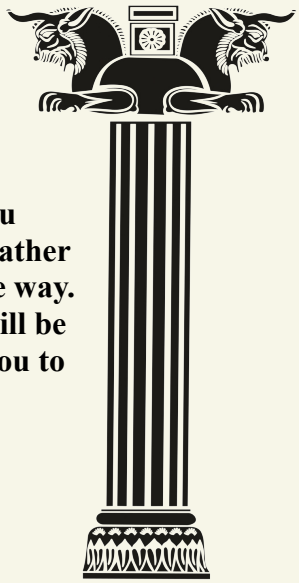
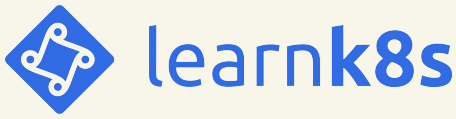
Kubernetes pocket guide

Arye Afshari
Mohsen Shojaei Yegane



The Kubernetes Pocket Guide is a small and easy-to-use document that helps you understand Kubernetes better. Inside this booklet, we have taken great care to gather and explain all the important ideas and knowledge about Kubernetes in a simple way. Whether you're just starting out or already have experience with it, this guide will be your helpful companion. It provides clear explanations and makes it easier for you to learn the basics of Kubernetes

Sponsored by learnk8s, this booklet is offered freely to the public. Learnk8s, an esteemed educational platform, specializes in Kubernetes training courses, workshops, and educational articles. Additionally, this booklet has another standardized format that was produced for learnk8s



@DEV_CHEATSHEET

Note: The content of this booklet is written based on Kubernetes version 1.25

Table of Contents

Core Concept

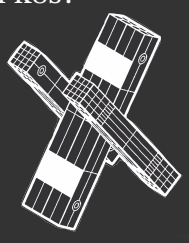
Scheduling

Lifecycle Management

Security

Storage

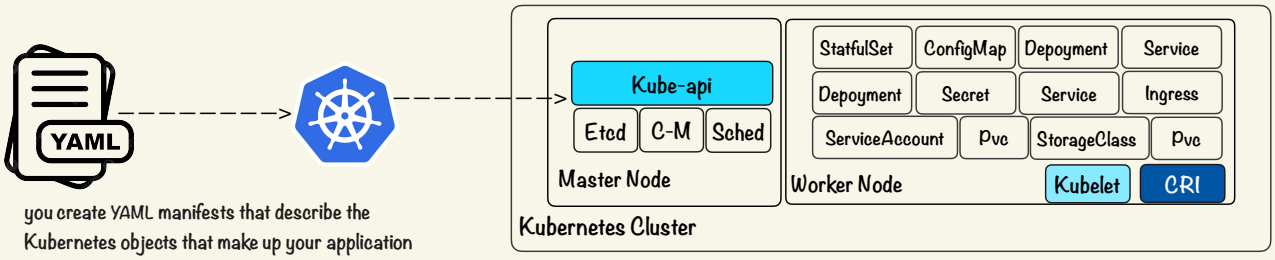
Addons



Kubernetes(k8s)

Kubernetes, also known as K8s, is an open-source platform for managing containerized workloads and services. It provides a way to deploy, scale, and manage containerized applications across a cluster of nodes. Kubernetes was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF)

Kubernetes provides a set of powerful abstractions and APIs for managing containerized applications and their dependencies in a standardized and consistent way. It allows you to declaratively define your application's desired state in the form of a set of Kubernetes objects (such as pods, services, deployments, config maps, and many others), and then Kubernetes takes care of actually running and managing those objects on a cluster of machines



Why you need Kubernetes and what it can do?

Simplify container management: Kubernetes provides a unified API for managing containers, making it easier to deploy and manage containerized applications across multiple hosts or cloud providers

Enhance resiliency: Kubernetes provides built-in fault tolerance and self-healing capabilities, which can help keep applications running even in the face of hardware or software failures

Simplify application deployment: Kubernetes provides a consistent way to deploy and manage containerized applications across different environments, such as on-premises data centers or public cloud providers

Improve scalability: Kubernetes makes it easy to scale containerized applications up or down based on demand, ensuring that applications can handle increased traffic or demand without downtime or disruption

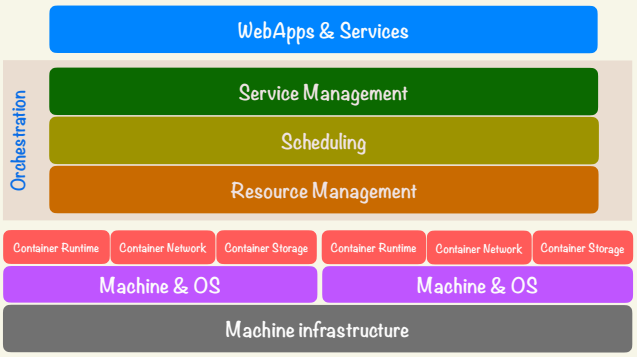
Increase automation: Kubernetes automates many of the tasks involved in deploying and managing containerized applications, such as rolling updates, scaling, and load balancing. This can help reduce the burden on operations teams and improve efficiency

Provide flexibility: Kubernetes is highly configurable and extensible, allowing developers and operations teams to customize it to meet their specific needs. This includes support for different container runtimes, storage systems, and networking plugins

Kubernetes allows you to choose the Container Runtime Interface (CRI), Container Network Interface (CNI), and Container Storage Interface (CSI) that you want to use with your cluster

The CRI is a standardized interface between Kubernetes and the container runtime that is responsible for starting and stopping containers. The CRI abstracts away the details of the container runtime, allowing Kubernetes to work with any container runtime that implements the CRI interface. This makes it possible to use different container runtimes on different nodes in the same cluster, or to switch to a different container runtime without having to modify your applications or infrastructure

The CNI is a standard for configuring network interfaces for Linux containers. Kubernetes uses a CNI plugin to configure the network interfaces for the containers running on your cluster. The CNI plugin is responsible for setting up the network namespace for the container, configuring the IP address and routing, and setting up any necessary network policies or security rules. By using a CNI plugin, Kubernetes makes it easy to switch between different networking solutions or to use multiple networking solutions in the same cluster



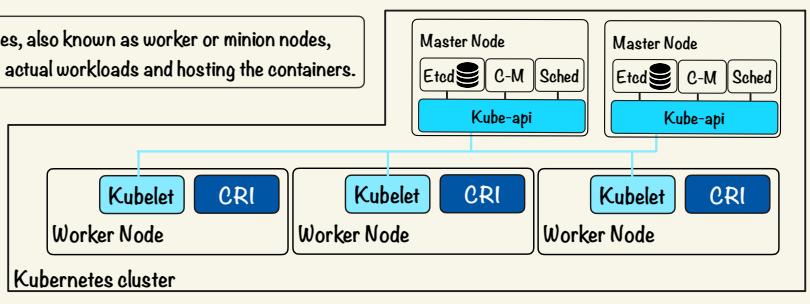
The CSI is a standard for exposing storage systems to container orchestrators like Kubernetes. Kubernetes uses a CSI driver to interact with the underlying storage system. The CSI driver is responsible for managing the lifecycle of the storage volumes used by your applications, including creating, deleting, and resizing volumes. By using a CSI driver, Kubernetes makes it easy to use a wide range of storage systems with your applications, including cloud-based storage solutions, on-premises storage systems, and specialized storage solutions for specific use cases

Container orchestration is the process of managing, deploying, and scaling containers in a distributed environment. It involves automating the deployment and management of containerized applications across a cluster of hosts, and ensuring that the containers are running as expected. Container orchestration systems typically provide features such as container scheduling, load balancing, service discovery, health monitoring, and automated scaling based on demand. Today, Kubernetes is the most popular container orchestration platform used globally

k8s Cluster is a set of nodes that work together to run containerized applications. The nodes can be virtual or physical machines, and they typically run Linux as the operating system. The cluster consists of two main types of nodes:

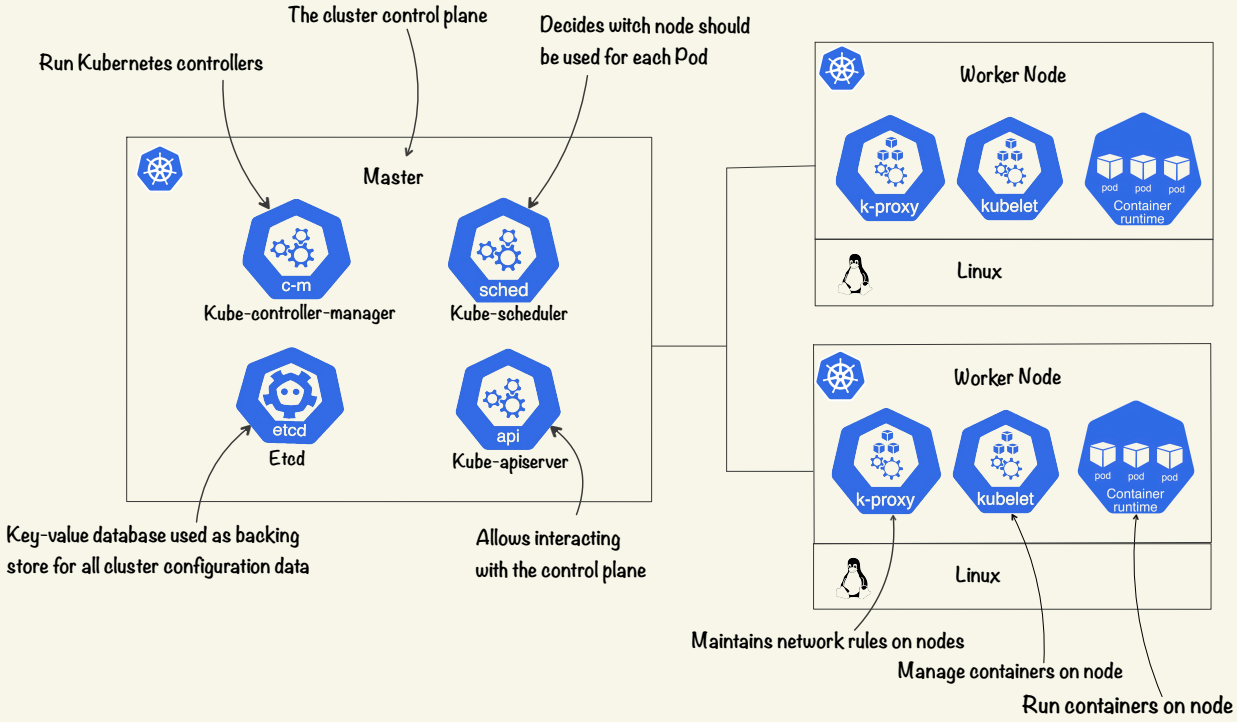
Master Node(s): The master node is responsible for managing the overall state and control of the cluster.

Worker Node(s): The worker nodes, also known as worker or minion nodes, are responsible for executing the actual workloads and hosting the containers.



