoded using base64, which is a simple reversible encoding mechanism. Kubernetes stores Secrets in **etcd**, the cluster's key-value store, which is typically encrypted at rest when configured properly.

Kube-proxy

Manages network forwarding to the correct service endpoints in a Kubernetes cluster.

Dynamic Scheduling

Kubernetes schedules workloads dynamically based on resource availability and node characteristics.

Helm

A package manager for Kubernetes, Helm simplifies the deployment and management of applications.

GitOps Tools

Tools like Argo CD and Flux continuously compare the desired state in Git with the actual state in production, acting on differences.

Custom Resource Definition (CRD)

A way to extend the Kubernetes API by defining new resource types. CRDs allow the creation of custom resource objects specific to an application, extending Kubernetes functionality without altering the core API. They are managed using standard `kubectl`

commands.

Service Mesh Interface (SMI)

A common standard for service meshes, helping with communication between microservices. In the context of cloud-native applications, the **Service Mesh Interface (SMI)** is recognized as the standardized interface specification for service meshes. Key features of service meshes interfaces include:

- Provides a common API for service meshes on Kubernetes.
- Defines traffic management, security, and observability interfaces across different service mesh implementations.

Ingress

Manages external access to services within a Kubernetes cluster by exposing HTTP routes to services. Ingress routes external HTTP and HTTPS traffic to services within the cluster

Service Discovery

Helps applications and microservices locate each other on a network, crucial for cloud-native applications.

Kubernetes Architecture

Master Node Components:

- **kube-apiserver**: Manages API requests and serves as the gateway to the cluster.
- **etcd**: Stores the state of the Kubernetes cluster. In Kubernetes, **etcd** is the "source of truth," storing all cluster state and configuration data, including objects, secrets, and configuration settings.
- **kube-scheduler**: Decides which nodes run new Pods based on resource availability.
- **kube-controller-manager**: Manages the core controllers like Node Controller, Replication Controller, and others.

Worker Node Components:

- **kubelet**: Ensures that containers run in Pods.
- **kube-proxy**: Handles network proxy for Services, routing traffic to the appropriate container.
- **container runtime**: Software responsible for running containers, e.g., containerd, CRI-O.

Kubernetes Networking

Service Types:

- **ClusterIP**: Exposes services only within the cluster.
- **NodePort**: Exposes services on a static port on each node’s IP.

- **LoadBalancer**: Exposes services externally using a cloud provider's load balancer.
- **ExternalName**: Maps a service to an external DNS name.

Ingress: Provides external access to services via HTTP and HTTPS. It handles routing based on URL paths and hostnames.
Network Policies: Define how Pods can communicate with each other and other network endpoints.

Kubernetes Objects

Pod: The smallest deployable unit, which contains one or more containers.
Deployment: Manages stateless Pods, allowing for rolling updates and scaling.
ReplicaSet: Ensures a specific number of Pods are running at any given time.
DaemonSet: Ensures that a Pod runs on all (or some) nodes in the cluster.
StatefulSet: Manages stateful applications, ensuring Pods have stable identities.
Job/CronJob: Runs Pods to completion. CronJobs run on a scheduled basis.
PersistentVolume (PV) and **PersistentVolumeClaim (PVC)**: Abstract storage resources in Kubernetes.

Container Orchestration

Container Scheduling: Kubernetes uses the scheduler to decide on which node Pods should run based on resource requirements and constraints.

Affinity/Anti-Affinity: Defines rules to determine how Pods are distributed across nodes (e.g., placing certain Pods together or apart).

Taints and Tolerations: Used to prevent or allow Pods to be scheduled on certain nodes. Taints are set on nodes, and Pods must have a matching toleration to be scheduled on tainted nodes.

Cloud-Native Concepts

12-Factor App: A set of guidelines to help design scalable and maintainable cloud-native applications, including containerization, environment configuration, and dependency management.

Microservices: An architecture style where applications are composed of small, independent services that communicate over APIs.

Service Mesh: A dedicated infrastructure layer for managing microservice communication. Common service mesh tools include Istio and Linkerd. A service mesh typically includes:

1. **Proxy**: Sidecar proxies (e.g., Envoy) manage traffic for each service.
2. **Control Plane**: Configures and manages proxies, handling traffic policies and security (e.g., Istio).
3. **Data Plane**: Proxies handle network traffic between services.
4. **Security**: Ensures encryption (mTLS), authentication, and authorization.

5. **Observability**: Collects metrics, traces, and logs for monitoring.
6. **Traffic Management**: Manages load balancing, retries, and traffic routing.

DevOps: A cultural and technical practice that bridges development and operations teams to automate and streamline software delivery.

CI/CD: Continuous Integration (CI) and Continuous Delivery/Deployment (CD) are essential practices to frequently and reliably deliver code changes.

Infrastructure as Code (IaC): Managing infrastructure using machine-readable configuration files rather than physical hardware configuration.

Observability in Kubernetes

Logs: View logs of running Pods using kubectl logs. To view logs in real-time, use the -f option.
Metrics: Gather performance data using tools like Prometheus or the Kubernetes metrics server.

Monitoring: Tools like Grafana, Prometheus, and Jaeger are used to monitor and trace Kubernetes workloads.

Prometheus - An open-source monitoring solution that collects and stores metrics as time series data, featuring a powerful query language for data analysis. It is commonly used in conjunction with Grafana for visualizing the metrics Prometheus supports four metric types:

Counter: Cumulative, only increases (e.g., request counts).
Gauge: Can increase or decrease (e.g., memory usage).
Histogram: Tracks observations in buckets (e.g., request durations).
Summary: Similar to histogram, but also provides quantiles (e.g., median request duration).

Grafana - A versatile open-source analytics and visualization web application that supports various data sources, including Prometheus. It is renowned for its ability to create detailed dashboards that provide insights into metrics and logs

Fluentd -An open-source data collector designed for a unified logging layer, allowing the integration of data collection and consumption to enhance data understanding. Its vast plugin ecosystem supports data connections with multiple sources and destinations

KubeCost: Focuses on cost management for Kubernetes, enabling teams to monitor, analyze, and optimize their Kubernetes spending efficiently.

