CONCEPT

Concise Kubernetes

CONCEPT

PART 1

- Kubernetes - Why it's essential
- Pods, Nodes, and Clusters.
- Kubernetes Manages Containers
- Pods- The smallest deployable unit.
- Scaling & managing workloads.
- Namespaces - Organizing resources
- Services- Exposing applications.

PART 2

- Understanding Kubernetes YAML Files
- Deployment-manage app updates
- ReplicaSets - Ensure high availability
- StatefulSets - Manage stateful app
- DaemonSets - Running on each node
- Jobs & CronJobs - Run & scheduled jobs
- ConfigMaps - Manage app config.

PART 3

- Secrets
- Ingress Controller
- Storage in Kubernetes
- RBAC Security Access
- Network Policies
- Service Discovery
- Editing Pods And Deployment

PART 4

- Blue-Green Deployments
- Canary Deployments
- Monitoring & Logging
- Argo CD- GitOps made Simple
- Helm Charts - Simplifying deployments
- Security Best Practices
- Troubleshooting - Handling issues

What we Learn

Concise Kubernetes

FUNDAMENTALS

What we Learn

- Kubernetes - Why it's essential
- Pods, Nodes, and Clusters.
- Kubernetes Manages Containers
- Pods- The smallest deployable unit.
- Scaling & managing workloads.
- Namespaces - Organizing resources
- Services- Exposing applications.

Part 1

Concise Kubernetes

Why is Kubernetes Essential

☞ **What is Kubernetes**?
Kubernetes is an open-source container **orchestration** platform that automates deploying, scaling, and managing applications in containers.

History:
Kubernetes was initially developed by Google to solve the challenges of managing containerized applications at scale. Google had an internal system called **Borg** that handled container orchestration.- Project 7 🚀
When a group of engineers started to work on open-sourcing a more streamlined, scalable orchestration platform based on their learnings from Borg, they code-named it "Project 7.

Project 7 evolved into a robust and open-source-ready platform, it was officially named "Kubernetes."

☞ **Why is it Essential in Modern DevOps**?
1.) **Efficient Resource Usage** ⚙️: Kubernetes optimizes resource use, helping teams save on hardware and cloud costs.
2.) **Environment Consistency**: It provides a consistent platform for development, testing, and production, ensuring smooth deployments.
3.) **Scalability** 📈: Automatically scales applications up or down to handle load changes efficiently.
4.) **Self-Healing**: Detects and replaces failed containers, maintaining uptime.
5.) **Security & Compliance** 🔐: Kubernetes offers built-in security features, like Role-Based Access Control (RBAC) and Secrets management.

CONCEPT

Concise Kubernetes

Know your Kubernetes Components

Cluster:
A Kubernetes cluster is a collection of nodes (machines) where Kubernetes manages and orchestrates containerized applications. A cluster represents a Kubernetes deployment as a whole, combining both worker nodes (where applications run) and a control plane (which manages and monitors the cluster).

Control Plane:
The control plane is responsible for the cluster's overall management. The cluster's control plane continuously monitors the actual state of the applications and resources, adjusting to match the desired state. This "desired state" is defined by the configurations (usually in YAML files) you apply to Kubernetes.

Control Plane comprises of several components that work together to manage cluster operations:
➲ **API Server:** Acts as the main access point for Kubernetes, handling communication between the user and the control plane.
➲ **Controller Manager:** Manages controllers, which ensure that the actual state matches the desired state (like ensuring the right number of pods).
➲ **Scheduler:** Assigns new pods to nodes based on available resources and other factors.
➲ **etcd:** A distributed key-value store that holds configuration data, cluster state, and metadata. It is similar to a database. It has the actual status of the pod.

Nodes:
A node is a single machine (either a virtual machine or a physical server) that runs in the Kubernetes cluster and hosts Pods. Nodes are the workhorses of Kubernetes; they perform all the tasks needed to keep applications running.

Types of Nodes:
Master Node (Control Plane Node): Manages the cluster and runs control plane components (API server, etcd, scheduler, controller manager).
Worker Node: Executes containerized applications and manages Pods.

Node Components:
➲ **Kubelet:** An agent that runs on each node, communicates with the API server, and ensures that containers in Pods are running as expected.
➲ **Kube-proxy:** Manages network rules and facilitates network communication between services, ensuring seamless Pod-to-Pod and Pod-to-external-traffic connections.
➲ **Container Runtime:** The software responsible for running the containers on each node (Docker, containerd, etc.).

In Kubernetes, nodes can be added or removed based on workload demands. Adding more worker nodes allows the cluster to run more applications or manage higher workloads, enhancing scalability.

How Kubernetes Manages Containers

Kubernetes manages containers by a process called container **orchestration**.
Container orchestration is the process of managing and automating the deployment, scaling, networking, and operation of containers across multiple servers.

How Kubernetes Manages Containers:

Automates Deployment 🚀: When you want to deploy a containerized application, Kubernetes handles all the steps automatically. You just tell Kubernetes what you want, and it takes care of all details.

Scales Up and Down 📈📉: Kubernetes can automatically add or remove containers based on the current demand. If your app is getting a lot of traffic, Kubernetes can spin up more containers to handle it.

Self-Healing ❤️: If a container crashes or has an issue, Kubernetes automatically detects it and replaces it with a new one.

Networking & Load Balancing 🌐: Kubernetes manages network connections between containers so they can communicate. It also distributes user requests across containers, balancing the load to prevent any single container from being overwhelmed.

Declarative Management 📜: You define the “desired state” of your application (like how many containers should be running, what version, etc.) in a configuration file. Kubernetes continuously monitors and makes sure the current state matches this desired state.

Pod: The Fundamental Building Block

A Pod is the smallest and most basic deployable unit in Kubernetes. It represents a single instance of a running process in the Kubernetes cluster. While containers are at the heart of modern application deployment, Kubernetes introduces Pods to provide an abstraction that manages one or more containers cohesively.

Why does Kubernetes use Pods?

Kubernetes could manage containers directly, but Pods provide:

1. Logical Grouping Containers in a Pod form a logical unit of deployment and scaling.
2. Enhanced Collaboration Shared storage and networking simplify interactions between tightly coupled containers.
3. Unified Resource Management Kubernetes can allocate resources, handle health checks, and manage container lifecycles more effectively by grouping them into Pods.

Key Features of a Pod

🔪 **Encapsulation**: A Pod encapsulates one or more containers.
Shared resources for those containers, including:

- Networking: All containers in a Pod share the same network namespace.
- Storage: Volumes are shared among the containers within a Pod.

✒️ **Lifecycle Management:** Kubernetes manages Pods rather than individual containers. This ensures containers in a Pod are always co-located, co-scheduled, and run in a tightly coupled manner.
✒️**Atomic Deployment Unit:** If a Pod fails, Kubernetes does not repair it but replaces it with a new Pod instance. Pods are designed to be ephemeral.

Components of a Pod
➡ **Containers**: Most Pods run a single container, but you can run multiple containers in a single Pod if they are tightly coupled.
➡**Shared Network Namespace:** Containers in the same Pod share:

- IP Address: Assigned at the Pod level.
- Ports: Containers within a Pod communicate via localhost.

➡**Volumes:** Storage shared among containers in the Pod.

Pod Lifecycle

1. **Pending**: The Pod is created but not yet running. This happens while Kubernetes schedules the Pod to a node.
2. **Running**: The Pod is successfully scheduled and all containers are running or in the process of starting.
3. **Succeeded**: All containers in the Pod have terminated successfully (exit code 0).
4. **Failed**: At least one container in the Pod has terminated with a non-zero exit code.
5. **Unknown**: The state of the Pod cannot be determined.

Nodes and Clusters - Scaling and managing workloads

A **node** is a single machine (virtual or physical) in a Kubernetes cluster that runs the actual workloads.

Node contains:

- **Kubelet**: Ensures that containers defined in the pod spec are running.
- **Kube-proxy**: Manages networking and communication for the pods.
- **Container Runtime:** Software to run containers (e.g. Docker, containerd).

Nodes can be worker nodes or control-plane nodes.

A **cluster** is a set of nodes working together, managed by Kubernetes, to run containerized applications.

- It includes a control plane that orchestrates and a group of worker nodes that handle workloads.
- Clusters ensure high availability and fault tolerance by distributing workloads across multiple nodes.

Scaling in Kubernetes

Kubernetes supports two types of scaling:

1. **Horizontal Pod Autoscaling (HPA):** Dynamically adjusts the number of pod replicas for a deployment or replica set. It monitors metrics like CPU, memory, or custom application metrics.Adds or removes pod replicas based on thresholds.
2. **Node Scaling:** Adding or removing nodes to/from the cluster.

Managed manually or automatically using tools like Cluster Autoscaler.
Cluster Autoscaler integrates with cloud providers to add remove virtual machines dynamically.

Managing Workloads in Kubernetes

Workloads: Workloads are the applications or services running on Kubernetes.