Kubernetes Architecture:

Kubernetes is built on a master-worker architecture. The master node is responsible for managing the overall state of the cluster, while the worker nodes run the actual application workloads. The components of the Kubernetes master node include the API server, etcd, scheduler, and controller manager. The worker nodes run the kubelet, kube-proxy, and the container runtime



The kube-apiserver is the control plane component that serves as the primary management entity for the cluster. It handles all communication and authentication, and controls all other components of the cluster. Additionally, the kube-apiserver is also responsible for monitoring and controlling the state of the cluster, making sure that all components are running as expected.

Etcd is a distributed key-value database that is used by Kubernetes to store cluster state data. It is responsible for maintaining the configuration details of the Kubernetes cluster and is the only component that interacts directly with the kube-apiserver. etcd provides a reliable and highly available data store for Kubernetes, ensuring that the cluster can recover quickly from failures and maintain consistency across all nodes.

The kube-scheduler is responsible for assigning newly created pods to nodes in the cluster. It reads the list of unassigned pods from etcd and, using a variety of algorithms and configurations, determines which node each pod should run on. Once it has made its decision, the kube-scheduler informs the kube-apiserver, which in turn communicates with the kubelet on the chosen node to start the pod's containers and begin running the workload.

The kube-control-manager is a collection of controllers that manage various aspects of the Kubernetes cluster. These controllers include the node controller, which watches the state of nodes in the cluster and takes actions to ensure that nodes are stable and healthy. For example, if a node fails, the node controller will take actions to ensure that the workloads running on the failed node are rescheduled onto other nodes in the cluster. Other controllers in the kube-control-manager include the replication controller, endpoint controller, and service account and token controllers, which manage other aspects of the cluster such as scaling, networking, and security

The kubelet is the primary node agent that runs on each worker node in the Kubernetes cluster. It is responsible for managing and monitoring the state of containers running on the node, as well as ensuring that the containers are healthy and running as expected. The kubelet communicates with the kube-apiserver to receive instructions on which pods to run on the node, and reports back to the master node with updates on the status of the containers and their health. Additionally, the kubelet also manages the networking and storage configurations for the containers running on the node

The Kube-proxy is responsible for managing the networking and routing configurations for services within the cluster. In Kubernetes, a service functions as an abstraction layer that facilitates communication between pods in the cluster. When a service is established, Kubernetes generates a set of iptables rules on each node within the cluster. Managed by kube-proxy, these rules enable traffic to be accurately directed to the appropriate pods associated with the service, irrespective of the node they operate on. This ensures that communication between the pods and services is both reliable and efficient.

The container runtime is responsible for running containers on each node in the cluster. The container runtime is a software component that manages the lifecycle of containers, including pulling container images from a registry, creating and starting containers, monitoring their health, and stopping or deleting them when they are no longer needed.

Kubernetes components can be run in a Kubernetes cluster as containers or system-level services, depending on their requirements and the needs of the cluster. In general, Kubernetes components that require access to system resources or need to run on the node itself (such as the kubelet and kube-proxy) are run as system-level services on each node. Components that do not require direct access to system resources and can be run in a container (such as the API server, etcd, kube-scheduler, and kube-controller-manager) are typically deployed as containers in pods

Methods of building a Kubernetes cluster:

There are several ways to build a k8s cluster, depending on your requirements and the resources you have available. Here are some common approaches:

Self-hosted Kubernetes cluster: In this approach, you set up and manage your own Kubernetes cluster on your infrastructure. This requires expertise in Kubernetes and infrastructure management, but gives you full control over the environment. You can use tools like kubeadm, kops, Rancher, kubespray to set up and manage the cluster. This approach can be a good fit if you have specific security or compliance requirements, or if you need to customize the environment to your needs.

Cloud-hosted Kubernetes cluster: Most cloud providers offer managed Kubernetes services, such as Amazon EKS, Google Kubernetes Engine (GKE), or Microsoft Azure Kubernetes Service (AKS). With this approach, the cloud provider manages the underlying infrastructure and Kubernetes control plane, while you manage the worker nodes that run your applications. This approach can be more cost-effective and reduces the operational overhead of managing your own infrastructure. It's a good fit if you're already using a cloud provider and want to leverage their managed Kubernetes service.

Cluster as a Service: Cluster as a Service (CaaS) is a cloud-based service that lets you create and manage Kubernetes clusters without worrying about the underlying infrastructure. Providers like DigitalOcean, Linode, and Platform9 offer CaaS solutions that simplify the process of creating and managing Kubernetes clusters. With this approach, you get the benefits of managed Kubernetes services without being tied to a specific cloud provider.

Containerized Kubernetes: You can run k8s as a containerized application on your infrastructure or in the cloud. This approach is useful for development and testing environments, as it lets you spin up a Kubernetes cluster quickly and easily. You can use tools like Minikube, or KinD to create containerized Kubernetes clusters.

In summary, there are several ways to build a k8s cluster, each with its own benefits and trade-offs. The approach you choose will depend on your specific needs and constraints.

How to connect to a Kubernetes cluster

To connect to a Kubernetes cluster, you usually use kubectl. kubectl is a powerful and flexible command-line tool for managing Kubernetes clusters, providing a simple and consistent interface for interacting with Kubernetes resources and performing operations on the cluster

When a user runs a kubectl command, kubectl sends an HTTP request to the Kubernetes API server using the API endpoint specified in the kubectl configuration file. The API server then processes the request, performs the requested operation, and returns a response to kubectl.

The API server uses authentication and authorization mechanisms to ensure that only authorized users can access and modify resources in the cluster.
By default, kubectl uses the credentials and configuration information stored in the .kube/config file to authenticate and authorize requests to the API server

K8s uses a configuration file called "kubeconfig" to store information about how to connect to a Kubernetes cluster. This file contains information about clusters, users, and contexts

```
apiVersion: v1
kind: Config

clusters:
- name: k8s-st1
  cluster:
    certificate-authority-data: <certificate data>
    server: https://127.0.0.1:41285

users:
- name: arye
  user:
    client-certificate-data: <certificate data>
    client-key-data: <key data>

contexts:
- name: arye@k8s-st1
  context:
    cluster: k8s-st1
    user: arye
    namespace: dev

current-context: arye@k8s-st1
```

An example kubeconfig file

provides information about a Kubernetes cluster. Each cluster configuration includes the cluster name, server URL, and any necessary authentication information such as a certificate authority

provides information about a user that can authenticate to a k8s cluster. Each user configuration includes the user name and any necessary authentication information such as a client certificate and key

specifies a cluster and a user to use when connecting to a k8s cluster. Each context configuration also includes an optional namespace that specifies the default namespace to use when executing commands against the cluster

This field specifies the default context to use when executing "kubectl" commands

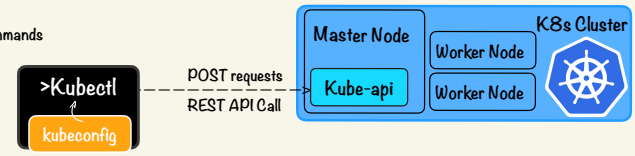
```
sudo kubectl --kubeconfig /etc/kubernetes/admin.conf get node
```

If a configuration file is not present in the ~/.kube directory, we must pass it each time we run a command. To avoid this inconvenience, we can follow these steps

```
mkdir -p $HOME/.kube
sudo scp user@cluster-ip/etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

`kubectl config view` — used to display the current kubeconfig file. It shows all of the clusters, users, and contexts defined in the file

`kubectx` — a third-party utility that can be used to switch between contexts defined in the kubeconfig file



You can use autocompletion for kubectl in zsh and bash

This script provides auto-completion support for kubectl commands and flags when using the zsh shell with the Oh My Zsh framework

```
kubectl completion zsh > ~/.oh-my-zsh/custom/plugins/kubectl.plugin.zsh
```

Once the script is generated and saved in the appropriate directory, you can enable it by adding kubectl to the plugins array in your ~/.zshrc config file

```
...
plugins=(
  git
  kubectl
)
```

To generate a shell completion script for the bash shell, you can use the following command

```
kubectl completion bash > /etc/bash_completion.d/kubectl
```

Having access to the cluster configuration file can potentially allow an attacker to view, modify, or delete resources in the cluster, as well as perform other malicious actions. Therefore, it is important to ensure that access to the cluster configuration file is tightly controlled and restricted to only those who need it

Kubernetes objects

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers

Pods encapsulate and manage application processes and are created using a pod specification, which describes the desired state of the pod, including the containers to run, the network configuration, and any storage volumes to use. Pods are scheduled to run on nodes in the cluster by the Kubernetes scheduler and can be managed using labels and selectors to group and organize them based on their attributes

When creating a pod , you can specify various settings for the pod and the containers running in it. Here's an example YAML manifest that creates a pod

pod-def.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
  namespace: default
  labels:
    app: nginx
    type: front-end
spec:
  containers:
        - name: nginx-container
          image: nginx:1.18
```

specifies which version of the Kubernetes API is used to create and manage the resource

specifies the type of Kubernetes resource

This field contains metadata about the pod, such as its name, namespace,labels, annotations,. .

The spec field contains the specification for the pod, including the containers to run, the network configuration, and any storage volumes to use,..

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#pod-v1-core

To run a container, it must be part of a pod. This means that containers cannot be directly brought up in the cluster without being part of a pod

`kubectl create -f pod-df.yml` The kubectl create -f command is used to create a Kubernetes resource from a YAML file

k is a alias for the kubectl

```
k get pods -o wide -w
NAME        READY   STATUS    RESTARTS   AGE     IP              NODE           NOMINATED NODE   READINESS GATES
pod-nginx   1/1     Running.  0          7m53s   10.244.83.193   kubeworker-1   <none>           <none>
```

kubectl explain command provides detailed information about Kubernetes API resources. It allows you to view the structure, properties, and possible values of any Kubernetes resource

```
kubectl explain pods
Kubectl explain pods.spec.containers
```

`kubectl delete -f pod-df.yml`

this command is used to delete a k8s resource that was created using a YAML file

In Kubernetes, if you update a YAML file and want to apply the changes to a running pod, only a few fields can be updated, and you cannot update all fields in the YAML file. If you make changes that affect fields outside the scope of updateable fields, you must delete the pod and then apply the new YAML file to create a new pod