OpenTracing and **OpenTelemetry** primarily target the **application layer** of a software architecture for monitoring and observability.

- These tools focus on **distributed tracing**, **metrics**, and **logging** within applications, enabling the tracking of requests as they flow through different services and components in a microservices architecture.
- By instrumenting the application layer, they provide insights into performance, latency, and dependencies between services, helping developers monitor and troubleshoot complex, distributed systems.

Service Mesh

Service Meshes *offer a robust infrastructure layer that facilitates secure, reliable service-to-service communication within microservices architectures, enhancing observability, reliability, and security.*

Cilium: Utilises eBPF technology to provide advanced security features, network connectivity, and load balancing for Kubernetes clusters. It's integral for securing network communication and enforcing security policies

Linkerd: A lightweight, high-performance service mesh that simplifies service-to-service communication with built-in observability, tracing, and security, requiring minimal configuration

Istio: Offers a comprehensive service mesh solution, enabling fine-grained control over service communication with features like secure service-to-service communication, traffic management, and extensive observability. Istio uses Envoy as its default service proxy

Envoy: Designed for modern cloud-native applications, this edge and service proxy is the foundation for various service mesh implementations, including Istio, providing dynamic service discovery, load balancing, and TLS termination

Container Runtime and Execution

This section covers tools essential for container management, highlighting the distinction between container runtimes and Kubernetes management solutions.

Docker: The most popular container platform, providing the ability to package software into standardized units for development, shipment, and deployment

Kata Containers: Combines the speed of containers with the security of virtual machines, offering an isolated environment for container execution

containerd: An industry-standard runtime focused on simplicity, robustness, and portability, used by Docker and Kubernetes

CRI-O: Tailored for Kubernetes, providing a lightweight container runtime that fully complies with the Kubernetes Container Runtime Interface

runc: A CLI tool for spawning and running containers according to the Open Container Initiative (OCI) specification

gVisor: An open-source container runtime, designed to provide a sandboxed environment for running containers, enhancing security by isolating the workload from the host kernel

Kubernetes Management and Distributions

Tools and platforms designed to simplify the deployment, management, and operation of Kubernetes clusters.

k3s: A lightweight, easy-to-install Kubernetes distribution, ideal for edge computing, IoT, and CI/CD environments due to its minimal resource requirements

minikube: Enables the running of a Kubernetes cluster on personal computers, ideal for development and testing purposes

EKS (Amazon Elastic Kubernetes Service): A managed service that simplifies running Kubernetes on AWS and on-premises, providing seamless integration with AWS services

Continuous Integration/Continuous Deployment (CI/CD) & GitOps

These tools automate steps in the software delivery process, such as initiating automatic builds, tests, and deployments to streamline the development lifecycle.

Jenkins: A versatile open-source automation server, Jenkins facilitates building, testing, and deploying software, supporting a wide array of plugins for CI/CD purposes

Argo CD: Specialises in Kubernetes-native workflows, enabling continuous delivery and automating deployment pipelines directly from Git repositories

Argo Workflows: An extension of Argo as a workflow engine for orchestrating parallel jobs on Kubernetes

Flux: Embraces GitOps principles by automatically applying changes from Git to Kubernetes, ensuring that the state of clusters matches the configuration stored in version control

Security and Compliance

Tools designed to enforce security and compliance within Kubernetes environments, from deployment to runtime.

Falco: Detects abnormal application behavior and potential security threats in real-time by monitoring system calls and Kubernetes audit logs

Kubescape: Assesses Kubernetes clusters against known security standards and best practices, identifying compliance issues and vulnerabilities according to NSA and CISA guidelines.

OPA (Open Policy Agent): A versatile policy engine that enforces policies across the entire stack, ensuring compliance with security policies and regulations

4C's framework: In the context of Cloud Native Security, the correct order for the 4C's framework is:

- Cloud: Security begins with the cloud infrastructure, which provides the foundational security policies and controls.
- Clusters: Focuses on securing the Kubernetes or container orchestration clusters that run workloads.
- Containers: Involves securing the container runtime and ensuring that containers are isolated and protected.
- Code: Ensures the security of the application code, including secure coding practices and vulnerability management.

Storage in Kubernetes

Storage solutions in Kubernetes are essential for data persistence and management in containerized environments, supporting dynamic provisioning and scaling.

Volumes: Store data for containers, supporting ephemeral and persistent storage.

- **emptyDir**: Temporary storage that is erased when a Pod is deleted.
- **hostPath**: Mounts a directory from the host node into a Pod.
- **ConfigMap**: Provides configuration data for Pods.
- **Secret**: Stores sensitive data like passwords, API tokens, or SSH keys.

ETCD: Critical for Kubernetes, acting as the primary store for cluster state and metadata, ensuring high availability and consistency across a distributed environment

Rook: An open-source cloud-native storage orchestrator for Kubernetes, providing the platform, framework, and support for a diverse set of storage solutions to integrate with cloud-native environments natively

Ceph: A unified, distributed storage system designed for excellent performance, reliability,

and scalability

In Kubernetes, **Ephemeral Storage** is temporary storage tied to a Pod's lifecycle. It's used for non-persistent data, like logs or temporary files, and is deleted when the Pod is terminated.

In Kubernetes, persistent storage is added to a Pod by using a **PersistentVolume (PV)** and a **PersistentVolumeClaim (PVC)**. The PVC requests storage, and you attach it to the Pod by defining a volume in the Pod's YAML, which mounts the storage to the container. This storage persists across Pod restarts.

Serverless and Event-Driven Architectures

These tools help in building applications that can scale from zero to planet-scale without managing infrastructure, focusing on event-driven and serverless paradigms.