It is defined in Kubernetes using manifests (YAML or JSON files).

Types of Workloads:

- **Deployments**: For stateless applications; supports scaling and updates.
- **StatefulSets**: For stateful applications that require unique identities and stable storage (e.g., databases).
- **DaemonSets**: Ensures a copy of a pod runs on every node (e.g., log collectors).
- **Jobs and CronJobs**: For running one-time or scheduled tasks.

Other Features:

- **Load Balancing:** Kubernetes ensures workloads are balanced across the cluster using Services and Ingress.
- M**onitoring and Logging:** Tools like Prometheus, Grafana, and ELK Stack (Elasticsearch, Logstash, Kibana) help monitor workloads and log activities.
- **High Availability and Resilience:** Kubernetes automatically restarts failed pods and reschedules them to healthy nodes.

Benefits of Nodes and Clusters:

- Efficient Resource Utilization:
- Fault Tolerance
- Dynamic Scaling
- Centralized Management

Namespace in Kubernetes

Namespaces are a way to organize and isolate resources within a cluster. They provide a mechanism for dividing cluster resources between multiple users or teams. Namespaces are like virtual clusters within a single physical Kubernetes cluster.

You can think of Namespaces as separate "rooms" within a "house" (the cluster), where each room contains its own set of furniture (resources) and can be managed independently.

☛ Key Features of Namespaces:

🔷 **Isolation of environments** for different teams or projects. Resources within one namespace do not interact directly with resources in another.
🔷 **Resource Quota Management:** It define limits (e.g., CPU, memory) for resources within a namespace to avoid resource contention.
🔷 **Access Control:** Kubernetes Role-Based Access Control (RBAC) can be applied at the namespace level to restrict access to specific users or teams.
🔷 **Ease of Management:** Namespace is a group that manage resources related to a particular application or environment (e.g., dev, staging, prod) more effectively.

☛ When to Use Namespaces?

- For large-scale clusters with multiple teams or projects.
- To separate environments such as development, staging, and production.
- When implementing multi-tenancy to serve multiple customers.
- To enforce resource quotas and policies.

CONCEPT

Concise Kubernetes

How Does a Namespace Organize Resources in Kubernetes?

♦ **Scoping Resources:** Resources like Pods, Services, ConfigMaps, and Secrets are created within a specific namespace. You can have a Pod named app in the **dev** namespace and another Pod with the same name in the **prod** namespace. Example:

kubectl get pods -n dev
kubectl get pods -n prod

♦ **Default Namespace:** If no namespace is specified, resources are created in the default namespace.

♦**System Namespaces:** Kubernetes reserves certain namespaces for system resources:

1. **kube-system:** For core Kubernetes components (e.g., the scheduler, controller manager).
2. **kube-public:** For publicly accessible resources.
3. **kube-node-lease**: For node heartbeat leases.

♦ **Resource Isolation:** Namespaces isolate resources logically. A Service in the dev namespace cannot directly resolve or access a Pod in the prod namespace without explicit configuration.

♦ **Cross-Namespace Operations:** Admin-level operations can view or manage resources across all namespaces using the --all-namespaces flag.
kubectl get pods --all-namespaces

Services in Kubernetes: Exposing to the world

Before getting started with Services in Kubernetes, the first question comes in our mind is "**Why do we need Services?**". Suppose we have a website where we have 3 pod replicas of front-end and 3 pod replicas of backend. These are the following scenarios we have to tackle:

1. How would the front-end pods be able to access the backend pods?
2. If the front-end pod wants to access the backend pod to which replica of the backend pod will the requests be redirected. Who makes this decision?
3. As the IP address of the pods can change, who keeps the track of the new IP addresses and inform this to the front-end pods?
4. As the containers inside the pods are deployed in a private internal network, which IP address will the users use to access the front-end pods?

To overcome the above mentioned cases the services object is created. Services enables **loose coupling** between microservices in our application. It enables communication between various components within and outside of the application.

In Kubernetes, Services are an abstraction that define a logical set of Pods and a policy for accessing them. Since Pods in Kubernetes are ephemeral (can be created and destroyed dynamically), their IP addresses are not static. This is where Services come in—to provide stable networking and access to the Pods.

Why Do We Need Services in Kubernetes?

- **Pods Are Ephemeral:** Pods are temporary and can be destroyed or recreated for reasons like scaling, updates, or failures. Each new Pod gets a different IP address, making direct communication with Pods unreliable.
- **Stable Communication:** Services provide a consistent way to access the Pods, regardless of changes in the underlying Pods or their IPs.
- **Load Balancing:** Services distribute network traffic among multiple Pods.
- **Discovery:** They simplify service discovery by acting as a single access point to a group of Pods.

Feature	ClusterIP	NodePort	LoadBalancer
Exposition	Exposes the Service on an internal IP in the cluster.	Exposing services to external clients	Exposing services to external clients
Cluster	This type makes the Service only reachable from within the cluster	A NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.	A LoadBalancer service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on
Accessibility	It is **default** service and Internal clients send requests to a stable internal IP address.	The service is accessible at the internal cluster IP-port, and also through a dedicated port on all nodes.	Clients connect to the service through the load balancer's IP.
Yaml Config	`type: ClusterIP`	`type: NodePort`	`type: LoadBalancer`
Port Range	Any public ip form Cluster	30000 - 32767	Any public ip form Cluster

How Services Work Internally?

Label Selector: The Service identifies the set of Pods it manages using label selectors.

Endpoints Object: Kubernetes creates an Endpoints object, which keeps track of the IPs of Pods matching the Service's selector.

Service Proxying: Kubernetes uses kube-proxy to handle traffic to the Service and forward it to the appropriate Pod.kube-proxy uses methods like iptables or IPVS for load balancing.

Key Concepts of Services

- **Stable Endpoint:** A Service gives a consistent IP address (called a ClusterIP) and DNS name that remains constant, even as the underlying Pods change.
- **Discovery and Load Balancing:** Services allow clients to discover and communicate with the right Pods. They automatically distribute traffic among the Pods using label selectors and load-balancing mechanisms.
- **Selector and Labels:** A Service uses selectors to identify which Pods it should target. Pods with labels that match the selector will automatically become part of the Service.

WORKING WITH OBJECTS

What we Learn

- Understanding Kubernetes YAML Files
- Deployment-manage app updates
- ReplicaSets - Ensure high availability
- StatefulSets - Manage stateful app
- DaemonSets - Running on each node
- Jobs & CronJobs - Run & scheduled jobs
- ConfigMaps - Manage app config.

Part 2

Concise Kubernetes

Understanding Kubernetes YAML Files

Kubernetes has become a leading container orchestration platform, offering scalability, resilience, and portability.

There are two different ways to configure all components in Kuberneetes - **Declarative** and **Imperative**. Declarative way brings manifest file in the discussion which is written in either JSON or in YAML.

So, in general, YAML files are a fundamental aspect of defining Kubernetes resources. Key components of a YAML file, namely apiVersion, kind, metadata, and spec. By understanding these key elements, you will gain insights into how to create and configure Kubernetes resources effectively.

✍️ **apiVersion :**
The apiVersion field in a Kubernetes YAML file specifies the version of the Kubernetes API that the resource adheres to. It ensures compatibility between the YAML file and the Kubernetes cluster.
☞ **Core API Group**: Includes fundamental resources.
- Pods: apiVersion: v1
- Services: apiVersion: v1
- ConfigMaps: apiVersion: v1
- Secrets: apiVersion: v1

☞ **Apps API Group** : Used for managing workloads.
- Deployments: apiVersion: apps/v1
- DaemonSets: apiVersion: apps/v1
- StatefulSets: apiVersion: apps/v1
- ReplicaSets: apiVersion: apps/v1

CONCEPT

Concise Kubernetes

✍️ **kind :**
The kind field defines the type of resource being created or modified. It determines how Kubernetes interprets and manages the resource.
☞ Each kind has a specific purpose. For instance:
- Pod: Represents a single or multiple containers.
- Service: Exposes a set of Pods as a network service.
- Deployment: Manages rolling updates for applications.

✍️ **metadata :**
The metadata field contains essential information about the resource, such as its name, labels, and annotations. It helps identify and organize resources within the cluster.

- **name**: Specifies the name of the resource, allowing it to be uniquely identified within its namespace.
- **labels**: Enables categorization and grouping of resources based on key-value pairs. Labels are widely used for selecting resources when using selectors or applying deployments.
- **annotations**: Provides additional information or metadata about the resource. Annotations are typically used for documentation purposes, tooling integrations, or adding custom metadata.

✍️ **spec** :
The spec field describes the desired state of the resource. It outlines the configuration details and behavior of the resource. The structure and content of the spec field vary depending on the resource kind.