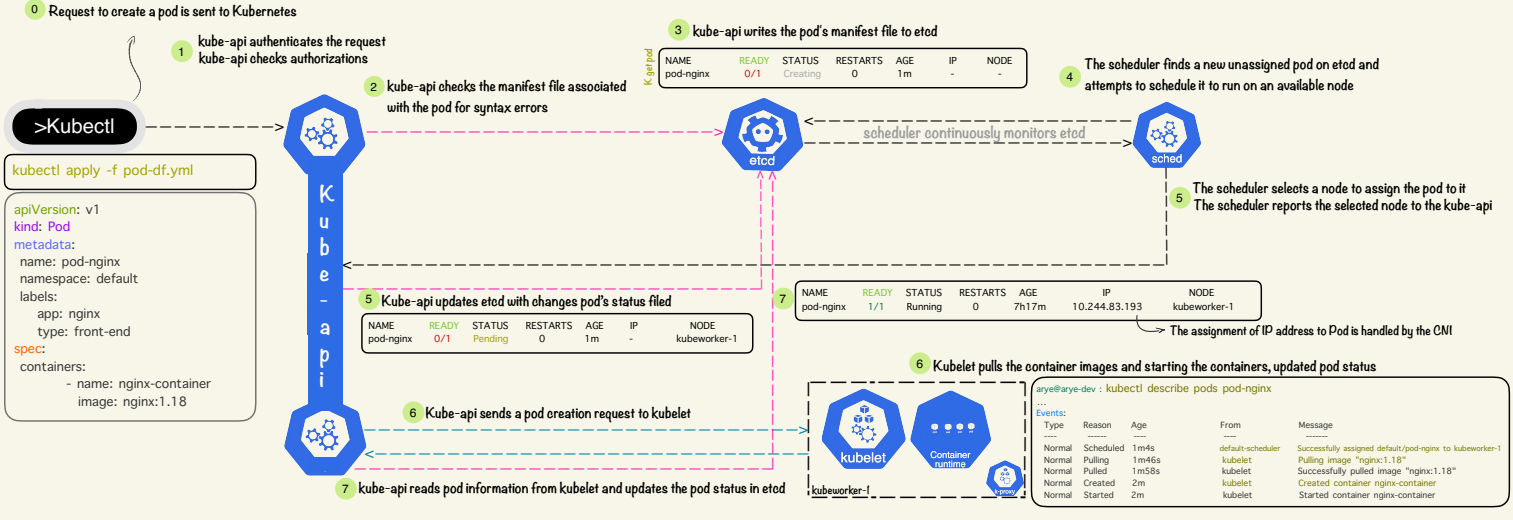
To check the status of a pod , you can use the kubectl describe pod command and check the Events section. This section shows a list of events related to the pod, including the time of occurrence, type of event, and a description of the event. This information can be useful for monitoring and troubleshooting issues with the pod

```
arye@arye-dev : kubectl describe pods pod-nginx

...
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  8m4s               default-scheduler  Successfully assigned default/pod-nginx to kubeworker-1
  Warning  Failed     8m1s               kubelet            Error: ErrImagePull
  Normal   BackOff    8m1s               kubelet            Back-off pulling image "nginx:1.18"
  Warning  Failed     8m1s               kubelet            Error: ImagePullBackOff
  Normal   Pulling    7m46s (x2 over 8m3s)  kubelet         Pulling image "nginx:1.18"
  Normal   Pulled     3m58s              kubelet            Successfully pulled image "nginx:1.18"
  Normal   Created    3m58s              kubelet            Created container nginx-container
  Normal   Started    3m57s              kubelet            Started container nginx-container
```

Process of creating a pod in Kubernetes:

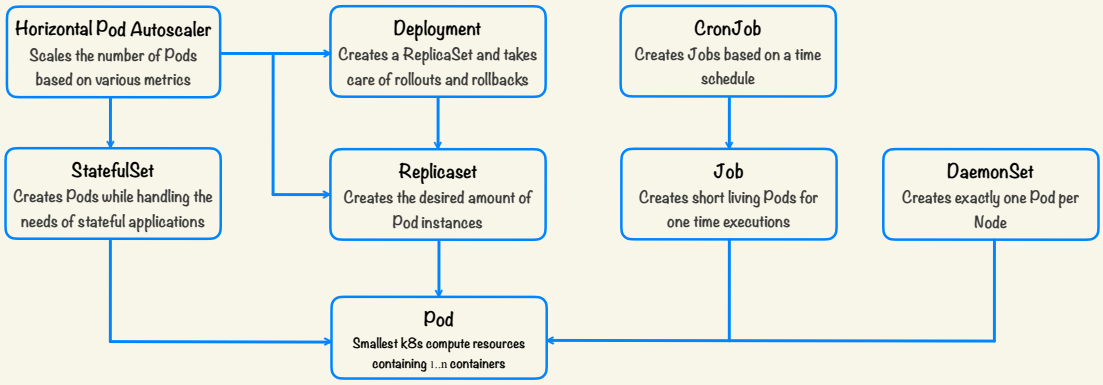
0. **Define Pod specification** : This involves creating a pod manifest yaml file that defines the pod properties like name, labels, containers, volumes etc.
1. **Authentication and Authorization:** When a request to create a pod is sent to Kubernetes through kubectl or the Kubernetes API, the kube-api module first authenticates the request and then checks for the necessary permissions or authorizations to create the pod.
2. **Manifest Syntax Check:** If the authentication and authorization processes are successful, kube-api checks the manifest file associated with the pod for syntax errors. This ensures that the manifest file is well-formed and adheres to the Kubernetes API schema.
3. **Writing to etcd:** If the syntax check is successful, kube-api writes the pod's manifest file to etcd
4. **Pod Scheduling:** The scheduler is responsible for assigning pods to nodes in the cluster based on resource availability and other factors. The scheduler continuously monitors the cluster for new pods and nodes and attempts to schedule the pods to run on the available nodes.
5. **Reporting to API:** The scheduler requests unassigned pods from the Kubernetes API and selects a node to assign the pod to. The scheduler then reports the selected node back to the API, which updates etcd with this information.
6. **Sending Creation Request to Kubelet:** Once the API updates etcd with the selected node information, it sends a creation request to kubelet, the agent running on each node responsible for running the pod. Kubelet then starts the process of creating the pod on the selected node, pulling the necessary container images and starting the containers.
7. **Pod Status Update:** As the pod is being created, kubelet updates the pod status in etcd to reflect the current state of the pod. This includes information such as the pod's phase, container statuses, and IP address.



Workloads

Workload object is a resource that defines how to run a containerized application or a set of containerized applications in a cluster. Workload objects are used to manage the deployment, scaling, and management of containerized applications within a Kubernetes cluster.
The most basic workload object in Kubernetes is the Pod, which represents a single instance of a running container. However, managing Pods directly can be complex and error-prone, which is why Kubernetes provides higher-level workload objects that abstract away the details of Pod management.

Kubernetes workload objects



workload objects in Kubernetes provide a declarative and automated way of managing containerized applications in a cluster. By defining the desired state of your application using workload objects, Kubernetes can handle the details of creating, scaling, and updating the underlying pods that run your application

If a pod is deleted, the system does not automatically recreate it because there is no pod controller in `kube-control-manager`. Therefore, even if you have only one pod, it is better to place it as a subset of a new object called a ReplicaSets that it can be managed by the replication controller. This tool can automatically perform load balancing and scaling.

ReplicaSet is a k8s object that ensures a specified number of replica Pods are running at all times. If a Pod managed by a ReplicaSet fails or is deleted, the ReplicaSet will automatically create a new replica to replace it

Groups

```yaml
apiVersion: apps/v1   # rs-def.yml
kind: ReplicaSet
metadata:
  name: replicaset-nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.17
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

This section is the same as defining a pod, and if a pod is deleted, it can be recreated based on this template

The pods that are subsets of this ReplicaSet must have a label with app: nginx

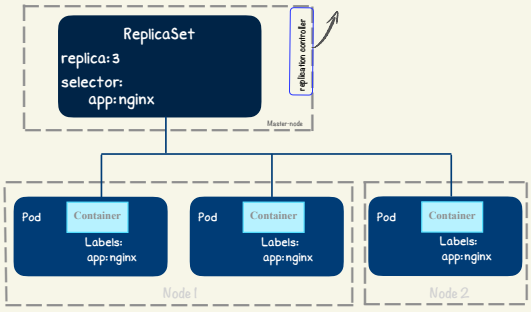
It is recommended to use kubectl apply instead of kubectl create

```
kubectl apply -f rs-def.yml
kubectl describe rs replicaset-nginx
```

```
K get rs
NAME                DESIRED   CURRENT   READY   AGE
replicaset-nginx    3         3         3       13s
```

ReplicaSet is designed to ensure that the current state of the cluster matches the desired state specified in its definition. The desired state is defined by the number of replicas of a specific Pod template that should be running at any given time

The ReplicaSet controller continuously monitors the state of the cluster and compares it to the desired state specified in the ReplicaSet definition. If there are fewer replicas than the desired number, the controller will create new replicas to bring the cluster back to the desired state. If there are more replicas than the desired number, the controller will delete the excess replicas



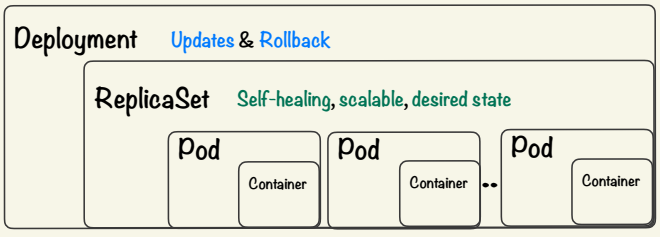

```
K get pod
NAME                    READY   STATUS    RESTARTS   AGE     IP              NODE
pod-nginx               1/1     Running   0          7h17m   10.244.83.193   kubeworker-1
replicaset-nginx-bnrcs  1/1     Running   0          23s     10.244.83.195   kubeworker-1
replicaset-nginx-hm26d  1/1     Running   0          23s     10.244.83.194   kubeworker-1
```

The selector section of the ReplicaSet definition specifies that the pods managed by this ReplicaSet should have a label with key app and value nginx. Since there is already a pod running with the label app: nginx, the ReplicaSet will select it as part of its subset and will only create the remaining two replicas to meet the desired number of 3 replicas.

you can scale the number of replicas of a ReplicaSet using the kubectl scale command, or by updating the replicas field in the ReplicaSet manifest and applying the changes using kubectl apply

```
kubectl scale replicaset=3 rs.def.yml —replicas=6
```

Deployment is a powerful higher-level abstraction that enables you to manage the desired state of your application in Kubernetes. It ensures that a specified number of replicas of your application are always running, by creating and managing other Kubernetes resources like ReplicaSets and Pods. With deployments, you can perform rolling updates and rollbacks, making it easy to update your application without any downtime or quickly revert to a previous version in case of issues.



recommended to use a Deployment to manage the replicas of a stateless application

A Deployment definition is similar to a ReplicaSet definition in that both are used to manage a set of replicas of a pod template. However, the main difference is that a Deployment provides additional functionality for rolling updates and rollbacks of the replicas, whereas a ReplicaSet does not

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: dev
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
  strategy:
    type: RollingUpdate
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

Namespace

In Kubernetes, Namespace is a way to organize and isolate resources within a cluster. A namespace provides a virtual cluster within a physical cluster, allowing multiple teams or applications to coexist within the same Kubernetes cluster

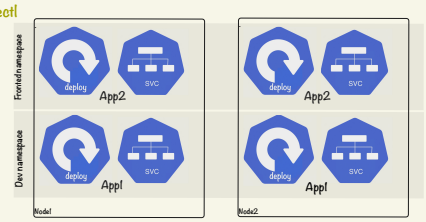
To create a namespace , you can use the kubectl create namespace command