KEDA (Kubernetes Event-Driven Autoscaling): Dynamically scales Kubernetes applications in response to events from various sources, optimizing resource utilization

Knative: Enables the building and deployment of serverless and event-driven applications on Kubernetes, streamlining the development of cloud-native applications

Package Management and Application Deployment

Simplifying the deployment and management of applications in Kubernetes with tools that manage packages and dependencies.

Helm: The Kubernetes package manager, Helm streamlines the deployment of applications through Helm charts, which define, install, and upgrade even the most complex Kubernetes applications.

Kubernetes Best Practices

Namespace Usage: Use namespaces to logically separate resources and policies for different environments (e.g., dev, test, production).

Resource Requests and Limits: Define CPU and memory requests and limits to ensure that Pods get the resources they need while not over-provisioning.

Use of Labels and Selectors: Use labels to organize and select groups of objects in Kubernetes.

Configuration Management: Store configuration separately from application code using ConfigMaps and Secrets.

Security in Kubernetes

Role-Based Access Control (RBAC): Controls access to the Kubernetes API based on roles and permissions.

Pod Security Policies (PSP): Controls the security context in which Pods operate, including privileges and file system access.

Network Policies: Define which Pods can communicate with each other and with external endpoints.

Secrets Management: Use Kubernetes Secrets to store sensitive data, but consider using external tools like HashiCorp Vault for advanced security needs.

Image Security: Ensure containers are built from trusted sources and scanned for vulnerabilities (e.g., using tools like Clair or Trivy).

Kubernetes Troubleshooting

kubectl commands:

kubectl get – Lists resources such as Pods, Deployments, and Services.

kubectl describe – Provides detailed information about a resource.

kubectl logs – Fetches logs from a running container.

kubectl exec – Executes a command within a container.

kubectl top – Displays resource usage of nodes and Pods.

Common Errors:

ImagePullBackOff - Kubernetes is unable to pull the specified container image. This often happens due to an incorrect image name, missing image in the registry, or authentication issues with a private registry.

CrashLoopBackOff - The container in the pod repeatedly fails to start. This can be caused by application crashes, incorrect configuration, or missing dependencies.

ErrImagePull - Kubernetes failed to pull the container image from the registry. This is typically due to an incorrect image name, the image not existing, or a network issue.

ImageInspectError - Kubernetes cannot inspect the image after pulling it, often due to corruption or compatibility issues.

ErrImageNeverPull - The imagePullPolicy is set to 'Never', but the image isn't present on the node. Kubernetes is configured not to pull the image, but it cannot find it locally.

CreateContainerConfigError - Kubernetes fails to create the container configuration. This might be due to an invalid configuration, such as incorrect environment variables, volume mounts, or missing secrets.

CreateContainerError - The container fails to be created, often due to problems with the underlying Docker configuration or a missing image.

InvalidImageName - The container image name specified in the pod spec is invalid. This can occur due to typos or incorrect formatting.

NodeAffinity/PodAffinity/PodAntiAffinity conflicts - The pod scheduling fails due to conflicts in node or pod affinity/anti-affinity rules. This prevents the pod from being scheduled on any available node.

NodeNotReady - The node on which the pod is scheduled is not ready, possibly due to a problem with the node's configuration or connectivity.

Pending - The pod is stuck in a pending state, often because there are insufficient resources (CPU, memory) on the nodes, or no nodes match the pod's scheduling requirements.

ContainerCannotRun - The container starts but fails immediately. This might be due to an invalid command or entry point, incorrect environment variables, or missing dependencies within the container.

OOMKilled - The container was terminated by the Kubernetes Out of Memory (OOM) killer because it exceeded its memory limits.

DeadlineExceeded - The job or pod did not complete within the specified time limit. This is commonly seen with Kubernetes Jobs or CronJobs that have a time limit.

Evicted - The pod was evicted from the node due to resource constraints, such as disk pressure, memory pressure, or CPU limits.

Unschedulable - The pod cannot be scheduled on any node. This often occurs due to insufficient resources, taints, or node selector issues.

DNSResolutionError - The pod cannot resolve DNS names, often due to issues with the Kubernetes DNS service or network policies blocking DNS traffic.

VolumeMountError - Kubernetes cannot mount the volume specified in the pod spec, often due to incorrect configuration, missing PersistentVolumeClaims (PVCs), or permissions issues.

VolumeNotFound - The volume specified in the pod configuration cannot be found, often due to a missing PersistentVolume (PV) or PersistentVolumeClaim (PVC).

Readiness/Liveness Probe Failure - The pod fails to pass the readiness or liveness probes, causing Kubernetes to mark it as not ready or to restart it.

Unauthorized - Kubernetes cannot authenticate with the API server or a private container registry due to invalid credentials or expired tokens.

Forbidden - An action is blocked due to insufficient permissions, often related to Role-Based Access Control (RBAC) policies.

ConfigMapKeyRefError - Kubernetes cannot find a specific key in the ConfigMap referenced by the pod or deployment, leading to an error when the pod tries to start.

SecretKeyRefError - Similar to 'ConfigMapKeyRefError', this occurs when Kubernetes cannot find a specific key in the Secret referenced by the pod.

ServiceUnavailable - The Kubernetes service is not available, often due to issues with the underlying service pods or networking.

Pod CrashLoopBackOff - The Pod is crashing repeatedly; investigate logs using kubectl logs.

Pending Pods - Pods are unable to schedule, often due to insufficient resources or affinity rules.

Common and Important Kubernetes Commands

Basic Commands

kubectl version

Displays the version of both the client and server Kubernetes.

- *Example:*
  kubectl version

kubectl get [resource]

Lists one or more resources of a specific type, such as Pods, Deployments, or Services.

- *Example:*
  kubectl get pods
  kubectl get services
  kubectl get deployments

kubectl describe [resource] [name]

Shows detailed information about a specific resource.

- *Example:*
  kubectl describe pod [pod-name]
  kubectl describe service [service-name]

kubectl logs [pod-name]

Fetches the logs from a Pod. Add -f to follow the logs in real-time.