Deployment - How to manage application updates

A Deployment provides replication functionality with the help of ReplicaSets and various additional capabilities like rolling out of changes and rollback changes.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Key Features of Deployments

1. **Declarative Management:** You describe the desired state in a YAML or JSON file. Kubernetes takes care of the changes.
2. **Rolling Updates:** Updates occur incrementally, ensuring zero downtime.
3. **Rollback Capability:** Kubernetes can revert to previous versions if an update fails.
4. **Version History:** Deployments maintain a history of ReplicaSets, enabling easier rollbacks.
5. **Scaling:** Deployments manage the number of replicas dynamically.

Steps to Manage Application Updates

1. **Define the Deployment:** A Deployment describes the application's desired state, including the image version, number of replicas, and update strategy.

2. **Apply Updates Using kubectl:** You can update the application by changing the image tag in the Deployment file and applying it

3. Monitor the Update Progress: Use the following commands to observe the update:
Check Deployment status: ***kubectl rollout status deployment/my-app***
View update history: ***kubectl rollout history deployment/my-app***

4. **Rollback if Needed:** If the update causes issues, rollback to the previous version:
kubectl rollout undo deployment/my-app
You can also specify a specific revision to rollback to:
kubectl rollout undo deployment/my-app --to-revision=2

5. **Blue-Green Deployment (Optional)**
Instead of a rolling update, you can create a new Deployment with the updated version and use a Service to switch traffic between the old and new versions.

6. Canary Deployment (Optional)
This approach deploys the new version to a small subset of users first, allowing testing in a production-like environment. Gradually, the new version replaces the old one.

Challenge	Solution
Application Downtime	Use rolling updates or blue-green deployments.
Failed Updates	Rollback using `kubectl rollout undo`.
Long Update Durations	Adjust `maxUnavailable` and `maxSurge` parameters for faster updates.
Traffic Routing Issues	Use Service objects or ingress controllers to manage traffic properly.

Replicaset: How does it ensure high avaialbility

A ReplicaSet is a Kubernetes resource that ensures a **specified number of identical pod** replicas are running at any given time.

ReplicaSet is a process that run multiple instances of Pods. It constantly monitors the status of Pods and if any one fails or terminates then it restores by creating new instance of Pod and by deleting old one.

Key Features

1. **Desired State Management:** Maintains the desired number of replicas.
2. **Automatic Recovery:** Recreates pods that are deleted or fail.
3. **Selectors**: Matches pods using label selectors to manage them.

ReplicationController

A ReplicationController is an older Kubernetes resource with a similar purpose to ReplicaSets. It ensures that a specified number of pod replicas are running at all times.

How They Ensure High Availability

1. Maintaining Desired Pod Count: ReplicaSet and ReplicationController continuously monitor the pod count. If a pod fails or is deleted, they create new pods to match the desired state.
2. Automatic Rescheduling: If a node fails, pods are recreated on healthy nodes, ensuring availability.
3. Workload Distribution: Pods are distributed across nodes to prevent single points of failure.
4. Health Checks: Integration with liveness and readiness probes ensures only healthy pods serve traffic.

Differences between ReplicaSet and ReplicationController:

➢ **Replicaset**:

- Selectors: Supports set-based selectors (more flexible).
- Use Case: Used with Deployments for modern applications.
- Efficiency: More advanced and flexible.

➢ **ReplicationController**

- Selectors: Supports only equality-based selectors.
- Use Case: Considered legacy, replaced by ReplicaSet.
- Efficiency: Limited to basic replication tasks.

- ReplicaSets are the preferred approach as part of Deployments for robust application lifecycle management.
- ReplicationControllers are rarely used now and have been largely replaced by ReplicaSets for advanced capabilities and flexibility.

StatefulSet in Kubernetes

A StatefulSet in Kubernetes is a resource designed to manage and deploy stateful applications— applications that require **persistent storage**, **stable** network identity, and **ordered deployment** or scaling.

Unlike Deployments, which handle stateless applications, StatefulSets ensure that each replica of an application has a unique identity and stable, consistent storage.