```
kubectl create namespace dev
```

1

```
apiVersion: v1
kind: Namespace
metadata:
 name:dev
```

you can also create a YAML file that defines your namespace and use the kubectl apply command to create the namespace



Each namespace has its own set of resources, such as pods, services, storage volumes that are isolated from resources in other namespaces. This helps to prevent naming conflicts between resources and allows different teams or applications to manage their own resources independently

Namespaces provide a way to organize resources and apply resource quotas, network policies, and other settings at a namespace level. For example, you can limit the number of pods or services that can be created in a namespace, or restrict network traffic between pods in different namespaces.

Resource quotas

Resource quotas in k8s are a way to limit the amount of compute resources that can be consumed by a set of pods in a namespace. A resource quota is defined as a Kubernetes object that specifies the maximum amount of CPU, memory, and other resources that can be used by pods in a namespace

```
kubectl describe resourcequota saas-team-quota
```

=>

This command will display detailed information about the saas-team-quota ResourceQuota object, including the current usage and maximum limits for each resource

```
Name:           saas-team-quota
Namespace:      dev
Resource        Used  Hard
--------        ----  ----
configmaps      0     5
limits.cpu      1     4
limits.memory   2     4Gi
persistentvolumeclaims  0     5
pods            5     10
requests.cpu    1     2
requests.memory 2     2Gi
secrets         0     5
services        1     5
services.loadbalancers  0     2
services.nodeports      0     3
count/deployment.apps   1     4
```

2

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: saas-team-quota
  namespace: dev
spec:
  hard:
    pods: "10"
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "4"
    limits.memory: 4Gi
    configmaps: "5"
    persistentvolumeclaims: "5"
    replicationcontrollers: "5"
    secrets: "5"
    services: "5"
    services.loadbalancers: "2"
    services.nodeports: "3"
    count/deployment.apps: "4"
```

ResourceQuota object specifies the maximum limits for the following resources

The maximum number of pods that can be created in the namespace

The total amount of CPU that can be requested by all pods in the namespace

The total amount of memory that can be requested by all pods in the namespace

The total amount of CPU that can be used by all pods in the namespace

The total amount of memory that can be used by all pods in the namespace

If a ResourceQuota is applied to a namespace but no resource constraints are defined for the pods in the template section of a Deployment YAML file, then the Deployment and ReplicaSet will still be created. However, no pods will enter the running state, as the ResourceQuota will prevent them from consuming any resources

```
k -n dev get events
```

```
$ k describe ns dev
Name:         dev
Labels:       <none>
Annotations:  <none>
Status:       Active
Resource Quotas
 Name:                   comput-quota
 Resource                Used  Hard
 --------                ---   ---
 count/deployments.apps  1     2
 cpu                     6m    100m
 memory                  60M   100M
 pods                    6     10
No LimitRange resource.
```

LimitRange

LimitRange is a resource object that is used to specify default and maximum resource limits for a set of pods in a namespace

When a LimitRange is applied to a namespace, it will only affect newly created pods. Existing pods will not have their resource limits automatically updated to match the LimitRange settings

3

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-resource-limits
  namespace: dev
spec:
  limits:
  - default:
      cpu: 100m
      memory: 128Mi
    defaultRequest:
      cpu: 50m
      memory: 64Mi
    max:
      cpu: 500m
      memory: 512Mi
    min:
      cpu: 50m
      memory: 32Mi
    type: Container
```

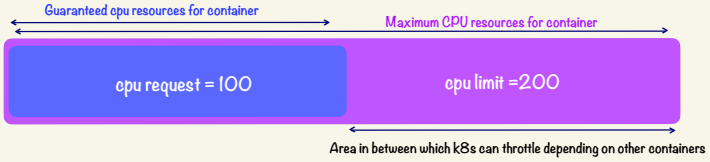
LimitRange is used to set default and maximum resource limits for individual pods or containers within a namespace, while ResourceQuota is used to set hard limits on the total amount of resources that can be used by all the pods in a namespace

Resource Requirements & Limits

Resource requirements and limits are used to specify the amount of CPU and memory resources that a container requires in order to run properly

Resource requirements are set in the pod specification and indicate the minimum amount of CPU and memory resources that a container needs to run. Kubernetes uses these requirements to determine which nodes in the cluster have the necessary resources to schedule the pod

Resource limits specify the maximum amount of CPU and memory resources that a container is allowed to use. Kubernetes enforces these limits by throttling the container's resource usage if it exceeds the specified limit



Container requires at least 100 milliCPU (0.1 CPU) and 10 megabytes of memory to run

Container is limited to using no more than 200 milliCPU (0.2 CPU) and 50 megabytes of memory

```
kunectl describe node kubeworker-1
```

==>

```
...
Capacity:
  cpu:                4
  memory:             8192Mi
  pods:               110
Allocatable:
  cpu:                3
  memory:             7168Mi
  pods:               110
...
```

The Capacity section shows the maximum amount of resources (such as CPU and memory) that a node in the Kubernetes cluster has available

Allocatable section, shows the amount of resources that Kubernetes has allocated for use by containers and pods on the node

4

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: dev
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          resources:
            requests:
              cpu: "100m"
              memory: "10M"
            limits:
              cpu: "200m"
              memory: "50M"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

if you do not specify the request and limits values for a container , the pod will be assigned default values for CPU and memory. The default request value is 0.5 CPU and 256Mi memory, while the default limits value is 1 CPU and 256Mi memory

Setting resource requirements and limits is important for ensuring that containers have the necessary resources to run effectively without overloading the system. By specifying resource limits, you can prevent containers from using too many resources and causing performance issues or crashes

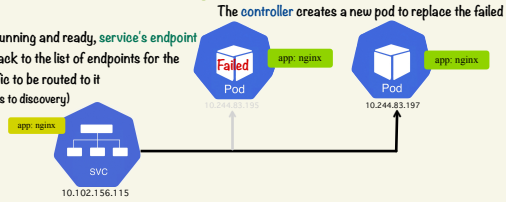
When a container reaches or exceeds its memory limit, the Linux kernel's Out of Memory Killer (OOM Killer) is invoked. The OOM Killer is responsible for selecting and terminating processes to free up memory when system memory becomes critically low. By default, Kubernetes lets the OOM Killer select and terminate the process within the container that triggered the OOM condition.

Service

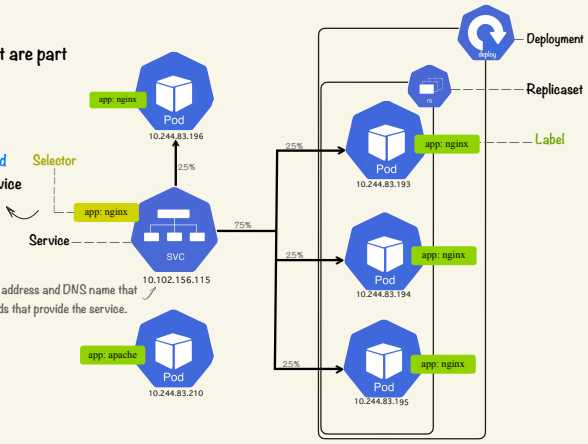
Services are a core component in Kubernetes that are used to manage networking and traffic flow within a cluster. They provide a stable IP address and DNS name for a set of pods and allow for communication between different components within and outside of the application. Services also enable load balancing , service discovery and traffic management, making them a critical component for building scalable and resilient applications in Kubernetes.

When a service is created, it is assigned a virtual IP address (known as a ClusterIP), which is used to route traffic to the pods that are part of the service. The service also has a DNS name, which can be used to access the service from within the cluster

1. If a pod fails or is removed from the service, controller will automatically remove it from the list of endpoints for the service. This ensures that traffic is not sent to a non-existent pod.
2. When a pod managed by a deployment fails, The controller creates a new pod to replace the failed pod
3. Once the new pod is running and ready, service's endpoint controller will add it back to the list of endpoints for the service, allowing traffic to be routed to it (used labels and selectors to discovery)



services use labels and selectors to discover and route traffic to the pods that are part of the service



Each service has a unique IP address and DNS name that can be used to access the pods that provide the service.

Service types in k8s

ClusterIP

This is the default type of service, which exposes the service on a cluster-internal IP address that is only accessible from within the cluster

```
apiVersion: v1
kind: Service
metadata:
    name: nginx-internal
    namespace: dev
spec:
    type: ClusterIP
    ports:
      - targetPort: 80
        port: 80
        protocol: tcp
    selector:
        app: nginx
```

service will load balance traffic to any Pods that have the label app=nginx

The port is used to specify the port number that clients should use to access the service

NodePort

This type of service exposes the service on a static port on each node in the cluster, which can be accessed from outside the cluster using the node's IP address

```
apiVersion: v1
kind: Service
metadata:
    name: nginx-ext
    namespace: dev
spec:
    type: NodePort
    ports:
      - targetPort: 8080
        port: 80
        nodePort:8080
    selector:
        app: nginx
```

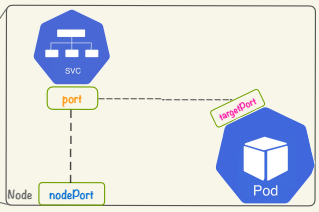
The targetPort is used to specify the port number on the Pods that the service should forward traffic to

The nodePort field is used to specify the high port number on each node in the cluster that can be used to access the service from outside the cluster

LoadBalancer

This type of service exposes the service using a cloud provider's load balancer, which distributes traffic to the different Pods