- *Example:*
  kubectl logs [pod-name]
  kubectl logs -f [pod-name]

The **kubectl logs** command with the **--previous** flag is used to view the logs of a terminated container in a multi-container pod in Kubernetes.

Example:

```
kubectl logs <pod-name> -c <container-name> --previous
```

The `--previous` flag can be abbreviated to `--p`

kubectl exec [pod-name] -- [command]

Runs a command inside a running container in a Pod.

- *Example:*
  kubectl exec [pod-name] -- [command]
  kubectl exec -it [pod-name] -- /bin/sh

kubectl delete [resource] [name]

Deletes a specific resource.

- *Example:*
  kubectl delete pod [pod-name]
  kubectl delete deployment [deployment-name]

Creating and Exposing Resources

kubectl create -f [file.yaml]
Creates a resource using a YAML or JSON configuration file.

- *Example:*
  kubectl create -f [file.yaml]

kubectl expose [resource] --port=[port] --target-port=[target-port]
Exposes a resource such as a Pod or Deployment as a service.

- *Example:*
  kubectl expose pod [pod-name] --port=80 --target-port=8080
  kubectl expose deployment [deployment-name] --type=NodePort --port=80

kubectl apply -f [file.yaml]
Applies a configuration to a resource by filename or stdin.

- *Example:*
  kubectl apply -f [file.yaml]

kubectl scale [resource] --replicas=[num]
Scales a Deployment or ReplicaSet to the desired number of replicas.

- *Example:*
  kubectl scale deployment [deployment-name] --replicas=5

Viewing Resource Status

kubectl get pods -o wide
Lists all Pods with additional details such as node, IP, and container status.

- *Example:*
  kubectl get pods -o wide

kubectl top nodes/pods
Displays resource (CPU/Memory) usage of nodes or Pods.

- *Example:*
  kubectl top nodes
  kubectl top pods

kubectl get events
Lists recent events in the cluster.

- *Example:*
  kubectl get events

Namespace Management

kubectl get namespaces
Lists all namespaces in the cluster.

- *Example:*
  kubectl get namespaces

kubectl create namespace [namespace]
Creates a new namespace.

- *Example:*
  kubectl create namespace [namespace]

kubectl delete namespace [namespace]
Deletes a namespace and all resources within it.

- *Example:*
  kubectl delete namespace [namespace]

kubectl config set-context --current --namespace=[namespace]
Switches the current context to the specified namespace.

- *Example:*
  kubectl config set-context --current --namespace=[namespace]

Editing and Patching Resources

kubectl edit [resource] [name]
Opens an editor to modify the configuration of a resource.

- *Example:*
  kubectl edit pod [pod-name]

kubectl patch [resource] [name] --patch [json|yaml]
Updates a resource using a patch.

- *Example:*
  kubectl patch deployment [deployment-name] --patch '{"spec":{"replicas":3}}'

Debugging and Troubleshooting

kubectl describe [resource] [name]
Provides detailed information about a resource, including recent events and state changes.

- *Example:*
  kubectl describe pod [pod-name]

kubectl get pod [pod-name] -o yaml
Outputs the full YAML configuration for a resource.

- *Example:*
  kubectl get pod [pod-name] -o yaml

kubectl port-forward [pod-name] [local-port]:[remote-port]
Forward a local port to a port on a Pod for debugging.

- *Example:*
  kubectl port-forward pod/[pod-name] 8080:80

Job and CronJob Management

kubectl create job [job-name] --image=[image]
Creates a job that runs a specific image.

- *Example:*
  kubectl create job [job-name] --image=busybox

kubectl get jobs
Lists all jobs.

- *Example:*
  kubectl get jobs

kubectl create cronjob [name] --image=[image] --schedule="[cron-schedule]"
Creates a CronJob that runs on a scheduled basis.

- *Example:*
  kubectl create cronjob [cronjob-name] --image=busybox --schedule="/5 * * * *"*

Service Account and Role-Based Access Control (RBAC)

kubectl create serviceaccount [account-name]
Creates a new service account.

- *Example:*
  kubectl create serviceaccount [account-name]

kubectl get serviceaccounts
Lists all service accounts.

- *Example:*
  kubectl get serviceaccounts

kubectl create rolebinding [binding-name] --clusterrole=[role] --serviceaccount=[namespace]:[account] --namespace=[namespace]
Binds a role to a service account in a specific namespace.

- *Example:*
  kubectl create rolebinding [binding-name] --clusterrole=view --serviceaccount=default:[account-name] --namespace=default

Context and Configuration Management

kubectl config view
Displays the current Kubernetes context configuration.

- *Example:*
  kubectl config view

kubectl config get-contexts
Lists all available contexts.

- *Example:*
  kubectl config get-contexts

kubectl config use-context [context-name]
Switches to a different Kubernetes context.

- *Example:*
  kubectl config use-context [context-name]

Certificates and Security:

- **TLS Certificates in Kubernetes:**
  - Kubernetes uses TLS to secure communication between components like the kube-apiserver, kubelets, and the etcd cluster.
  - Ensure familiarity with how certificates are generated, rotated, and managed in Kubernetes clusters, including managing certificate expiration.
- **Certificate Signing Requests (CSR):**
  - How Kubernetes handles CSRs and how administrators approve or deny them.
- **KubeConfig:**
  - The KubeConfig file structure and its importance in managing access to the cluster for different users and contexts.
  - Managing multiple clusters with KubeConfig.
- **Securing the API Server:**
  - Using admission controllers like `NodeRestriction`, `PodSecurityPolicies` (deprecated in 1.25, replaced by OPA or Gatekeeper), and `NetworkPolicies`.