See, how we write YAML for a StatefulSet:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: web
spec:
 serviceName: "web-service"
 replicas: 3
 selector:
 matchLabels:
 app: web
 template:
 metadata:
 labels:
 app: web
 spec:
-----
-----
```

CONCEPT

Concise Kubernetes

Key Features of StatefulSets

♦ **Stable Network Identity:** Each pod in a StatefulSet gets a unique, persistent hostname that follows the pattern ***<statefulset-name>-<ordinal>.***

♦ **Stable Persistent Storage:** Each pod is associated with its own persistent volume (PV). Even if a pod is deleted or rescheduled, it retains its associated data by reattaching the same PV.

♦ **Ordered Deployment and Scaling:**

- Pods are created, updated, or deleted in a sequential order (e.g., Pod 0 before Pod 1).
- When scaling down, the highest-numbered pod is removed first.

♦ **Ordered Rolling Updates:** Updates to pods are performed in a controlled, sequential manner, ensuring minimal disruption to the application.

Feature	StatefulSet	Deployment
Pod Identity	Stable, unique for each pod	Dynamic, shared across replicas
Persistent Storage	Dedicated per pod	Shared or ephemeral storage
Scaling Behavior	Ordered scaling	All pods can scale simultaneously
Rolling Updates	Sequential updates	Parallel updates

DaemonSets: ensure Running a pod on each node

A DaemonSet is a Kubernetes resource that ensures a specific pod is running on all or selected nodes in a cluster. This is particularly useful for running background tasks, system monitoring, or log collection on every node.

Key Features of DaemonSets:

Pod Deployment to All Nodes:

- DaemonSets ensure that a copy of a pod runs on every eligible node in the cluster.
- When a new node is added, the DaemonSet automatically creates a pod on it.
- If a node is removed, the pod associated with it is also removed.

Selective Node Deployment:

- You can restrict DaemonSets to certain nodes using node selectors, node affinity, or taints and tolerations.

Automatic Updates:

- DaemonSets automatically maintain the desired pod specification on nodes. If you update the DaemonSet, the pods on the nodes will be updated accordingly.

How DaemonSets Ensure Running a Pod on Each Node:

Controller Mechanism: DaemonSet Controller monitors the cluster state. It ensures the desired number of pods (one per eligible node) is maintained by reconciling the cluster's actual state with the desired state defined in the DaemonSet.

Pod Scheduling:
When you create a DaemonSet, Kubernetes:
- Schedules the DaemonSet pod on all eligible nodes using the kube-scheduler.
- Ensures these pods run even if there are changes to the node pool, such as adding or removing nodes.

Node Addition or Removal:
New Node Added: When a new node is added, the DaemonSet controller detects it and schedules a pod on the node.
Node Removed: If a node is removed, the corresponding pod is deleted.

Immutable Management:
Each DaemonSet pod is managed as an individual unit, but Kubernetes ensures that every eligible node runs exactly one instance of the pod.

When to use DaemonSets:

- **Log Collection:** Running logging agents like Fluentd or Logstash on every node.
- **Monitoring:** Running system monitoring agents like Prometheus Node Exporter.
- **Networking:** Running network-related tools, such as a CNI plugin or a network proxy.

Jobs and CronJobs: help in Running jobs?

Jobs:
A Job is a Kubernetes resource that runs a specific task to completion. It's designed for one-time or on-demand tasks. Once the task is finished, the Job and its associated Pods are automatically cleaned up.

Key Characteristics of Jobs:

- **One-time execution:** Jobs are intended for tasks that need to be run once.
- **Parallelism:** Jobs can be configured to run multiple Pods concurrently to speed up processing.
- **Completion guarantee:** Kubernetes ensures that the Job completes successfully, even if Pods fail.
- **Automatic cleanup:** Once the task is finished, the Job and its Pods are removed.

Use Cases for Jobs:

- **Data processing:** Processing large datasets that require significant computational resources.
- **Batch jobs:** Running batch jobs like data imports, exports, or report generation.
- **One-time tasks:** Executing tasks that need to be run only once, such as database migrations or system updates.

CronJobs:
A CronJob is a Kubernetes resource that schedules Jobs to run periodically based on a specified schedule. It's like a time-based trigger that initiates Jobs at predefined intervals.

Key Characteristics of CronJobs:

- **Scheduled execution:** CronJobs define a schedule using a cron expression to determine when to run Jobs.
- **Recurring tasks:** They are ideal for tasks that need to be executed repeatedly, such as backups, log rotation, or data synchronization.
- **History:** CronJobs keep a history of past executions, allowing you to track Job completion and failures.

Use Cases for CronJobs:

- **Backup and restore:** Automating regular backups of databases, filesystems, or applications.
- **Log rotation:** Deleting old log files to save disk space.
- **Data synchronization:** Keeping data consistent across different systems or databases.
- **Monitoring and alerting:** Running scripts to monitor system health and send alerts.
- **Report generation:** Generating reports at regular intervals.

How Jobs and CronJobs Work Together

1. CronJob Scheduling: The CronJob controller monitors the cluster for CronJob objects. When a CronJob's schedule matches the current time, it creates a new Job.
2. Job Execution: The Job controller creates Pods to execute the task defined in the Job spec.
3. Task Completion: The Pods run the task and report their status to the Job controller.
4. Job Completion: Once all Pods associated with the Job complete successfully, the Job is considered finished.
5. Cleanup: The Job and its Pods are automatically deleted.

By effectively utilizing Jobs and CronJobs, you can automate routine tasks, optimize resource utilization, and ensure the reliability and efficiency of your Kubernetes applications.

Feature	Job	CronJob
Purpose	Executes tasks once or until complete.	Schedules recurring tasks.
Trigger	Manually or programmatically triggered.	Runs on a defined schedule (cron syntax).
Use Case	Database migration, file processing.	Nightly backups, scheduled reports.
Retries	Supports retries on failure.	Inherits retries from the Job template.

ConfigMaps - Managing application configurations

ConfigMaps in Kubernetes is a **key-value** store that allows you to manage application configuration data independently from the application code.

This separation of configuration from the application code adheres to the 12-factor app methodology and ensures flexibility and portability across environments (development, staging, production).

In Kubernetes, Configmap is an API object that is mainly used to store non-confidential data. The data that is stored in ConfigMap is stored as key-value pairs. **ConfigMaps** are configuration files that may be used by pods as command-line arguments, environment variables, or even as configuration files on a disc.

This feature allows us to decouple environment-specific configuration from our container images, after this is done, our applications are easily portable. The thing to be noted here is that ConfigMap does not provide any sort of secrecy or encryption, so it is advised to store non-confidential data only We can use secrets to store confidential data.

Working with Kubernetes ConfigMaps allows you to separate configuration details from containerized apps. ConfigMaps are used to hold non-sensitive configuration data that may be consumed as environment variables by containers or mounted as configuration files.

ConfigMaps - Managing application configurations

How ConfigMaps Help Manage Application Configurations:

✅ **Decouples Configuration from Code** ConfigMaps store configuration details outside of the application codebase, allowing you to:

- Update configurations without rebuilding or redeploying the application.
- Manage different configurations for different environments

✅ **Centralized Management** All configuration data can be stored in one place (ConfigMaps), making it easier to manage and update application settings.

✅ **Dynamic Updates** If a ConfigMap is mounted as a volume, changes to the ConfigMap automatically propagate to the pod's filesystem (though not all applications reload these changes dynamically without a restart).

✅ **Portability Across Environments** The same container image can be used across multiple environments by just updating the ConfigMap with environment-specific configurations.

✅ **Flexible Injection Methods** ConfigMaps provide multiple ways to inject configurations into your application:

- As environment variables.
- As files mounted into the container (via volumes).
- Through command-line arguments.

✅ **Simplifies Secret Management (with Limits)** ConfigMaps can hold non-sensitive application parameters, simplifying their management and avoiding exposure in application source code.

WORKING WITH OBJECTS

What we Learn

- Secrets
- Ingress Controller
- Storage in Kubernetes
- RBAC Security Access
- Network Policies
- Service Discovery
- Editing Pods And Deployment

Part 3

Concise Kubernetes

Secrets - Handling sensitive information securely

Every software application is guaranteed to have some secret data. This secret data can range from database credentials to TLS certificates or access tokens to establish secure connections.

The platform you build your application on should provide a secure means for managing this secret data. This is why Kubernetes provides an object called Secret to store sensitive data you might otherwise put in a Pod specification or your application container image.

What are Kubernetes Secrets?
A Secret is an object that contains a small amount of sensitive data such as login usernames and passwords, tokens, keys, etc. The primary purpose of Secrets is to reduce the risk of exposing sensitive data while deploying applications on Kubernetes.

Key points about Kubernetes secrets:

- You create Secrets outside of Pods — you create a Secret before any Pod can use it.
- When you create a Secret, it is stored inside the Kubernetes data store (i.e., an etcd database) on the Kubernetes Control Plane.
- When creating a Secret, you specify the data and/or stringData fields. The values for all the data field keys must be base64-encoded strings. Suppose you don't want to convert to base64. In that case, you can choose to specify the stringData field instead, which accepts arbitrary strings as values.

- When creating Secrets, you are limited to a size of 1MB per Secret. This is to discourage the creation of very large secrets that could exhaust the kube-apiserver and kubelet memory.
- Also, when creating Secrets, you can mark them as immutable with immutable: true. Preventing changes to the Secret data after creation. Marking a Secret as immutable protects from accidental or unwanted updates that could cause application outages.
- After creating a Secret, you inject it into a Pod either by mounting it as data volumes, exposing it as environment variables, or as imagePullSecrets. You will learn more about this later in this article.

Types: The following are several types of Kubernetes Secrets:

- **Opaque Secrets:** Opaque Secrets are used to store arbitrary user-defined data. Opaque is the default Secret type, meaning that when creating a Secret and you don't specify any type, the secret will be considered Opaque.
- **Service account token Secrets:** You use a Service account token Secret to store a token credential that identifies a service account. It is important to note that when using this Secret type, you must ensure that the kubernetes.io/service-account.name annotation is set to an existing service account name.
- **Docker config Secrets:** You use a Docker config secret to store the credentials for accessing a container image registry.

- **Basic authentication Secret:** You use this Secret type to store credentials needed for basic authentication. When using a basic authentication Secret, the data field must contain at least one of the following keys:

username: the user name for authentication
password: the password or token for authentication

- **SSH authentication secrets:** You use this Secret type to store data used in SSH authentication. When using an SSH authentication, you must specify a ssh-privatekey key-value pair in the data (or stringData) field as the SSH credential to use.
- **TLS secrets:** You use this Secret type to store a certificate and its associated key typically used for TLS. When using a TLS secret, you must provide the tls.key and the tls.crt key in the configuration's data (or stringData) field.
- **Bootstrap token Secrets:** You use this Secret type to store bootstrap token data during the node bootstrap process. You typically create a bootstrap token Secret in the kube-system namespace and named it in the form bootstrap-token-<token-id>.

Best Practices for Handling Secrets in Kubernetes

1. Encrypt Secrets at Rest: Configure encryption for etcd to secure Secrets in storage
2. Limit Access with RBAC: Restrict who can view or modify Secrets using strict RBAC policies.
3. Avoid Hardcoding Sensitive Data: Use Secrets to inject sensitive data dynamically.
4. Use Minimal Permissions: Only provide the necessary permissions required by pods or users.
5. Enable Automatic Secret Rotation: Automate Secret rotation using CI/CD pipelines or tools like SealedSecrets.
6. Use External Secrets Manager for Critical Workloads: For highly sensitive data, rely on an external Secret management solution.
7. Monitor and Audit: Continuously monitor for unauthorized access to Secrets and audit logs regularly.
8. Disable Secrets Access to Unauthorized Pods: Implement network policies to prevent accidental exposure.

Ingress Controllers - Managing external access

An Ingress Controller is a Kubernetes component responsible for managing external access to the services running inside a Kubernetes cluster. It works in conjunction with Kubernetes Ingress resources, which define rules for routing traffic to services.
While Kubernetes provides services like NodePort and LoadBalancer for exposing services, these methods have limitations in complex scenarios. Ingress Controllers overcome these limitations by providing advanced traffic management capabilities such as URL-based routing, SSL termination, and load balancing.

✅ Key Components

1. **Ingress Resource:** A Kubernetes object that defines HTTP/HTTPS routing rules
2. **Ingress Controller:** Actual implementation that reads the Ingress resources and configures the underlying load balancer, reverse proxy, or API gateway to enforce the routing rules.

✅ How Does an Ingress Controller Manage External Access?

❶ **Centralized Traffic Routing**

The Ingress Controller acts as a central point of entry for all external traffic to the cluster.
It uses rules defined in Ingress resources to determine how traffic is routed to backend services.

❷ URL-Based Routing

Enables routing based on:

- Hostnames (e.g., example.com, api.example.com).
- Paths (e.g., /api, /app).

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: example-ingress
spec:
 rules:
 - host: example.com
 http:
 paths:
 - path: /api
 pathType: Prefix
 backend:
 service:
 name: api-service
 port:
 number: 80
 - path: /app
 pathType: Prefix
 backend:
 service:
 name: app-service
 port:
 number: 80
```

❸ Load Balancing
Distributes incoming requests across multiple instances of backend services, ensuring high availability and better performance.
❹ Secure Connections (TLS/SSL Termination)
Supports HTTPS traffic by handling TLS termination.
Allows you to define certificates for secure communication.

```
spec:
 tls:
 - hosts:
 - example.com
 secretName: tls-secret
```

❺ Path Rewriting
Modifies request paths before forwarding them to backend services, if needed.

❻ Advanced Features
Authentication and Authorization.
Rate limiting to prevent abuse.
Web Application Firewall (WAF) for additional security.

Flow of Traffic
DNS: A domain name resolves to the Ingress Controller's public IP.
Ingress Controller: Receives incoming traffic and matches it against defined rules in Ingress resources.
Backend Service: Traffic is forwarded to the appropriate service and its pods.