```
apiVersion: v1
kind: Service
metadata:
    name: nginx-ext-lb
    namespace: dev
spec:
    type: LoadBalancer
    ports:
      - targetPort: 8080
        port: 8080
        nodePort:31090
    selector:
        app: nginx
```



The selector field specifies the label selector that the service will use to find the Pods that it should load balance traffic to

if you don't specify a nodePort value , Kubernetes will automatically allocate a random high port number (30000-32767)for the service

ExternalName

This type of service maps the service to an external DNS name, allowing the service to be accessed from within the cluster using a consistent name

To expose a service to the outside world in k8s, you can use one of the following methods

LoadBalancer Service

This method requires that your cloud provider supports LoadBalancer services, and it can incur additional costs

NodePort Service

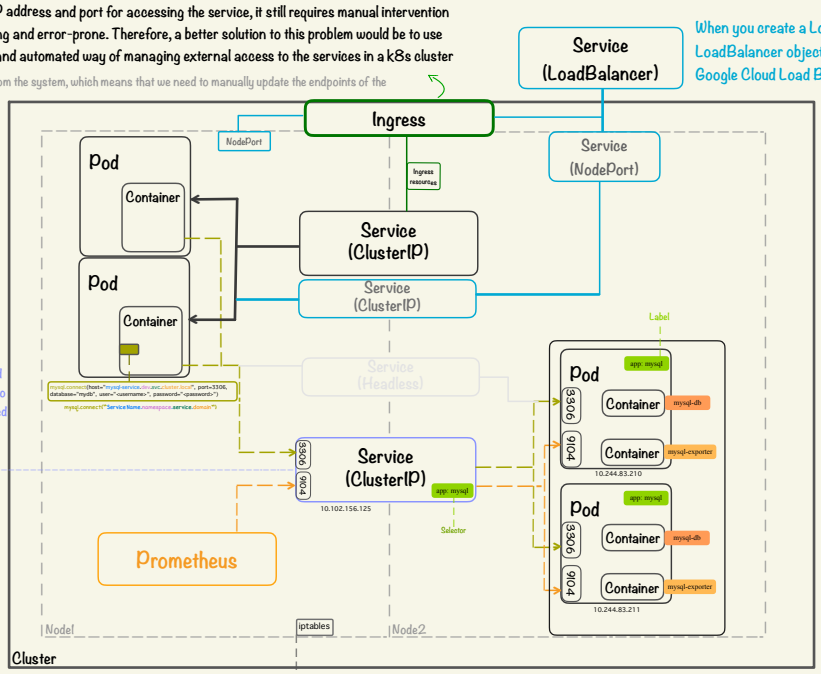
This type of service exposes the service on a static port on each node in the cluster, which can be accessed from outside the cluster using the node's IP address

Ingress

An Ingress is a Kubernetes resource that defines a set of rules for routing external HTTP(S) traffic to a service. Ingress resources require an Ingress controller to be deployed in the cluster, which is responsible for implementing the routing rules. Ingress controllers are available for many popular web servers, such as Nginx, Traefik

While a load balancer service can provide a stable IP address and port for accessing the service, it still requires manual intervention to update the endpoints, which can be time-consuming and error-prone. Therefore, a better solution to this problem would be to use Kubernetes Ingress, which provides a more flexible and automated way of managing external access to the services in a k8s cluster

it is possible that a node might be added or removed from the system, which means that we need to manually update the endpoints of the load balancer or recreate the service.



When you create a LoadBalancer service , it creates a cloud provider-specific LoadBalancer object, such as an Amazon Elastic Load Balancer (ELB) or Google Cloud Load Balancer, which is external to the Kubernetes cluster

When you create a Service of type LoadBalancer, k8s will automatically create a NodePort and ClusterIP for the Service.

Headless services are used for direct access to pods

For services that we do not want to expose to the outside world (such as database clusters like mysql), we set the service type to ClusterIP. If we do not specify the service type, it will be selected as ClusterIP by default

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    matchLabels:
      app: mysql
  ports:
    - name: mysql
      port: 3306
      targetPort: 3306
    - name: mysql-exporter
      port: 9104
      targetPort: 9104
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deployment
  namespace: dev
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:latest
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: password
            - name: MYSQL_DATABASE
              value: mydb
        - name: mysql-exporter
          image: prom/mysqld-exporter:v0.12.1
          ports:
            - containerPort: 9104
          env:
            - name: DATA_SOURCE_NAME
              value: "root:$MYSQL_ROOT_PASSWORD@(localhost:3306)/$MYSQL_DATABASE?tls=false"
```

In the real world, StatefulSets used instead of Deployments for stateful applications

When a service is created, the kube-proxy on each node in the cluster automatically creates iptables rules to forward traffic to the service endpoints.

```
iptables -A KUBE-SERVICES -d 10.102.156.125/32 -p tcp -m comment --comment "/* dev/mysql: cluster IP" -m tcp --dport 3306 -j KUBE-SVC-ABC123
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR	NODE-PORT	ENDPOINTS
mysql-service	ClusterIP	10.102.156.125	<none>	3306/TCP,9104/TCP	1d	app=mysql	<none>	10.244.83.210:3306, 10.244.83.211:3306

```
iptables -A KUBE-SVC-<service-uid> -m comment --comment "dev/mysql-service:" -j KUBE-SEP-<endpoint-uid-1>
iptables -A KUBE-SVC-<service-uid> -m comment --comment "dev/mysql-service:" -j KUBE-SEP-<endpoint-uid-2>
```

Process of creating a service in Kubernetes:

❶ **Define the service:** The first step in creating a service is to define the service using a YAML file or through the Kubernetes API. The YAML file specifies details such as the name of the service, the selector used to identify the pods that the service should route traffic to, and the type of service (ClusterIP, NodePort, or LoadBalancer).

❷ **Submit the service definition:** Once the service definition is created, it can be submitted to the Kubernetes API server

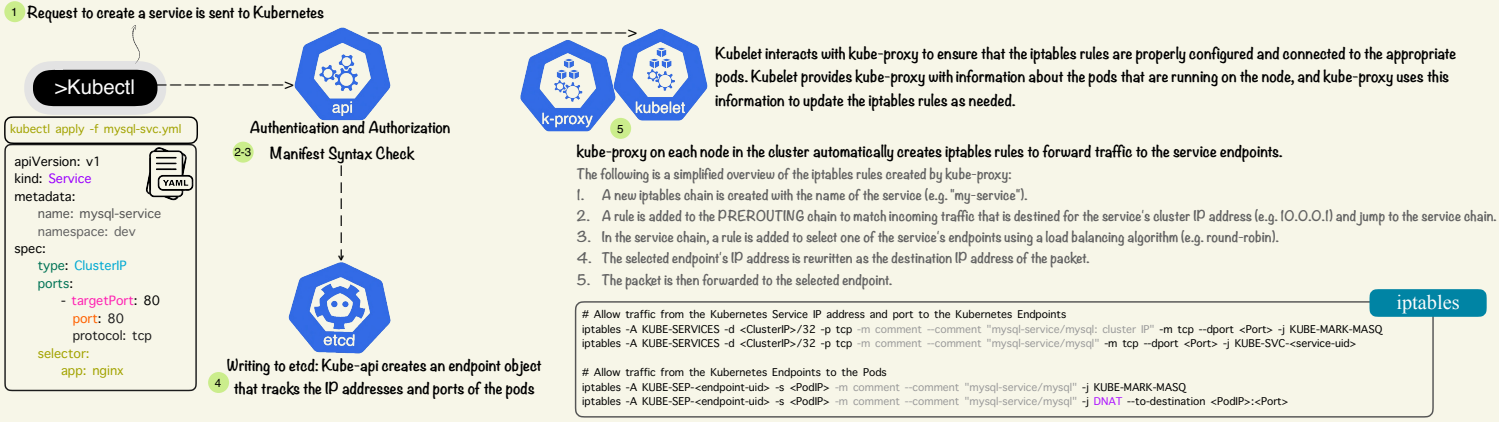
❸ **API Server validates the service definition:** The Kubernetes API server receives the service definition and validates it to ensure that it is well-formed and contains all the required information.

❹ **Service is created:** Once the service is created, Kubernetes creates an endpoint object that tracks the IP addresses and ports of the pods that the service should route traffic to. This information is stored in etcd

❺ **iptables rules are created:** Once the endpoint object is created, kube-proxy creates iptables rules on each node in the cluster to route traffic to the pods that are part of the service. These iptables rules are used to ensure that traffic is routed to the correct pod, and that traffic is load balanced across multiple pods if more than one pod matches the selector.

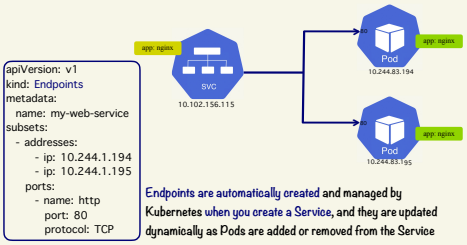
❻ **Access the service:** The service is now accessible within the cluster using its name or DNS name, and can be used to route traffic to the pods that are part of the service.

❼ **Monitor the service:** Once the service is running, Kubernetes monitors its health and takes action if any issues arise. For example, if a pod fails, Kubernetes will automatically remove it from the list of endpoints for the service.



Endpoint

When a Service is created, the Service controller queries the Kubernetes API server to get a list of all Pods that match the Service's label selector. It then creates an Endpoint object that includes the IP addresses and ports of these Pods, and associates the Endpoint with the Service. The Kubernetes networking layer uses this Endpoint information to route traffic to the appropriate Pods that make up the Service.



Endpoints are automatically created and managed by Kubernetes when you create a Service, and they are updated dynamically as Pods are added or removed from the Service

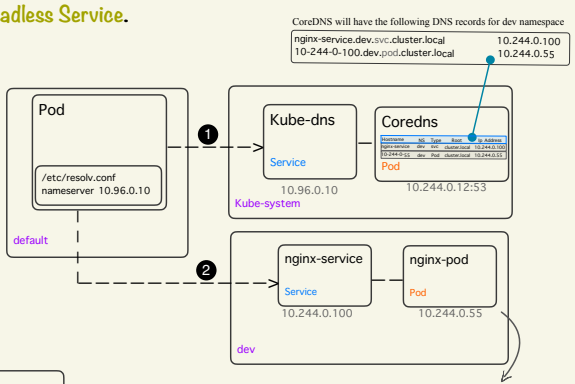
DNS

Kubernetes has a built-in DNS component that provides naming and discovery between pods running on the cluster. It assigns DNS records (A records, SRV records, etc) for each pod/service automatically. The DNS name follows a specific format, such as <service-name>.<namespace>.svc.cluster.local for accessing a Service or <pod-name>.<service-name>.<namespace>.svc.cluster.local for accessing a specific Pod associated with a headless Service.

If a Pod located in the "default" namespace needs to communicate with a service named "nginx-service" residing in the "dev" namespace, it can do so by using the URL "http://nginx-service.dev.svc.cluster.local".

<pod-name>.<service-name>.<namespace>.svc.cluster.local

In Kubernetes, FQDN stands for Fully Qualified Domain Name. It is a complete domain name that specifies the exact location of a resource within the DNS hierarchy. By using FQDNs, Kubernetes simplifies the process of resource discovery, network routing, and namespace isolation within the cluster



The default DNS provider in Kubernetes is CoreDNS, which runs as pods/containers inside the cluster. CoreDNS retrieves pod/service information from the Kubernetes API to update its DNS records.

Notice:Kubernetes does not automatically create DNS records for Pod names directly. This is because Pod IPs keep changing whenever Pods are recreated or rescheduled.Instead, stable DNS records are maintained at the Service level in Kubernetes. Services have unchanging virtual IPs that act as stable endpoints

Pods get DNS resolution indirectly via records in the Pod DNS subdomain:
Each Pod gets a DNS record in the format :
<pod-ip-address>.<namespace>.pod.cluster.local

Pod dns policy

Pod's DNS settings can be configured based on the dnsPolicy field in a Pod specification. This dnsPolicy field accepts three possible values:

ClusterFirst: Any DNS query that does not match the configured search domains for the Pod are forwarded to the upstream nameserver. This is the default policy if dnsPolicy is not specified.

None: Allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the dnsConfig field in the Pod Spec.

Default: Use the DNS settings of the node that the Pod is running on. This means it will use the same DNS as the node that the Pod runs on.

Please note that the Pod's DNS config allows you to customize the DNS parameters of a Pod

In this example, the Pod mypod uses a custom DNS resolver (1.2.3.4) and a custom search list (ns1.svc.cluster-domain.example and my.dns.search.suffix). The option ndots:2 means that if a DNS query name contains less than 3 dots, then the search list mechanism will be used. For example, a query for mypod will be first tried as mypod.ns1.svc.cluster-domain.example and if that fails, as mypod.my.dns.search.suffix.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: myimage
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
```

How scheduling works?