Networking Additions:

- **CNI (Container Network Interface):**
  - The role of CNI plugins in Kubernetes networking.
  - Example plugins: Calico, Flannel, Weave Net, and Cilium.
  - Understanding CNI installation and troubleshooting networking issues related to CNI plugins.
- **CoreDNS Configuration:**
  - CoreDNS as the default DNS server for Kubernetes.
  - Configuring and troubleshooting DNS resolution issues in the cluster.
- **Network Policy Examples:**
  - Detailed examples of how to create `NetworkPolicies` to control ingress and egress traffic between Pods, namespaces, and external resources.

Advanced Scheduling:

- **Pod Affinity and Anti-Affinity:**
  - Detailed examples and use cases for `PodAffinity` and `PodAntiAffinity`.
- **Node Affinity and Taints/Tolerations:**
  - Deep dive into `NodeAffinity` rules and how to use `taints` and `tolerations` effectively for advanced scheduling scenarios.

- **Resource Requests vs. Limits:**
    - Clarify the difference between resource requests (guaranteed minimum) and resource limits (maximum allowed) and how Kubernetes schedules Pods based on these constraints.
    - Using tools like `kubectl top` to monitor resource consumption.

Kubernetes Upgrades:

- **Kubernetes Version Skew Policy:**
    - Understanding the version compatibility between Kubernetes components (e.g., kube-apiserver, kubelet, etcd) during upgrades.
- **Upgrading a Kubernetes Cluster:**
    - Best practices for performing rolling upgrades on clusters, focusing on control plane and worker node upgrades.
    - Tools like `kubeadm upgrade` for managing upgrades.
    - Handling deprecated features during upgrades.

High Availability (HA) Clusters:

- **Kubernetes HA Control Plane Setup:**
    - Setting up highly available Kubernetes control planes, including load balancers and etcd clustering.
    - HA kube-apiserver, etcd clustering, and load balancing across multiple control plane nodes.
- **Etcd Backup and Restore:**
    - Procedures for backing up and restoring etcd, the key-value store that holds the cluster state.
    - Example commands for backup: `etcdctl snapshot save` and restoring from a snapshot.

Disaster Recovery:

- **Disaster Recovery Strategies:**
    - Detailed backup strategies, including etcd snapshots, and how to restore a Kubernetes cluster from a disaster.
    - Restoring cluster state from etcd backups and dealing with lost control plane nodes.

Helm and Kubernetes Operators:

- **Helm Templating Basics:**
    - Detailed explanation of Helm chart templating, creating custom Helm charts, and using Helm

Certificates and Security

TLS Certificates in Kubernetes

- **TLS (Transport Layer Security)** ensures secure communication between Kubernetes components, such as the API server, kubelet, and etcd.
- Kubernetes components communicate over secure HTTPS, using TLS certificates.
- TLS certificates are typically auto-generated when using `kubeadm` or can be managed manually.
  Key components that use TLS certificates:
  - **kube-apiserver**: The API server must have a certificate to secure its communications.
  - **etcd**: Secures communication between etcd members and clients (such as the kube-apiserver).
  - **kubelet**: Each kubelet has a client certificate to authenticate itself to the kube-apiserver.
- **Certificate rotation**:
  - Kubernetes automatically handles the rotation of certificates using kube-controller-manager. Administrators must ensure that the certificate's expiry does not affect the cluster.

Command to check the expiration of Kubernetes certificates:

```
sudo openssl x509 -noout -enddate -in
/etc/kubernetes/pki/apiserver.crt
```

Certificate Signing Requests (CSR)

- Kubernetes can handle **Certificate Signing Requests (CSRs)**, where a kubelet or another component requests a certificate from the cluster.

Administrators can approve, deny, or manually sign CSRs using the `kubectl certificate` command.
Example commands:

```
kubectl get csr
kubectl certificate approve <csr-name>
```

KubeConfig

- **KubeConfig** is the configuration file that defines how to connect to Kubernetes clusters.
- It stores information such as clusters, users, and contexts to interact with multiple clusters.
  Key fields in the `~/.kube/config` file:
  - `clusters`: Defines cluster addresses and certificates.
  - `users`: Stores authentication information like tokens or certificates.
  - `contexts`: Associates users with clusters.

Switching contexts:
`kubectl config use-context <context-name>`

Securing the API Server

- The **API server** is the main entry point to the cluster and needs proper security controls.
- Key security measures:
  - **Role-Based Access Control (RBAC)**: Controls who can access what resources in the cluster.
  - **Admission Controllers**: Modules that govern the behavior of requests. Important ones:
    - `NodeRestriction`: Prevents nodes from modifying other nodes' resources.
    - `PodSecurityPolicies` (deprecated): Governs pod-level security settings.
    - Use **Open Policy Agent (OPA)** or **Kyverno** as replacements for PSPs.

Networking Additions

CNI (Container Network Interface)

- **CNI** plugins manage network configurations and connectivity between Pods in a Kubernetes cluster.
- Kubernetes supports several CNI plugins, each offering unique features:
    - **Calico**: Provides networking and network policies with optional BGP-based routing.
    - **Flannel**: A simple overlay network provider that assigns subnets to nodes.
    - **Weave Net**: A mesh-based network solution.
    - **Cilium**: Uses eBPF to provide advanced networking, security, and observability.
- Installing CNI plugins:
    - CNI plugin must be installed on each node to allow Pod communication.
    - CNI plugin manifests can be applied using `kubectl apply -f <plugin.yaml>`.

CoreDNS Configuration

- **CoreDNS** is the DNS service in Kubernetes, providing internal name resolution for services and Pods.

Configuring CoreDNS involves updating the **CoreDNS ConfigMap**, typically found in the `kube-system` namespace.
Example:

```
kubectl edit configmap coredns -n kube-system
```

- Common CoreDNS issues:
    - DNS name resolution issues may occur if CoreDNS is down or misconfigured.

Debugging DNS:

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
kubectl logs <coredns-pod> -n kube-system
```

Network Policies

- **Network Policies** define how Pods communicate with each other and external systems.

Example of a **NetworkPolicy** that only allows traffic to a Pod from Pods in the same namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector: {}
```

Advanced Scheduling

Pod Affinity and Anti-Affinity

- **Pod Affinity** ensures Pods are scheduled on nodes close to other specific Pods.