Advantages

- Simplified Configuration: Single entry point for multiple services.
- Cost-Effective: Reduces dependency on multiple external LoadBalancers.
- Enhanced Flexibility: URL-based routing, SSL termination, and custom rules.
- Scalability: Works with Kubernetes' auto-scaling capabilities.

Ingress Controller vs Kubernetes services(NodePort and LoadBalancer)

Ingress Controllers and Kubernetes Services are both used to manage and expose network traffic to applications running inside a Kubernetes cluster. However, they serve different purposes and operate at different levels.

✅ **Ingress Controller:**

- Manages external HTTP/HTTPS traffic and routes it to the appropriate backend services.
- Focuses on layer 7 (HTTP/HTTPS) traffic and advanced routing rules (host-based, path-based).
- Reads Kubernetes Ingress resources to configure a load balancer or proxy server like NGINX, Traefik, etc.
- Advanced routing, SSL termination, and load balancing for HTTP/HTTPS traffic.
- Supports host-based and path-based routing (e.g., api.example.com, /app).
- Handles SSL/TLS termination and certificates at the ingress level.
- Provides intelligent load balancing at the HTTP/HTTPS level.

✅ **Kubernetes Services:**
- Provides networking and communication between pods, within the cluster, and optionally external traffic.
- Covers networking at both layer 4 (TCP/UDP) and optionally layer 7 for simple external exposure.
- Directly maps traffic to pods using mechanisms like ClusterIP, NodePort, and LoadBalancer.
- Internal service discovery, exposing services to external users (basic), and load balancing.
- Defined by a Service resource with types like ClusterIP, NodePort, or LoadBalancer.
- No routing logic; forwards all traffic to a backend service or pod.
- Basic layer 4 (TCP/UDP) load balancing across pods.

Use Kubernetes Services for:

- Internal communication between pods.
- Basic external traffic exposure using LoadBalancer or NodePort.

Use Ingress Controllers for:

- Advanced traffic routing needs like host-based or path-based rules.
- Applications requiring SSL/TLS termination and custom HTTP/HTTPS rules.

CONCEPT

Concise Kubernetes

Storage in Kubernetes - Persistent Volumes

In Kubernetes, storage refers to how data is stored, accessed, and managed across the cluster. Applications running in containers often require persistent or temporary storage to store data. Kubernetes provides various abstractions and mechanisms to manage storage efficiently, enabling applications to store and retrieve data seamlessly, even if the container is terminated or rescheduled.

Types of Storage in Kubernetes

- **Ephemeral Storage:**
  - Data exists only as long as the pod or container is running.
  - Suitable for caching, logs, or temporary data.
  - Example: emptyDir, configMap, secret.

- **Persistent Storage:**
  - Data persists beyond the lifecycle of a pod.
  - Ideal for databases or critical application data.
  - Managed using Persistent Volumes (PVs) and Persistent Volume Claims (PVCs).

Persistent Volumes (PVs)

A Persistent Volume (PV) is a storage resource in a Kubernetes cluster that provides a way to store data persistently. Unlike ephemeral storage, PVs are independent of pod lifecycles, ensuring data durability and availability.

Key Components of PVs

Persistent Volume (PV):

- A cluster-level resource that represents physical or virtual storage.
- Defined and provisioned by the administrator.
- Offers storage from a variety of sources (local disks, NFS, cloud storage, etc.).

Provisioning:

- **Static Provisioning:** The administrator manually creates PVs with specific storage configurations.
- **Dynamic Provisioning:** Kubernetes automatically provisions PVs using a storage class when a PVC is created.

Persistent Volume Claim (PVC):

- A request for storage made by a user or application.
- PVCs are bound to PVs to provide access to storage.
- Enables dynamic or static provisioning of storage.

Binding:

- PVCs are matched to PVs based on the requested size, access modes, and storage class.
- Once bound, a PV can be used exclusively by the PVC.

Using the Volume:

- Applications mount PVCs in pods to access the underlying PVs.
- The PV remains available even if the pod is deleted.

Reclaiming:
When a PVC is deleted, the reclaim policy of the PV determines the next step:

- Retain: Data remains intact for manual recovery.
- Recycle: Data is wiped, and the PV is made available again.
- Delete: The underlying storage is deleted.

AccessModes: Specifies how the volume of PV can be accessed:

1. ReadWriteOnce: Can be mounted as read-write by a single node.
2. ReadOnlyMany: Can be mounted as read-only by many nodes.
3. ReadWriteMany: Can be mounted as read-write by many nodes.

RBAC (Role-Based Access Control) - Securing access

Role-Based Access Control (RBAC) in Kubernetes is a mechanism that allows you to define and enforce permissions for users, groups, or service accounts to access and perform actions on Kubernetes resources. RBAC is a key feature for securing a Kubernetes cluster by ensuring that only authorized users can perform specific actions on resources.

Key Concepts of RBAC

Role:

- A role contains a set of rules that define the permissions (verbs like get, list, create, update, etc.) to access certain resources (like Pods, ConfigMaps, or Deployments) within a namespace.
- Roles are namespace-scoped.

ClusterRole:

- Similar to a Role but applies cluster-wide and can be used for resources outside a specific namespace (e.g., nodes, namespaces).

RoleBinding:

- Binds a Role to a user, group, or service account within a specific namespace.

ClusterRoleBinding:

- Binds a ClusterRole to a user, group, or service account across the entire cluster.

How RBAC Secures Access

Granular Permissions:
You can assign specific permissions for different users or applications, limiting their actions to only what's required for their role.

Principle of Least Privilege:
Ensures that entities are only granted the minimal access necessary to perform their functions, reducing the risk of accidental or malicious misuse.

Segregation of Duties:
By assigning different roles to different users or teams, RBAC prevents unauthorized access and ensures accountability.

Auditability:
RBAC configuration can be audited, providing visibility into who has access to what.

Dynamic Access Control:
As roles and responsibilities change, RBAC allows you to update permissions dynamically without disrupting the cluster.

Advantages of Using RBAC

- **Scalable**: Manage access control at scale by reusing roles.
- **Secure**: Reduces the attack surface by limiting permissions.
- **Flexible**: Accommodates complex access control requirements.

RBAC is an essential tool for enforcing security best practices in Kubernetes and should be tailored to the operational needs of your organization.

Network Policies - Controlling pod-to-pod communication

In Kubernetes, network policies are a crucial component for controlling and managing network traffic between pods. In this post, will assist you to walk through the core concept and the steps needed for implementation of network policies in the cluster and also, ensure you to have granular control over the communication within cluster.

By default, **all pods in a Kubernetes cluster can communicate with each other without any restrictions.** Network policies allow you to define rules that control both incoming and outgoing traffic to and from pods.

- **Ingress Policies:** Control incoming traffic to pods.
- **Egress Policies:** Control outgoing traffic from pods.

> Before you start, ensure that your Kubernetes cluster supports network policies. This typically involves having a network policy engine such as Calico, Cilium, or Azure Network Policy Manager installed and configured in your cluster.

CONCEPT

Deny All Ingress and Egress Policy

To start with, create a default policy that denies all ingress and egress traffic unless explicitly allowed. This policy ensures that no unintended communication can occur.

```
# Default deny all policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: default
spec:
  podSelector: {}
  policyTypes:
   - Ingress
   - Egress
```

- **ingress**: Each NetworkPolicy may include a list of allowed ingress rules. Each rule allows traffic which matches both the from and ports sections.
- **egress**: Each NetworkPolicy may include a list of allowed egress rules. Each rule allows traffic which matches both the to and ports sections.

Allow Specific Ingress Traffic

To allow specific ingress traffic, you need to create a network policy that defines which pods can communicate with each other.
Here's an example that allows ingress traffic to pods labeled app=backend from pods labeled app=frontend:

```
# Allow ingress from frontend to backend
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: allow-frontend-to-backend
namespace: default
spec:
podSelector:
matchLabels:
app: backend
policyTypes:
- Ingress
ingress:
- from:
- podSelector:
matchLabels:
app: frontend
```

Service Discovery - Connecting services within the cluster

Service discovery in Kubernetes is a mechanism that allows applications to find and communicate with each other without needing to hard code IP addresses or endpoint configuration.

An application deployment in Kubernetes consists of a pod or set of pods. These pods are **ephemeral**, which means their IP addresses and ports are continually changing.

In the Kubernetes world, this continual change makes service discovery a huge difficulty.

What is Service Discovery?
Service discovery is a mechanism by which services discover each other dynamically without the need for hard coding IP addresses or endpoint configuration.

In modern cloud-native infrastructure such as Kubernetes, applications are designed using microservices. The different components need to communicate within a microservices architecture for applications to function, but individual IP addresses and endpoints change dynamically.

As a result, there is a need for service discovery so services can automatically discover each other.

Key Components of Service Discovery
Kubernetes provides built-in support for service discovery through the use of Services and DNS.