When a Pod is created , it is not assigned to any specific Node initially. instead, the Pod is marked as "unscheduled" and is added to a scheduling queue. The scheduler continuously watches this queue and selects an appropriate Node for each unscheduled Pod.The scheduler uses a set of rules to determine which nodes are eligible for scheduling. These rules include:

Resource requirements:
The scheduler looks at the CPU and memory requirements specified in the pod's configuration and ensures that the selected node has enough available resources to run the pod.

Node capacity:
The scheduler considers the capacity of each node in the cluster, including the amount of available CPU, memory, and storage, and selects a node that has sufficient capacity to meet the pod's requirements

Once the scheduler has identified a set of eligible nodes, it evaluates each node's fitness and assigns a score based on these factors. The node with the highest score is selected, and the pod is scheduled to run on that node.

Taints and tolerations:
Nodes in a Kubernetes cluster can be tainted to indicate that they have specific restrictions on the pods that can be scheduled on them. Pods can specify tolerations for these taints, which allow them to be scheduled on the tainted nodes.

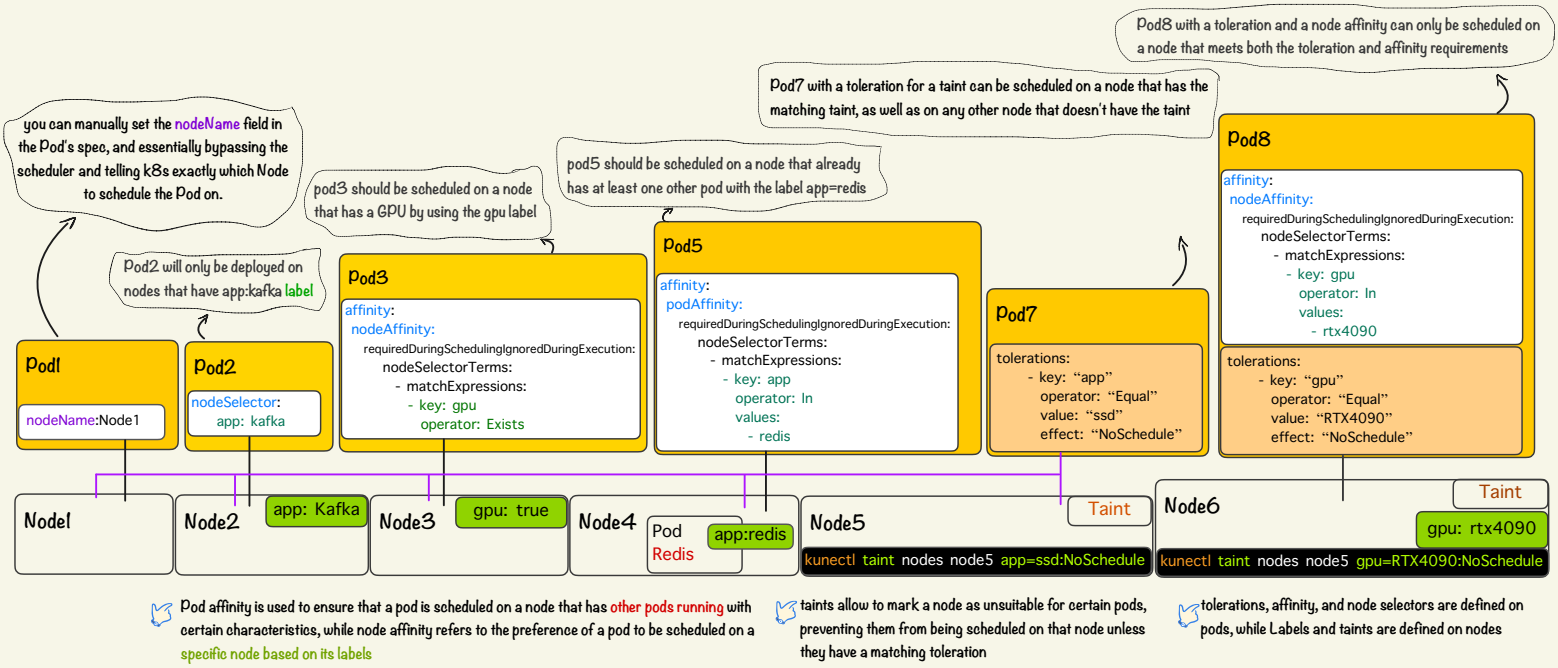
Node selectors:
Users can also specify node selectors, which are labels that are applied to nodes in the cluster. The scheduler can use these selectors to filter out nodes that don't match the pod's requirements.

Kubernetes also provides the ability to filter nodes based on various attributes before selecting them for scheduling. This allows users to specify additional constraints, such as selecting only nodes with specific labels or taints.

(Node,Pod) Affinity/anti-Affinity:
Kubernetes allows users to specify affinity and anti-affinity rules that control which nodes pods can be scheduled on. For example, a pod may be required to run on a node that has a specific label, or it may be prohibited from running on a node that already has a pod with a certain label.

If the scheduler is unable to find a suitable node for the pod, the pod remains unscheduled and enters a pending state until a suitable node becomes available.

You can constrain a Pod to run on specific nodes or prefer to run on particular nodes. There are several recommended approaches to achieve this, including Node Selector, Affinity/Anti-affinity, and Taint.



Pod affinity is used to ensure that a pod is scheduled on a node that has other pods running with certain characteristics, while node affinity refers to the preference of a pod to be scheduled on a specific node based on its labels

taints allow to mark a node as unsuitable for certain pods, preventing them from being scheduled on that node unless they have a matching toleration

tolerations, affinity, and node selectors are defined on pods, while Labels and taints are defined on nodes

Labels & selector

labels are a powerful mechanism for grouping and organizing related objects, such as Pods, Services, Deployments, and more. Labels are key-value pairs that can be attached to Kubernetes objects, and they can be used for a variety of purposes, such as grouping related objects for easy management, selecting objects for operations such as scaling or updating, and enabling fine-grained access control

there are several ways to use labels to group objects in Kubernetes

- Grouping by object type: You can use labels to group objects based on their type, such as Pods, Services, Deployments, ConfigMaps
- Grouping by application: You can use labels to group objects based on the application they belong to, such as a web application, a database, or a caching layer
- Grouping by functionality: You can use labels to group objects based on their functionality, such as front-end components, back-end components, databases, caches, authentication services, video processing services

Annotations

Annotations are similar to labels, but they are designed to store additional information that is not used for grouping or selection, They can be used to store information such as version numbers, timestamps, configuration details, and other metadata that is useful for debugging, monitoring, or other purposes

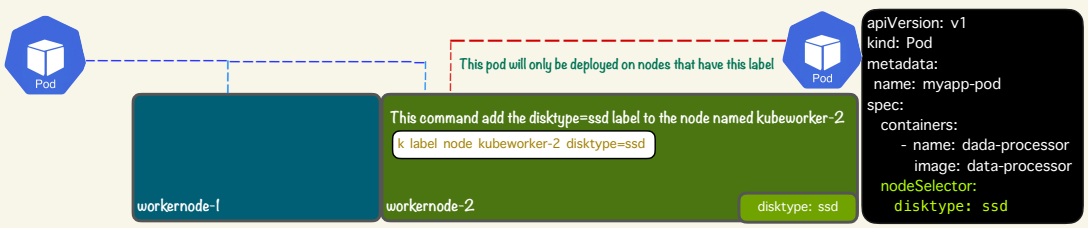
you can use annotations to configure the Nginx ingress controller. However, for more complex configurations, it can be easier to maintain and manage your Nginx configuration by using a ConfigMap

```
annotations:
nginx.ingress.kubernetes.io/proxy-cache: "on"
nginx.ingress.kubernetes.io/proxy-cache-path: "/data/nginx/cache"
nginx.ingress.kubernetes.io/proxy-cache-max-size: "100m"
```

Annotations can be up to 256 kilobytes in size, allowing you to store more complex metadata with Kubernetes objects (labels are limited to 63 characters)

Node selector

NodeSelector is a feature in Kubernetes that allows you to specify a set of labels that a node must have in order for a pod to be scheduled on that node. When you create a pod, you can specify a NodeSelector in the pod spec that will be used to match against the labels of all the nodes in the cluster. If any node has labels that match the NodeSelector, then the pod can be scheduled on that node. also for more complex and multiple constraints such as deploying a Pod on two nodes with different labels, it's better to use Affinity or Anti-Affinity



If a pod's NodeSelector specifies labels that don't exist on any node, the pod won't be scheduled until a node is labeled appropriately

Pending...

This command removes the disktype label from the node named kubeworker-2

```
k label node kubeworker-2 disktype-
```

Affinity and anti-affinity

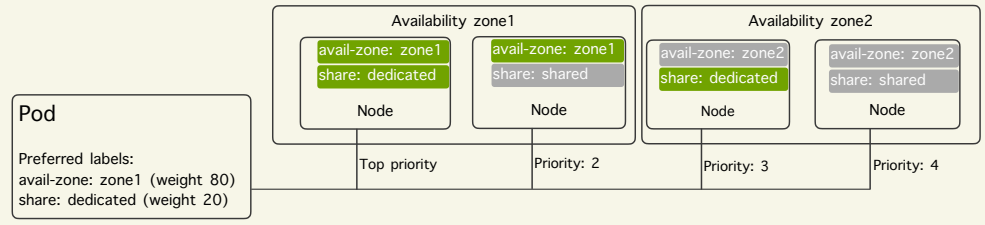
Affinity gives you more control over the scheduling process, allowing you to set rules based on the node's labels or pod's labels. Anti-affinity prevents Pods from being scheduled on the same node or group of nodes.

Affinity Type	Description
Node Affinity	Used to specify rules for which nodes a Pod can be scheduled on based on the labels of the nodes.
Pod Affinity	Used to specify rules for which Pods should be co-located on the same node based on the labels of other Pods running on the node.
Pod Anti-Affinity	Used to specify rules for which Pods should not be co-located on the same node based on the labels of other Pods running on the node.