Pod Anti-Affinity ensures Pods are scheduled away from other specific Pods.
Example of **PodAffinity**:

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - frontend
      topologyKey: "kubernetes.io/hostname"
```

Node Affinity and Taints/Tolerations

Node Affinity ensures Pods are scheduled on nodes that match certain labels.
Example of **Node Affinity**:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "kubernetes.io/e2e-az-name"
          operator: In
          values: ["e2e-az1", "e2e-az2"]
```

Taints: Used to make a node "unavailable" unless the Pod has a matching toleration.
Taint a node:

```
kubectl taint nodes node1 key=value:NoSchedule
```

Kubernetes Upgrades

Version Skew Policy

- Kubernetes supports **version skew**, where kubelet can be one minor version behind the control plane.
- Ensure you upgrade control plane components first, followed by worker nodes.

Upgrading a Kubernetes Cluster

Upgrading a cluster managed by `kubeadm`:

```
kubeadm upgrade plan
kubeadm upgrade apply <version>
```

Upgrade kubelet and kubectl after upgrading control plane:

```
sudo apt-get update && sudo apt-get install -y kubelet kubectl
```

High Availability (HA) Clusters

Kubernetes HA Control Plane Setup

- An **HA control plane** includes multiple API servers, etcd instances, and a load balancer.
- You must set up a load balancer to distribute traffic among the API servers.

Etcd Backup and Restore

Backup etcd using `etcdctl`:

```
etcdctl snapshot save /path/to/backup.db
```

Restore etcd:

```
etcdctl snapshot restore /path/to/backup.db
```

Disaster Recovery

Disaster Recovery Strategies

- Always keep **etcd snapshots**.
- Keep cluster manifests backed up and use GitOps for infrastructure state.

Helm and Kubernetes Operators

Helm Templating Basics

- **Helm** is the package manager for Kubernetes. Helm charts define, install, and upgrade Kubernetes resources.

Example of a **Helm chart structure**:

```
mychart/
  Chart.yaml    # Metadata about the chart
  values.yaml   # Default values for templates
  templates/    # Kubernetes resources defined here
```

Kubernetes Operators

- **Operators** extend Kubernetes functionality by automating the lifecycle of applications using custom controllers.

Namespaces

- **Namespaces** provide a way to divide cluster resources between multiple users or teams. They help organize and manage Kubernetes objects such as Pods, Services, and Deployments in a logical manner.
- **Key Points:**
    - Every resource in Kubernetes (Pod, Service, etc.) belongs to a namespace.
    - By default, Kubernetes comes with four namespaces:
        - `default`: The default namespace for objects with no other namespace specified.
        - `kube-system`: Contains Kubernetes system components (e.g., kube-dns).
        - `kube-public`: Publicly accessible, typically used for resources that should be visible cluster-wide.
        - `kube-node-lease`: Used for node heartbeats.

Namespace Commands:

```
# List all namespaces
kubectl get namespaces

# Create a new namespace
kubectl create namespace <namespace-name>

# Delete a namespace
kubectl delete namespace <namespace-name>

# Switch to a specific namespace
kubectl config set-context --current --namespace=<namespace-name>
```

Objects

- **Kubernetes Objects** are persistent entities in the Kubernetes system. They represent the desired state of the cluster (e.g., running applications, workload configurations).
  Common objects:
  - **Pod**: The smallest deployable unit, representing one or more containers.
  - **Service**: Exposes an application as a network service.
  - **Deployment**: Manages stateless applications and handles updates and scaling.
  - **ReplicaSet**: Ensures a specified number of identical Pods are running at any given time.
  - **StatefulSet**: Manages stateful applications.
  - **Job**: Manages batch jobs, ensuring Pods run to completion.
  - **ConfigMap**: Holds configuration data as key-value pairs.
  - **Secret**: Holds sensitive data such as passwords or API keys.

Kubernetes API

- **Kubernetes API** is the central management interface to interact with Kubernetes components. It exposes all cluster operations via RESTful endpoints.
- **API Server** is the front-end for the control plane, which processes REST API requests.
- **Common API Resources**:
  - `/api/v1`: Accesses core Kubernetes resources (Pods, Services, Nodes, etc.).
  - `/apis/apps/v1`: Accesses additional resources (Deployments, StatefulSets, etc.).

Example API call using `curl`:

```
curl https://<api-server>/api/v1/namespaces/default/pods
```

  - Kubernetes API is used by both kubectl and internal components to interact with the cluster.

Role-Based Access Control (RBAC)

- **RBAC** is a mechanism in Kubernetes to control who can access what resources in the cluster.
- **Key Components:**
  - **Role**: Grants permissions within a specific namespace.
  - **ClusterRole**: Grants permissions cluster-wide.
  - **RoleBinding**: Associates a Role with users or service accounts within a namespace.
  - **ClusterRoleBinding**: Associates a ClusterRole with users across the entire cluster.

Example: Creating a Role and RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: "jane"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Pods

- A **Pod** is the smallest deployable unit in Kubernetes, representing a single instance of an application running in a container.
- **Key Concepts:**
    - A Pod can run one or more containers, which are tightly coupled and share the same network and storage.
    - Pods are ephemeral by nature, meaning they are created and destroyed dynamically.

Example Pod Configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Deployments

- A **Deployment** is a higher-level Kubernetes object used to manage stateless applications. It defines the desired state of your application (e.g., how many replicas of a Pod should be running).
- **Key Features:**
    - **Rolling Updates**: Ensures zero downtime when updating applications.
    - **Scaling**: Can scale Pods up or down by adjusting the number of replicas.
    - **Self-Healing**: Automatically replaces unhealthy Pods.