When a service is created, Kubernetes assigns it a stable DNS name and a virtual IP address (ClusterIP). This allows workloads to connect to the service using either its DNS name or IP address, even if the underlying pods' IPs change.

1. **Services**: Abstract a set of pods and provide a single point of access.Example types:ClusterIP: Internal access within the cluster.NodePort: Exposes the service on a static port on each node.LoadBalancer: Exposes the service externally using a cloud provider's load balancer.
2. **Kube-DNS/ CoreDNS:** Kubernetes automatically creates DNS records for services.Example: A service named my-service in the namespace default can be resolved as my-service.default.svc.cluster.local.
3. **Endpoints**: Tracks the IPs of the pods backing a service. Kubernetes automatically updates these endpoints as pods come and go.

How service discovery works in Kubernetes:
There are two different ways of Service discovery in Kubernetes:
A.) for API-aware clients
An application deployment consists of set of pods. These pods are ephemeral, which means their IP addresses and ports are continually changing. In the Kubernetes, this change makes service discovery a huge difficulty.

CONCEPT

Concise Kubernetes

Kubernetes' endpoints API is one method it supports service discovery. Client applications can use the endpoints API to discover the IP addresses and ports of pods in an application.

The Kubernetes control plane ETCD serves as a service registry, where all endpoints are registered and kept up to date by Kubernetes.

B.) Client having no API support:

Not all clients support APIs, Kubernetes supports service discovery in other methods also.

A **Kubernetes service** object is a persistent endpoint that points to a collection of pods depending on label selectors. It uses labels and selectors to route requests to the backend pods.

Because pods can come and leave dynamically in Kubernetes, a service object ensures that the endpoint or IP address that points to the list of operating pods never changes. If numerous pods are operating in the same application, the requests are also load-balanced across a group of pods.

Clients can utilize the Kubernetes service's DNS name. Kubernetes' internal DNS manages service mapping.

The usage of DNS for name-to-IP mapping is optional, and Kubernetes can do so with environment variables. The fundamental implementation of Kubernetes Service is handled by a kube-proxy instance running on each worker node.

Editing Pods and Deployments

You CANNOT edit specifications of an existing POD other than the below:

1. spec.containers[*].image
2. spec.initContainers[*].image
3. spec.activeDeadlineSeconds
4. spec.tolerations

For example you cannot edit the environment variables, service accounts, resource limits of a running pod.

But if you really want to, you have 2 options:

1. Run the kubectl edit pod <pod name> command.
- This will open the pod specification in an editor (vi editor). Then edit the required properties. When you try to save it, you will be denied. This is because you are attempting to edit a field on the pod that is not editable.

- A copy of the file with your changes is saved in a temporary location.
- You can then delete the existing pod by running the command:
kubectl delete pod podname
- Then create a new pod with your changes using the temporary file
kubectl create -f /tmp/kubectl-edit.yaml
2. The second option is to extract the pod definition in YAML format to a file using the command
kubectl get pod webapp -o yaml > my-new-pod.yaml
- Then make the changes to the exported file using an editor (vi editor).
- Save the changes

Editing Pods and Deployments

- Then delete the existing pod

kubectl delete pod webapp

- Then create a new pod with the edited file

kubectl create -f my-new-pod.yaml

Edit Deployments

With Deployments you can easily edit any field/property of the POD template. Since the pod template is a child of the deployment specification, with every change the deployment will automatically delete and create a new pod with the new changes. So if you are asked to edit a property of a POD part of a deployment you may do that simply by running the command

kubectl edit deployment my-deployment

BEST PRACTICES & ECOSYSTEM

What we Learn

- Blue-Green Deployments
- Canary Deployments
- Monitoring & Logging
- Argo CD- GitOps made Simple
- Helm Charts - Simplifying deployments
- Security Best Practices
- Troubleshooting - Handling issues

Part 4

Concise Kubernetes

Blue-Green Deployments

Deploying new versions of applications is a crucial part of the development cycle in modern software world. Rolling out updates to production environments is always a critical thing and can be a risky proposition anytime, as even small issues can result in significant downtime and lost revenue.

Blue-Green Deployments are a deployment strategy that mitigates this risk by ensuring that new versions of applications can be deployed with zero downtime.

In this blog, we will discuss how Blue-Green Deployments can be implemented using Kubernetes. We will cover the steps involved in setting up a Blue-Green Deployment in Kubernetes, along with the benefits of using this strategy.

What is Blue-Green Deployment?
A Blue-Green Deployment is a deployment strategy where two identical environments, the “blue” environment and the “green” environment, are set up.

1. Blue environment is the production environment, where the live version of the application is currently running, and
2. Green environment is the non-production environment, where new versions of the application are deployed.

When a new version of the application is ready to be deployed, it is deployed to the green environment. Once the new version is deployed and tested, traffic is switched to the green environment, making it the new production environment.

The blue environment then becomes the non-production environment, where future versions of the application can be deployed.

At any point in time, **only one environment (Blue or Green) handles live traffic.** After testing and verification, traffic is switched from Blue to Green.
How Blue-Green Deployment Works in Kubernetes?

- **Prepare the Blue Environment:**

This is your current application version running as a Kubernetes Deployment.
It serves user traffic via a Kubernetes Service (e.g., blue-svc).

- **Deploy the Green Environment:**

Create a new Deployment for the Green version of your application.
The Green environment does not receive live traffic initially.
Test this new version thoroughly.

- **Switch Traffic to the Green Environment:**

Update the Kubernetes Service to point to the Green Deployment.
This is done by changing the selector of the Service to match the Green Pods.

- **Monitor the Green Environment:**

Monitor the application behavior after switching traffic.
If there are any issues, you can switch back to the Blue environment by re-updating the Service.

- **Clean Up:**

Once the Green version is stable, you can remove the Blue Deployment to save resources.

Benefits of Blue-Green Deployments

1. **Zero Downtime:** Blue-Green Deployments allow new versions of applications to be deployed with zero downtime, as traffic is switched from the blue environment to the green environment seamlessly.
2. **Easy Rollback:** If a new version of the application has issues, rolling back to the previous version is easy, as the blue environment is still available.
3. **Reduced Risk:** By using Blue-Green Deployments, the risk of deploying new versions of applications is reduced significantly. This is because the new version can be deployed and tested in the green environment before traffic is switched over from the blue environment.
4. **Increased Reliability:** By using Blue-Green Deployments, the reliability of the application is increased. This is because the blue environment is always available, and any issues with the green environment can be quickly identified and resolved without affecting users.
5. **Flexibility**: Blue-Green Deployments provide flexibility in the deployment process. Multiple versions of an application can be deployed side-by-side, allowing for easy testing and experimentation.

Canary Deployments - Testing changes in production

Canary Deployment in Kubernetes is a deployment strategy where a new version of an application is gradually rolled out to a small subset of users before rolling it out to everyone. It helps minimize risk by exposing the new version to a limited audience and allows you to monitor its behavior and performance before fully switching over.

What is Canary Deployment?

When running containerized applications in Kubernetes, the platform's inherent flexibility and scalability make it well-suited for canary deployment strategies. For example, if an application is distributed across ten Kubernetes pods, you can designate one pod as the canary, deploy the new version only on that pod, and if all is well, deploy it to the remaining nine pods.

It's important to note that canary deployments are not available by default in Kubernetes—they are not one of the deployment strategies in the Deployment object. Therefore, to carry out canary deployments in Kubernetes you will need some customization or the use of additional tools.

In SHORT, in Canary Deployment:

1. A small number of users (e.g., 5-10%) are served by the new version (Canary).
2. The majority of users continue using the old version (Stable).
3. If the Canary version performs well and meets expectations, traffic is gradually increased until 100% of users are served by the new version.
4. If there's an issue, you can stop the rollout and revert the traffic to the old version.

How Canary Deployment Works in Kubernetes?

1. **Stable Deployment (Current Version)**

Start with your existing application running in Kubernetes.

2. **Deploy the Canary Version**

Create a new Deployment for the Canary version with fewer replicas.

3. Split Traffic Between Stable and Canary

Use a Service to distribute traffic between the stable and canary versions.
Traffic Splitting: Use replicas to mimic traffic weights.

- 90% traffic to Stable (9 replicas).
- 10% traffic to Canary (1 replica).

We can also use an Ingress Controller or Service Mesh for precise control.

4. Monitor the Canary Deployment

- Use monitoring tools like Prometheus, Grafana, or Kubernetes logs to track:LatencyError ratesUser behavior
- If the Canary version performs well, gradually increase traffic.

5. **Gradually Scale Canary Deployment**

Increase the number of Canary replicas or the percentage of traffic routed to the Canary version. For example:

- Step 1: 10% traffic to Canary.
- Step 2: 25% traffic to Canary.
- Step 3: 50% traffic to Canary.

Monitor after each step before proceeding.

6. Full Rollout or Rollback

- Full Rollout: Once the Canary version is stable, scale down the Stable Deployment and scale up the Canary Deployment to handle 100% of traffic.
- Rollback: If issues arise, scale down or stop the Canary Deployment entirely. The Stable Deployment will continue serving users.

C**anary Deployment Benefits**

- **Capacity testing** – when deploying a new microservice to replace a legacy system, it is useful to be able to test in a production environment how much capacity you'll need.

By launching a canary version and testing it on a small fraction of your users, you can predict how much capacity you'll need to scale the system to full size.

- **Early feedback** – many issues that affect end-users only occur in a production environment. Canary deployments can expose features to users in a realistic environment, to observe errors or bugs and obtain user feedback.

This allows quick feedback from users, allowing developers to add new features and deliver what the end-user needs. This helps improve the software and the user experience.

- **Easy rollback** – in a canary deployment, if any severe issues are detected, rollback is instantaneous. It is just a matter of switching traffic back to the primary version or adjusting a feature flag.

Monitoring & Logging - Essential tools and approaches

Kubernetes monitoring helps you identify issues and proactively manage Kubernetes clusters. Effective monitoring for Kubernetes clusters makes it easier to manage your containerized workloads, by tracking uptime, utilization of cluster resources and interaction between cluster components.

Kubernetes monitoring allows cluster administrators and users to monitor the cluster and identify issues such as insufficient resources, failures, pods that are unable to start, or nodes that cannot join the cluster.

Monitoring in Kubernetes focuses on collecting metrics from clusters, nodes, and pods to analyze performance and detect anomalies.

Key Aspects of Kubernetes Monitoring

- **Resource Usage:** Track CPU, memory, disk, and network usage.
- **Application Performance:** Measure application health using metrics such as latency, throughput, and error rates.
- **Cluster Health:** Monitor control plane components (e.g., etcd, API server, scheduler).
- **Event Monitoring:** Capture Kubernetes events for insights into deployments, scaling, and pod failures.

Essential Monitoring Tools

Monitoring solutions must be able to aggregate metrics from across the distributed environment, and deal with the ephemeral nature of containerized resources.

The following are popular monitoring tools designed for a containerized environment.

1. Prometheus: Open-source metrics collection and alerting system.
2. Grafana: Visualization and dashboarding tool.
3. Kube-State-Metrics: Generates metrics about the state of Kubernetes objects.
4. Thanos: Highly available Prometheus setup with long-term storage.

What to Monitor

- **Cluster monitoring** – Keeps track of the health of an entire Kubernetes cluster. Helps you verify if nodes are functioning properly and at the right capacity.
- **Pod monitoring** – Keeps track of issues affecting individual pods, such as resource utilization of the pod, application metrics of the pod.
- **Deployment metrics** – When using Prometheus, you can monitor Kubernetes deployments. This metric shows cluster CPU, Kube state, cAdvisor, and memory metrics.
- **Ingress metrics** – Monitoring ingress traffic can help identify and manage various issues.
- **Persistent storage** – Setting up monitoring for volume health enables Kubernetes to implement CSI. You can also use the external health monitor controller to monitor node failures.

- **Control plane metrics** – You should monitor schedulers, API servers, and controllers to track and visualize cluster performance for troubleshooting purposes.

- **Node metrics** – Monitoring CPU and memory for each Kubernetes node can help ensure they never run out. Several conditions describe the status of a running node, such as Ready, MemoryPressure, DiskPressure, OutOfDisk, and NetworkUnavailable.

Best Practices for Monitoring

1. **Centralize Logs and Metrics:** Use tools like Prometheus and Fluentd to centralize and process data effectively.
2. **Automate Alerts:** Set up Prometheus Alertmanager to automate notifications for critical issues.
3. **Retain Historical Data:** Use long-term storage solutions like Thanos for metrics and Elasticsearch for logs.
4. **Leverage Labels:** Use Kubernetes labels to organize logs and metrics for easier filtering.
5. **Secure Data:** Encrypt logs and metrics, especially when transferring them to external systems.
6. **Monitor Control Plane:** Always monitor the Kubernetes control plane for critical events and anomalies.

Argo CD - Kubernetes GitOps Made Simple

Argo CD simplifies Kubernetes application deployments by automating synchronization with Git, providing robust disaster recovery, and ensuring consistency and visibility across clusters.

Argo CD is a Kubernetes-native continuous deployment tool. Unlike external CD tools that only enable push-based deployments, Argo CD can pull updated code from Git repositories and deploy it directly to Kubernetes resources. It enables developers to manage both infrastructure configuration and application updates in one system.

- By adopting GitOps principles with Argo CD, teams can achieve greater efficiency, security, and reliability in their DevOps workflows.
- GitOps is a methodology for managing software infrastructure and deployments using Git as the single source of truth.

Argo CD offers the following key features and capabilities:

- Manual or automatic deployment of applications to a Kubernetes cluster.
- Automatic synchronization of application state to the current version of declarative configuration.
- Web user interface and command-line interface (CLI).
- Ability to visualize deployment issues, detect and remediate configuration drift.
- Role-based access control (RBAC) enabling multi-cluster management.
- Single sign-on (SSO) with providers such as GitLab, GitHub, Microsoft, OAuth2, OIDC, LinkedIn, LDAP, and SAML 2.0
- Support for webhooks triggering actions in GitLab, GitHub, and BitBucket.

CONCEPT

Concise Kubernetes

Core Concepts:
Git as the Source of Truth: All configuration files, including deployments, services, secrets, etc., are stored in a Git repository.
Automated Processes: Argo CD detects changes in the Git repository and automatically apply them to the Kubernetes cluster. This ensures that the live infrastructure aligns with the desired state defined in Git.
Continuous Deployment: Any changes made in the Git repository are automatically reflected in the Kubernetes cluster, enabling continuous deployment.

Advantages of Using Argo CD:
1. Automated State Reconciliation:
- Argo CD ensures that the live cluster state matches the desired state defined in Git.
- Manual changes made directly to the cluster using kubectl are rejected, ensuring consistency.
2. Change Tracking:
- Changes made in Git (e.g., increasing replicas) are automatically applied to the cluster.
- Maintains a detailed history of changes for audit and review.
3. Rollback Capability:
- Easy rollback to previous configurations using Git history if issues arise.
4. Disaster Recovery:
- If the cluster is deleted or affected by network issues, configurations stored in Git can be reapplied to restore the cluster.

Helm Charts - Simplifying Kubernetes deployments

Helm is a **package manager for Kubernetes**, much like apt for Ubuntu or yum for CentOS. It simplifies the deployment and management of Kubernetes applications by using Helm Charts, which are reusable templates for Kubernetes resources.

A **Helm Chart** is a collection of YAML templates that describe a set of Kubernetes resources needed to deploy and run an application. These templates are parameterized, allowing you to define variables (values) that can customize the deployment without altering the template structure.

Why Helm Charts Are a Game Changer?

Helm Charts reduce the complexity of Kubernetes by:

- Eliminating the need to manually manage large numbers of YAML files.
- Providing a consistent way to deploy, upgrade, and roll back applications.
- Supporting customization through parameterized templates.
- Enabling collaboration of reusable and shareable application templates.

Helm has become an essential tool for Kubernetes users, especially in environments where agility, consistency, and efficiency are critical. By leveraging Helm Charts, teams can focus more on building and scaling their applications instead of dealing with deployment complexities.

Helm helps you manage Kubernetes applications — Helm Charts help you define, install, and upgrade even the most complex Kubernetes application. Charts are easy to create, version, share, and publish — so start using Helm and stop the copy-and-paste.

Components of a Helm Chart

- **Chart.yaml**: Provides metadata about the chart (name, version).
- **Values.yaml:** Contains default configuration values that can be overridden during deployment.
- **Templates Directory:** Contains the YAML templates for Kubernetes resources like Pods, Services, ConfigMaps, etc.
- **Charts Directory:** Used for dependencies, allowing you to bundle other charts required by your application.
- **README.md (Optional):** Documentation about the chart and its usage.

How Helm Charts Simplify Kubernetes Deployments

1. **Ease of Reusability** Helm Charts allow you to package and reuse your Kubernetes configurations. You can create a chart for deploying a web application and use it across multiple environments (development, staging, production) by just overriding configuration values.
2. **Configuration Management** Helm separates the application logic (in templates) from the environment-specific configurations (in values). This allows you to customize deployments by providing a different values.yaml file, making deployments flexible and consistent.
3. **Dependency Management** Charts can include dependencies on other charts. Helm automatically manages these dependencies, ensuring all required components are deployed.
4. **Version Control** With Helm, you can version your charts, making it easy to roll back to a previous version if something goes wrong during an upgrade.