Example Deployment Configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

ReplicaSets

- A **ReplicaSet** ensures a specified number of Pod replicas are running at all times.
- **Key Features:**
    - Ensures the desired number of Pods are running by adding or removing Pods as needed.
    - **Deployments** use ReplicaSets internally to maintain Pod replicas.

Example ReplicaSet Configuration:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

Ingress

- **Ingress** is an API object that manages external access to services within a Kubernetes cluster, typically HTTP/HTTPS routes.
- **Key Features:**
  - Can define rules for routing external traffic to internal services.
  - Supports SSL/TLS termination for secure communications.

Example Ingress Configuration:

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

Vertical Autoscaling

- **Vertical Autoscaling** dynamically adjusts the resource requests and limits of Pods based on their usage.
- This is useful for applications that need more CPU or memory under load but don't require scaling in terms of the number of replicas.
- **Key Features:**
    - **Vertical Pod Autoscaler (VPA)** automatically adjusts the CPU and memory of Pods.
    - Helps improve resource utilization and prevent over-provisioning.

Cluster Autoscaling

- **Cluster Autoscaler** automatically adjusts the number of nodes in a cluster based on the resource demands of the workloads.
- **Key Features:**
    - Adds nodes when Pods cannot be scheduled due to insufficient resources.
    - Removes underutilized nodes when workloads decrease.
- **Enabling Cluster Autoscaler:**
    - Most managed Kubernetes services (e.g., AWS EKS, GCP GKE, Azure AKS) provide Cluster Autoscaler as a built-in feature.
    - It requires defining minimum and maximum node counts for the cluster.

CI/CD

- **CI/CD (Continuous Integration/Continuous Delivery)** refers to the process of automatically building, testing, and deploying code changes to a production environment.
- **Key Concepts:**
    - **Continuous Integration (CI)**: Developers frequently merge code changes into a shared repository, and automated tests are run to ensure quality.
    - **Continuous Delivery (CD)**: Automates the deployment of code changes to production or staging environments.
- Common tools:
    - **Jenkins**: An open-source automation server that supports building, testing, and deploying code.
    - **Argo CD**: A Kubernetes-native continuous delivery tool that follows the GitOps model.
    - **Flux**: Another Kubernetes-native continuous delivery solution.

12-Factor Methodology for Creating Cloud-Native Apps

The **12-Factor App** methodology is a set of best practices for building modern, scalable, and maintainable cloud-native applications.

1. **Codebase**: One codebase tracked in version control, many deploys.
2. **Dependencies**: Explicitly declare and isolate dependencies.
3. **Config**: Store configuration in the environment.
4. **Backing Services**: Treat backing services (like databases) as attached resources.
5. **Build, Release, Run**: Strictly separate build and run stages.
6. **Processes**: Execute the app as one or more stateless processes.
7. **Port Binding**: Export services via port binding
8. **Concurrency**: Scale out via the process model.
    - Applications should be designed to handle workload scaling by running multiple processes (instances) across containers or VMs. Kubernetes excels in managing this process through Pods and ReplicaSets.
9. **Disposability**: Maximize robustness with fast startup and graceful shutdown.
    - Applications should be designed to start quickly and shut down gracefully. Kubernetes helps with this via readiness and liveness probes to manage the lifecycle of containers.
10. **Dev/Prod Parity**: Keep development, staging, and production as similar as possible.

- By using containerized environments and orchestration tools like Kubernetes, teams can maintain parity between development, staging, and production environments, minimizing the "it works on my machine" issues.

11. **Logs**: Treat logs as event streams.

- Applications should not manage or store their own log files. Instead, logs should be output to stdout/stderr, where they can be captured and aggregated by centralized logging systems like **Fluentd**, **ELK (Elasticsearch, Logstash, Kibana)**, or **Prometheus/Grafana**.

12. **Admin Processes**: Run admin/management tasks as one-off processes.

- Administrative tasks like database migrations, backups, and data cleanup should be treated as short-lived, one-off tasks, often handled via Kubernetes Jobs or CronJobs.

Additional Concepts to Consider:

Service Mesh (Beyond 12-Factor Apps)

- A **service mesh** provides infrastructure-level support for managing service-to-service communications within a cloud-native microservices architecture.
- **Popular Tools**:
    - **Istio**: Provides traffic management, observability, security, and policy enforcement.
    - **Linkerd**: A lightweight, high-performance service mesh focused on simplicity and security.
    - **Consul**: A service mesh tool for service discovery, health checking, and traffic control.
- **Key Benefits of a Service Mesh:**
    - **Observability**: Provides metrics, logs, and tracing for services.
    - **Traffic Control**: Can apply rules for traffic routing, load balancing, and retries.
    - **Security**: Enforces mutual TLS (mTLS) between services, ensuring secure communication.

Cloud-Native Observability

- **Observability** refers to the ability to measure the internal states of a system by examining its outputs (metrics, logs, traces).
  Key Tools:
    - **Prometheus**: A monitoring tool that collects time-series data and exposes it via an API for alerting or visualizations.
    - **Grafana**: Works with Prometheus to create dashboards and visualize collected metrics.
    - **Jaeger**: A tool for distributed tracing to track the flow of requests in microservices architectures.
- **Observability Components in Kubernetes:**
    - **Logs**: Can be collected via tools like **Fluentd** or **Elastic Stack**.
    - **Metrics**: Tools like **Prometheus** and **Kubernetes Metrics Server** collect metrics for auto-scaling and monitoring.
    - **Tracing**: Distributed tracing tools like **Jaeger** or **Zipkin** provide insight into inter-service communications.

GitOps for Kubernetes

- **GitOps** is a set of practices that use Git as the single source of truth for managing Kubernetes clusters.
  Key Tools:
  1. **Argo CD**: A continuous delivery tool specifically designed for Kubernetes that follows the GitOps methodology.
  2. **Flux**: Another GitOps tool that automates deployments based on changes pushed to a Git repository.
- **GitOps Workflow**:
  1. Define the desired state of Kubernetes resources in a Git repository.
  2. When changes are made to the Git repository (such as updated configurations or new deployments), tools like Argo CD or Flux automatically apply the changes to the Kubernetes cluster.
  3. These tools continuously compare the actual state of the cluster with the desired state defined in Git, taking corrective actions if necessary.