5. **Simplifies Complex Deployments** For applications with multiple microservices, Helm Charts can streamline the deployment by bundling all configurations into a single package. This removes the need to individually manage numerous Kubernetes manifests.
6. **Easy Upgrades and Rollbacks** Helm makes upgrading deployments simple by allowing you to update the values or chart version. If the upgrade fails, Helm provides an easy way to roll back to the last known working state.
7. **Community Charts** The Helm ecosystem has a large repository of prebuilt charts for popular applications like Nginx, MySQL, Jenkins, etc. You can use these charts as-is or customize them according to your needs, saving time and effort.

Security Best Practices in Kubernetes

Kubernetes world is Dynamic and Complex, securing this, is quite challenging. Today, Kubernetes becomes a ready-to-go option in IT infra,so, it is becoming an attractive target for attackers.

And so, securing applications in Kubernetes is a multi-faceted process that involves safeguarding the cluster, workloads, and application data. By correctly implementing Kubernetes security measures, you can protect sensitive data, maintain system stability, and prevent unauthorized access.

Common Security Threats and Challenges

Kubernetes Pod-to-Pod Networking

Kubernetes pod-to-pod networking is the ability for pods to communicate with each other.

The default behavior in Kubernetes is to allow all pods to communicate freely with each other, within the cluster. This unrestricted communication can lead to a situation where a compromise in one pod can quickly lead to a compromise in others.

Configuration Management

Configuration management is another area where Kubernetes security risks can arise. Misconfigurations can lead to security vulnerabilities, making your Kubernetes deployments susceptible to attacks.

Common configuration missteps include the use of default settings, which often don’t prioritize security, granting root access to containers, and failure to limit privileges for Kubernetes API access.

Software Supply Chain Risks

Any Kubernetes deployment includes many software components, both within the Kubernetes distribution, included in container images, and running within live containers. All these components can be a source of security risks.

A primary risk in the insecure or outdated software components. These components might contain known vulnerabilities that can be exploited by attackers. Also, the use of software from untrusted sources can lead to the introduction of malicious software into your Kubernetes deployments.

Runtime Threats

Threats can affect nodes, pods, and containers at runtime. This makes runtime detection and response a critical aspect of Kubernetes security. It's important to monitor Kubernetes deployments for suspicious activity and respond quickly to potential security incidents.

Without effective runtime detection and response, attackers could gain access to a Kubernetes cluster, exfiltrate data, and disrupt critical services without being noticed.

Infrastructure Compromise

Kubernetes nodes run on physical or virtual computers, which can be compromised by attackers if not properly secured. Network and storage systems used by Kubernetes clusters are also vulnerable to attack. Compromised Kubernetes infrastructure can lead to widespread disruption of Kubernetes workloads, data loss, and exposure of sensitive information.

Kubernetes Security Best Practices

1. **Cluster Security**

- Enable Role-Based Access Control (RBAC): Define roles and permissions for users and applications to ensure they only access what's necessary.

Use kubectl auth can-i to verify permissions.

- Restrict Access to the API Server: Use network policies or firewalls to limit access to the Kubernetes API server.
- Audit Logs: Enable and monitor audit logs for suspicious activity.
- Use Secure Communication: Ensure all communication between components is encrypted using TLS certificates.
- Isolate Sensitive Components: Run etcd with secure configurations and restrict access to it.

2. **Workload Security**

- **Use Namespaces for Segmentation:** Isolate workloads by using namespaces to group resources with similar security requirements.
- **Apply Pod Security Standards (PSS):** Use policies (e.g., Pod Security Admission) to enforce:

1. Non-root containers
2. Read-only root file systems
3. Minimal capabilities

- **Run Containers with Least Privilege:**

1. Avoid running as root.
2. Use runAsUser and runAsGroup in the security context.

- **Limit Resource Usage:** Set resource requests and limits to prevent DoS attacks from overloading nodes.
- **Image Security:**

1. Use trusted base images.
2. Regularly scan container images for vulnerabilities using tools like Trivy or Clair.

3. **Network Security**

- **Implement Network Policies:** Use network policies to control traffic flow to and from pods. This includes:

1. Whitelisting ingress and egress traffic.
2. Blocking unnecessary communication.

- **Restrict External Access:** Limit the use of LoadBalancers and expose only necessary services.
- **Use Service Mesh:** Employ service meshes like Istio or Linkerd for fine-grained traffic control, mTLS, and policy enforcement.

4. **Secrets Management**

- **Store Secrets Securely:** Use tools like HashiCorp Vault or AWS Secrets Manager instead of Kubernetes Secrets when possible.
- **Encrypt Secrets at Rest:** Enable encryption for Secrets in etcd by configuring --encryption-provider-config.

Restrict Access to Secrets: Use RBAC to limit access to Secrets.

5. **Supply Chain Security**
   - **Image Provenance:** Use tools like Notary to sign and verify images.
   - **CI/CD Pipeline Security:**
   1. Use static and dynamic analysis tools.
   2. Scan IaC (Infrastructure as Code) templates for misconfigurations.

6. **Node Security**
   - **Harden Node Configurations:**
   1. Regularly update and patch node operating systems.
   2. Disable unused services and ports.
   - **Restrict Access to Node Filesystem:** Prevent pods from accessing the host filesystem unless necessary.
   - **Use Seccomp and AppArmor Profiles:** Implement Seccomp and AppArmor for kernel-level security.

7. **Monitoring and Incident Response**
   - Set Up Monitoring Tools:to monitor cluster metrics.
   - Log Aggregation: Centralize logs using tools like Elasticsearch.
   - Set Alerts: Define alerts for abnormal activity or resource usage.
   - Regular Penetration Testing: Conduct periodic audits and tests

8. **Compliance and Governance**
   - **Enforce Policies:** Use tools like Open Policy Agent (OPA) and Kyverno to enforce security and compliance rules.
   - **Document and Train:** Maintain detailed documentation and train teams on security practices.

Troubleshooting - Handling common K8s issues

Kubernetes is the most popular container orchestration tool. It facilitates a wide range of functionalities like scaling, self-healing, container orchestration, storage, secrets, and more.Troubleshooting in Kubernetes is critical and challenging due to the complex and dynamic nature of Kubernetes architecture.

Why is troubleshooting difficult?

Kubernetes Complexity

- **Distributed Architecture**: Kubernetes is a distributed system with multiple components (e.g., API Server, Controller Manager, etcd, Scheduler). A failure in one component can cascade into others, making diagnosis harder.
- **Multi-Layered Abstractions:** Kubernetes abstracts infrastructure through nodes, pods, services, and deployments. Issues may arise at any layer, and pinpointing the root cause requires understanding all layers.
- **Declarative Configuration:** The desired state is declared in manifests, but debugging issues between the desired and actual states requires interpreting logs, events, and resource descriptions.

Dynamic Environment

- **Ephemeral Workloads:** Pods are transient and can be created, destroyed, or rescheduled dynamically. Troubleshooting issues is challenging since logs and states can disappear when pods terminate.
- **Scaling and Auto-healing:** Kubernetes automatically scales and replaces resources. While this is beneficial, it can obscure the root cause by quickly mitigating symptoms.

Distributed Networking

- **Pod-to-Pod Communication:** Kubernetes uses a complex networking model involving overlays (e.g., CNI plugins like Calico, Flannel). Networking issues can arise from misconfigurations, network policies, or DNS failures.
- **External Traffic:** Debugging issues with ingress controllers, load balancers, or service mesh configurations adds additional layers of complexity.

Shared Responsibility

- **Multiple Stakeholders:** Kubernetes involves developers, DevOps engineers, and infrastructure teams. A misconfiguration in one area (e.g., manifests, resource quotas) can cause issues elsewhere.
- **Shared Resources:** Clusters host multiple applications, often from different teams. Resource contention or namespace conflicts can make troubleshooting challenging.

Observability Challenges

- **Log Aggregation:** Kubernetes logs are scattered across multiple components (pods, nodes, and cluster services). Without centralized logging, gathering relevant logs is cumbersome.
- **Limited Insights:** Out-of-the-box tools like kubectl provide basic insights but often lack depth, requiring third-party observability tools like Prometheus, Grafana, or Fluentd for comprehensive monitoring.

Scaling Adds Complexity

- **Large Clusters:** As the number of nodes, pods, and namespaces grows, so does the complexity of pinpointing an issue.
- **Cross-Cluster Issues:** In multi-cluster setups, debugging issues related to federation or inter-cluster communication can be daunting.

Dependencies on External Systems

- **Storage Systems:** Misconfigured Persistent Volumes (PVs) or storage classes can lead to application failures.
- **CI/CD Pipelines:** Faulty pipelines deploying incorrect or buggy configurations can trigger cascading failures.
- **Cloud Providers:** Cloud-based Kubernetes clusters (e.g., EKS, AKS, GKE) depend on cloud provider services, adding another layer to debug when something fails.

Security Implications

- **RBAC and Policies:** Misconfigured Role-Based Access Control (RBAC) rules or network policies can cause unexpected access issues or application failures.
- **Multi-Tenancy:** Ensuring security in multi-tenant clusters often leads to complex configurations that are hard to troubleshoot.