Microservices vs. Monolithic Applications in Cloud-Native Development

- **Microservices Architecture**: Breaks down applications into small, loosely coupled, independently deployable services that communicate over APIs.
  Advantages of Microservices:
  - **Scalability**: Each service can be scaled independently.
  - **Resilience**: Failures in one service don't necessarily impact the whole system.
  - **Faster Deployment**: Each service can be developed, tested, and deployed independently, allowing for quicker release cycles.
- **Monolithic Architecture**: A single, large application where all components are tightly integrated.
  Challenges with Monolithic Architecture:
  - **Scalability**: Requires scaling the entire application even if only one part needs more resources.
  - **Deployment Complexity**: Small changes can require the entire application to be redeployed.
  - **Maintenance**: Becomes increasingly difficult to maintain and test as the application grows.
- **Kubernetes and Microservices**: Kubernetes is ideal for microservices architecture as it supports dynamic scaling, service discovery, and traffic management.

Serverless in Kubernetes

- **Serverless Computing** allows developers to build and run applications without managing infrastructure. In Kubernetes, tools like **Knative** or **KEDA** enable serverless functionality.
  Knative:
  - **Knative Serving**: Automatically scales applications based on traffic. Can scale down to zero when no traffic is present.
  - **Knative Eventing**: Handles event-driven architectures, allowing functions to trigger based on external events.
- **KEDA (Kubernetes Event-Driven Autoscaling)**:
  - Provides autoscaling for event-driven applications by automatically scaling resources in response to external events, such as messages in a queue or HTTP requests.
- **Key Features of Serverless**:
  - **Automatic Scaling**: Scales up and down based on demand.
  - **Pay-per-Use**: Charges based on actual resource usage (CPU, memory, execution time).
  - **Zero Management**: Developers focus on code, not the underlying infrastructure.

Pod Communication

In Kubernetes, Pod-to-Pod communication within the same node has the following characteristics:

1. **Direct Communication**:
   - Pods within the same node can communicate directly with each other using the internal IP addresses assigned by the cluster's network.
2. **Pod Network Overlay**:
   - Kubernetes uses a network overlay (depending on the network plugin, like Calico or Flannel) to ensure that Pods can communicate with each other, even within the same node.
3. **No NAT (Network Address Translation)**:
   - For Pod-to-Pod communication within the same node, Kubernetes typically does not require Network Address Translation (NAT). Pods communicate directly using their IP addresses without any translation.
4. **Same Network Namespace**:
   - Pods within the same node are part of the same network namespace, allowing them to reach each other directly through their assigned IPs without involving external load balancing or routing.
5. **Pod IPs are Unique Across the Cluster**:
   - Even though Pods are running on the same node, each Pod has a unique IP address within the cluster, ensuring clear, direct communication.
6. **Efficient and Fast Communication**:

- Since the communication occurs within the same node, it's generally faster and more efficient than inter-node communication because it avoids network hops between nodes.

Deploying Stateless Applications

In Kubernetes, the **Deployment** object is ideal for deploying stateless applications. It manages replica Pods, supports rolling updates for zero downtime, and offers self-healing and scalability. Deployments are perfect for stateless apps since they don't need persistent storage, and Pods can be easily replaced or scaled as needed.

Labels

In Kubernetes, **Labels** are key for managing costs by categorizing and organizing resources. You can assign labels to group resources by project, team, or environment, making tracking usage and allocating costs easier. Cloud providers widely support labels for billing and monitoring purposes.

The `kubectl logs` command with the `--previous` flag is used to view the logs of a terminated container in a multi-container pod in Kubernetes.

Example:

```
kubectl logs <pod-name> -c <container-name> --previous
```

This can be abbreviated to --p

Site Reliability Engineer

In a cloud-native environment, the Site Reliability Engineer (SRE) is typically responsible for managing Service Level Agreements (SLAs), Service Level Indicators (SLIs), and Service Level Objectives (SLOs).

Deployments and StatefulSets

In Kubernetes, the container specifications in Deployments and StatefulSets are similar in the following ways:

1. **Pod Template**: Both use a Pod template to define the container specifications, including the container image, resources, ports, environment variables, and volume mounts.
2. **Replica Management**: Both allow you to define the number of replicas (instances) for the Pods.

3. **Container Behavior**: Both manage containers within Pods the same way, ensuring that containers are running based on the specified configuration.

Pod Disruption Budget

A Pod Disruption Budget (PDB) in Kubernetes is a mechanism that ensures a minimum number of Pods remain available during voluntary disruptions, such as maintenance or updates. It limits how many Pods can be simultaneously evicted or disrupted.

- Prevent Downtime: PDB helps maintain application availability during node upgrades or Pod evictions.
- Min/Max Availability: You can specify the minimum number of Pods that must be available or the maximum number of Pods that can be disrupted at once.
- Voluntary Disruptions: This applies to voluntary actions like draining nodes, not to involuntary disruptions like crashes.

Sidecar Pattern

In Kubernetes, the **sidecar pattern** involves co-locating containers in a Pod to provide complementary functionality, like logging, monitoring, proxying, or security. Sidecar containers extend or support the main container by managing tasks such as traffic routing, data processing, or observability.

Split Brain & Leader Election

Split-brain occurs in distributed systems when network issues cause multiple nodes to act as leaders, leading to conflicting actions and data inconsistencies. It disrupts system stability and consistency. **Leader Election** prevents conflicts like "split-brain" by ensuring only one instance of a distributed stateful application acts as the leader at a time.

Network Namespaces

In a Pod, containers usually share the network namespace. This allows containers in the same Pod to communicate with each other over localhost and share the same IP address and ports. They also typically share other namespaces like IPC (Inter-process Communication) and UTS (hostname).