Each type of Affinity can be further broken down into two categories

Affinity Caregory	Description
Required During Scheduling	Specifies that the rule must be satisfied for the Pod to be scheduled. If the rule is not satisfied, the Pod will not be scheduled.
Preferred During Scheduling	Specifies that the rule should be satisfied for the Pod to be scheduled, but is not required. If the rule is not satisfied, the Pod will still be scheduled.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: database-pod
spec:
  containers:
  - name: database-pod
    image: postgres: 13.11
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 80
          preference:
            matchExpressions:
              - key: avail-zone
                operator: In
                values:
                - zone1
        - weight: 20
          preference:
            matchExpressions:
              - key: share
                operator: In
                values:
                - dedicated
```



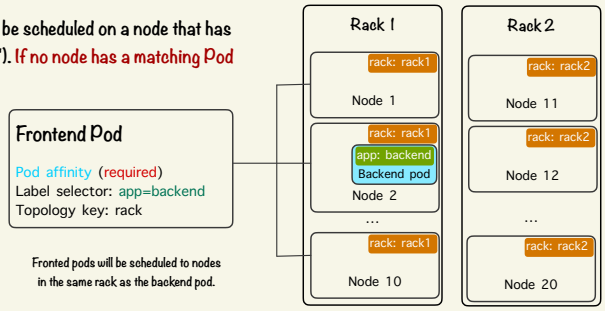
we used preferred Node Affinity to specify that the Pod prefers to be scheduled on nodes with the labels avail-zone: zone1 and share: dedicated. We also assigned a weight to each label to indicate the preference of the Pod. The higher the weight, the higher the priority of the label during scheduling

You can specify a weight between 1 and 100 for each instance of the preferredDuringSchedulingIgnoredDuringExecution affinity type

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend-pod
spec:
  containers:
  - name: frontend-container
    image: frontend-image
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - backend
        topologyKey: "rack"
```

we're using Pod Affinity to specify that the frontend-pod requires that it be scheduled on a node that has a Pod with the label app=backend in the same rack (topologyKey: "rack"). If no node has a matching Pod in the same rack, the frontend-pod will not be scheduled.

You can use the In, NotIn, Exists and DoesNotExist values in the operator field for affinity and anti-affinity.

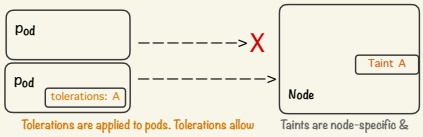


Taints & Tolerations

Node affinity is a property of pods that can either prefer or require certain nodes for scheduling. In contrast, taints allow nodes to reject certain pods. Tolerations are applied to pods and enable the scheduler to schedule them on nodes that have the corresponding taints

Taints are defined using the kubectl taint command, and they consist of a key-value pair and an effect. The key-value pair is used to identify the type of taint

```
kubectl  taint  nodes  node-name  key(=value) taint-effect
```



Taint-effect

- NoSchedule: This effect means that no new Pods will be scheduled on the Node unless they have a corresponding toleration. Existing Pods on the Node will continue to run
- NoExecute: This effect means that any Pods that do not have a corresponding toleration will be evicted from the Node. This can be useful for situations where a Node needs to be drained of its Pods for maintenance or other reasons
- PreferNoSchedule: This effect is similar to NoSchedule, but it allows Pods to be scheduled on the Node if there are no other Nodes available that match the Pod's scheduling requirements. However, if there are other Nodes available that do not have the taint, the Pod will be scheduled on one of those Nodes instead.

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: nginx-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:1.18
```


```yaml
apiVersion: v1
kind: Pod
metadata:
 name: db-pod
spec:
  containers:
    - name: mysql-container
      image: mysql:latest
  tolerations:
    - key: "app"
      operator: "Equal"
      value: "ssd"
      effect: "NoSchedule"
```

nginx-pod does not tolerate the taint on the workernode-3 , so it will not be deployed on it

db-pod can still be scheduled on other nodes that do not have that taint

Pods that have this toleration can be scheduled on workernode-3

When you want to deploy a Pod on a specific node, you need to use taint affinity in addition to taints. This is because taints only restrict which nodes a Pod can be scheduled on based on the characteristics of the node, but do not take into account any preferences or constraints specific to the Pod itself

Notice: the node-role.kubernetes.io/master taint is automatically applied by the kubelet on the master node when the cluster is initialized. Its purpose is to reserve the master node for running control plane components and system Pods, ensuring they have dedicated resources and are not scheduled with regular user Pods. To enable scheduling Pods on the master node in Kubernetes, there are two approaches: adding a toleration or removing the applied taint

```
kubectl describe node kubemaster | grep Taint
Taint:          node-role.kubernetes.io/master:NoSchedule
```

Adding a Toleration: By adding a toleration to the Pod's configuration that matches the taint on the master node, the Pod can be scheduled on the master node despite the taint. This allows specific Pods to run on the master node while preserving its dedicated role for control plane components and system Pods.

```
tolerations:
  - key: "node-role.kubernetes.io/master"
    operator: "Exists "
```

Removing the Taint: Another way to allow Pods to be scheduled on the master node is by removing the taint altogether. This approach effectively opens up the master node for scheduling any type of Pod, including regular user Pods. However, removing the taint means that the master node may no longer be exclusively reserved for control plane components and system Pods, potentially affecting the stability and performance of the cluster.

```
kubectl taint nodes <node-name> node-role.kubernetes.io/master-
kubectl taint nodes  kubemaster  node-role.kubernetes.io/master-
```

Warning: the default master taint exists to protect the stability and reliability of the control plane. Removing it is not recommended as it can lead to overloading the master, reduced HA, and potentially cluster failures

Notice: when a node becomes not ready, indicating that it is no longer available to run new workloads, two taints are automatically added to the node: "node.kubernetes.io/not-ready:NoSchedule" and "node.kubernetes.io/not-ready:NoExecute". These taints serve different purposes and affect the scheduling and behavior of pods on the node.

```
kubectl taint nodes <node-name> node.kubernetes.io/not-ready:NoSchedule-
kubectl taint nodes <node-name> node.kubernetes.io/not-ready:NoExecute-
```

You can remove the taints using the "kubectl taint" command with the "--remove" option

Taint/Tolerations & Node Affinity

To achieve fine-grained control over pod scheduling and ensure pods are scheduled on specific nodes while those nodes only accept certain pods, you can use a combination of Node Affinity and Taints and Tolerations.

First, we use Node Affinity to specify the rules for selecting nodes based on their labels

==> Node Affinity: Node Affinity is used to specify rules that determine which nodes a pod can be scheduled on. You can define node affinity rules based on node labels, node fields, or node selectors. By applying node affinity to a pod, you can restrict its scheduling to specific nodes that meet the defined criteria.

```
kunectl label node kubeworker-1 drive=ssd
kunectl label node kubeworker-2 cpu=xeon
kunectl label node kubeworker-3  gpu=yes
```

Second, we use Taints and Tolerations to indicate which pods can tolerate which taints on nodes. We can apply a taint to nodes that should only accept certain pods, and then specify the corresponding tolerations in the pod specification

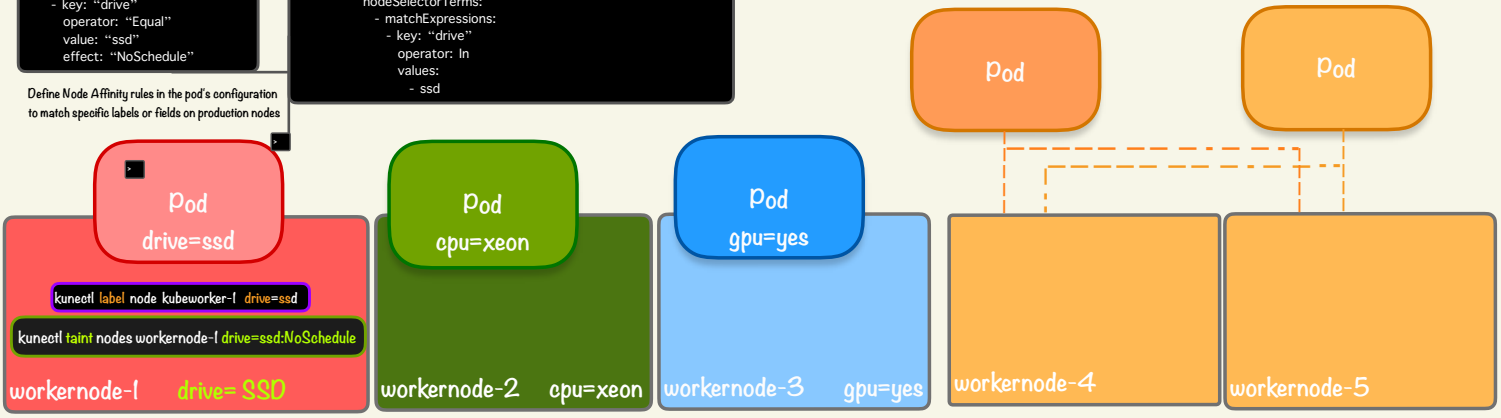
==> Taints and Tolerations: Taints are applied to nodes to repel or prevent pod scheduling by default. However, you can configure tolerations in the pod's configuration to allow specific pods to tolerate specific taints on nodes. Tolerations enable pods to be scheduled on tainted nodes by matching the taint's key and value.

```
kunectl taint nodes workernode-1 drive=ssd:NoSchedule
kunectl taint nodes workernode-2 cpu=xeon:NoSchedule
kunectl taint nodes workernode-3 gpu=yes:NoSchedule
```

```
tolerations:
  - key: "drive"
    operator: "Equal"
    value: "ssd"
    effect: "NoSchedule"
```

```
affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "drive"
              operator: In
              values:
                - ssd
```

Define Node Affinity rules in the pod's configuration to match specific labels or fields on production nodes



With this approach, only pods that have the appropriate tolerations and satisfy the Node Affinity rules will be scheduled on the these nodes. Other nodes without the specific taint or lacking the required labels/fields won't receive these pods

Two Pods do not have any tolerations specified in their PodSpec, while the other two nodes do not have any taints applied. Therefore, the scheduler can schedule these two Pods on either of the taintless nodes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-app1
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nfs
  template:
    metadata:
      labels:
        app: nfs
    spec:
      containers:
        - name: nfs
          image: nfs:latest
```

==>

```
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "drive"
                  operator: In
                  values:
                    - ssd
```

==>

```
      tolerations:
      - key: "drive"
        operator: "Equal"
        value: "ssd"
        effect: "NoSchedule"
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "drive"
                  operator: In
                  values:
                    - ssd
      tolerations:
      - key: "cpu"
        operator: "Equal"
        value: "xeon"
        effect: "NoSchedule"
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: image-processor-app1
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: image-processor
  template:
    metadata:
      labels:
        app: image-processor
    spec:
      containers:
        - name: image-processor
          image: image-processor
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "drive"
                  operator: In
                  values:
                    - ssd
      tolerations:
      - key: "gpu"
        operator: "Equal"
        value: "yes"
        effect: "NoSchedule"
```

priority class & Preemption

priority class is a way to assign a priority value to a Pod, which determines its relative importance compared to other Pods. The priority value can be any integer between 0 and 1000000, with higher values indicating higher priority.
Preemption policies determine whether a higher priority Pod can preempt(evict) a lower priority Pod to be scheduled on a node. There are three preemption policies:

"PreemptLowerPriority" (default): Pods with this priority class are allowed to preempt lower priority Pods if there are no nodes with available resources to schedule them without preemption

"IfNoOtherPods": Pods with this priority class are allowed to preempt lower priority Pods only if there are no other Pods in the cluster that can be evicted to make room for the higher priority Pods

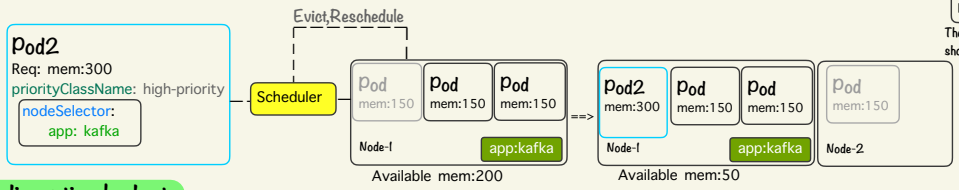
"Never": Pods with this priority class are never allowed to preempt lower priority Pods

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
preemptionPolicy: PreemptLowerPriority
```

The globalDefault field indicates whether this PriorityClass should be used for pods without a PriorityClass

```
apiVersion: v1
kind: Pod
metadata:
  name: db-1
spec:
  containers:
  - name: db-1
    image: mysql:latest
  priorityClassName: high-priority
```

Once you have defined the priority class, you can assign it to a Pod by specifying its spec.priorityClassName field.



Pod disruption budget

Pod Disruption Budget (PDB) in Kubernetes is a way to ensure that a certain number or percentage of pods with an application are not voluntarily evicted at the same time. This can help to maintain high availability during voluntary disruptions like upgrades and maintenance.

In this example, the Pod Disruption Budget named my-pdb specifies that at least two pods with the label app=my-app should be available at all times

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: my-app
```

Bin packing

Bin packing in k8s refers to the process of efficiently utilizing resources by scheduling pods on nodes in a way that maximizes resource usage and minimizes wasted resources. Kubernetes achieves bin packing through its scheduler, which considers factors such as resource requests, limits, and available resources on nodes to make optimal scheduling decisions. Kubernetes scheduler follows two strategies to decide the scheduling of Pods:

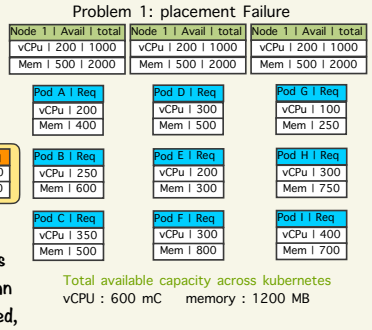
BestFit: In this approach, the scheduler places the incoming Pod in the node with the least amount of free resources after placement. This strategy aims to leave as much space free as possible on every other node.
WorstFit: In this approach, the scheduler places the incoming Pod in the node with the most amount of free resources after placement. This strategy aims to fill up nodes as much as possible.

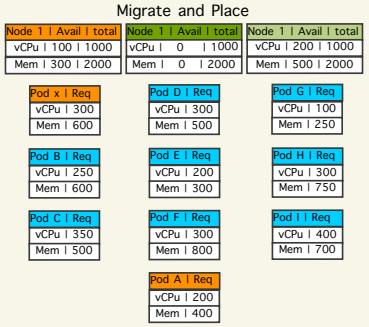
Placement failures can occur in bin packing scenarios in Kubernetes when the scheduler is unable to find a suitable node to schedule a pod due to resource constraints or other constraints defined in the cluster.

scenario: there are three nodes in the cluster, each with 1000m of CPU and 2GB RAM. Currently, there are nine running pods (blue) with their allocated resource requests. However, a new pod (orange) with a request of 300m CPU and 600MB RAM cannot be scheduled. This is due to the unavailability of any node that satisfies both the CPU and RAM requirements of the new pod. Surprisingly, even though the entire cluster has a total of 600m CPU and 1200MB RAM available, the scheduler is unable to find a suitable node.

you can consider moving Pod A from Node1 to Node2. By doing so, you would consolidate the required resources (400m CPU, 900MB RAM) on Node1. This would create enough available resources on Node1 for the pending pod X to be comfortably placed by the scheduler.

Pod X
priorityClassName: high-priority
—— | Req
vCPu | 300
Mem | 600





The operation of moving Pod A from Node1 to Node2 can be performed manually by directly interacting with the Kubernetes API.this can be done using command.Additionally, adjusting the priorities of your pods can help in scenarios where resources are scarce. By assigning appropriate priorities to your pods, you can ensure that critical pods have higher priorities compared to less critical pods. When resources become limited, the Kubernetes scheduler can use these priorities to make decisions about which pods to preempt in order to make room for higher priority pods. By preempting lower priority pods, Kubernetes ensures that critical pods get scheduled and receive the necessary resources. This helps in optimizing resource utilization and ensuring that important workloads are given priority even in resource-constrained environments.

Imbalanced placement in Kubernetes refers to a situation where the distribution of pods or workloads across the nodes in a Kubernetes cluster is uneven or skewed. This can lead to certain nodes being overloaded while others are underutilized, resulting in inefficient resource allocation and potential performance issues. There are a few common causes of imbalanced placement in Kubernetes:

There are a few common causes of imbalanced placement in Kubernetes:

Node labels and pod affinity/anti-affinity: Kubernetes provides mechanisms like node labels and pod affinity/anti-affinity rules to influence the placement of pods. If these rules are not properly configured or if there are inconsistencies in the labels, pods may not be distributed evenly across nodes.

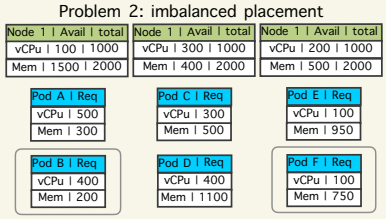
Resource requests and limits: Kubernetes allows you to specify resource requests and limits for pods, indicating the minimum and maximum amount of resources (CPU, memory) they require. If these values are set incorrectly or if there is a wide variation in the resource requirements of pods, it can lead to imbalanced placement.

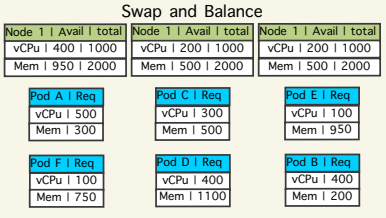
Node capacity and utilization: If the nodes in a Kubernetes cluster have different capacities in terms of CPU, memory, or other resources, it can result in imbalanced placement. Nodes with higher capacity may end up hosting more pods, while nodes with lower capacity may remain underutilized.

scenario: Node1 has high CPU usage (90%) but relatively low memory usage (25%). On the other hand, Node3 has low CPU usage (20%) but high memory usage (85%). This imbalance in resource utilization across the nodes can have the following impacts:

Pod B on Node1: Since Pod B is a CPU-intensive process, the high CPU usage on Node1 indicates that there might be limited CPU resources available for Pod B during peak load situations. This can result in Pod B experiencing CPU starvation, leading to degraded performance or even failures if it requires more CPU resources than what is available.

Pod E on Node3: As Node3 has high memory usage (85%), Pod E, which is running on Node3, might face memory starvation during peak load scenarios. If Pod E requires additional memory resources that are not available due to high memory usage on Node3, it can lead to out-of-memory errors or performance degradation





If we swap Pod B and Pod F between Node1 and Node3, the observation and impact remain the same. Node1 still has 40% CPU usage and 48% memory usage, while Node3 has 50% CPU usage and 55% memory usage. With these resource utilization levels, any pods on these two nodes should still be able to handle any kind of peak load without experiencing resource starvation or performance degradation

Daemonset

A DaemonSet is a type of controller that ensures that all (or some) nodes in a cluster run a copy of a specific pod. It is often used for system-level tasks that should be run on every node, such as log collection, monitoring, or other types of background tasks

When you create a DaemonSet, Kubernetes automatically creates a pod on each node that matches the specified label selector. If a new node is added to the cluster, Kubernetes automatically creates a new pod on that node as well

By using labels and node selectors, you can specify which nodes in the Kubernetes cluster should run a particular DaemonSet. This allows you to restrict the execution of the DaemonSet to specific nodes

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-deamon
  namespace: dev-drive-monitor
spec:
  template:
    metadata:
      labels:
        app: monitoring
    spec:
      containers:
        - name: monitoring-agent
          image: monitoring-agent
      nodeSelector:
        Drive: ssd

  selector:
    matchLabels:
      app: monitoring
```

Only on nodes that have this label, a pod of the DaemonSet type is automatically created

The selector section specifies the label selector used to identify which pods are managed by the DaemonSet

what is the best way to test a DaemonSet on a limited number of nodes without consuming too many resources from the customer's service?

One approach could be to create a separate namespace with a ResourceQuota that limits the amount of resources that can be used by the DaemonSet. This will ensure that the DaemonSet does not consume too many resources from the customer's service

Remove a node:
DaemonSet automatically terminate the corresponding pod

Add a new node:
DaemonSet automatically create a new pod on the node

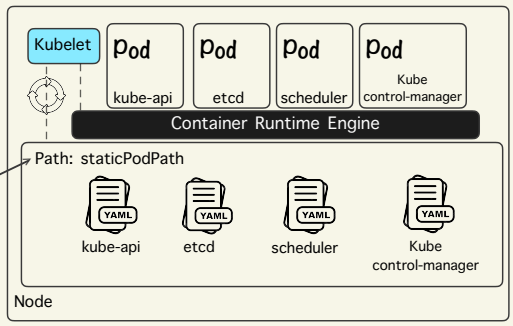
Static Pod

A static pod is a pod that is managed directly by the kubelet on a specific node, rather than by the Kubernetes API server. A static pod is defined by a YAML manifest file that is placed in a specific directory on the node, and the kubelet monitors that directory for changes to the manifest file

The containers come up statically, and their manifest file is located in the directory /etc/kubernetes/manifests. This means that the Kubernetes components, such as the API server, controller manager, and scheduler, are started as containers using pre-defined manifests located in the /etc/kubernetes/manifests directory

StaticPodPath is a directory path where static pod manifests are stored on a node
/var/lib/kubelet/<configuration>.yaml

```
...
staticPodPath: /etc/kubernetes/manifests
```




```
$ kubectl --namespace kube-system get pods -o wide
NAME                                  READY   STATUS    RESTARTS   AGE     IP              NODE
etcd-kubemaster-1                     1/1     Running   4          6d2h    192.168.100.11  kubemaster-1
kube-apiserver-kubemaster-1           1/1     Running   3          6d2h    192.168.100.11  kubemaster-1
kube-controller-manager-kubemaster-1  1/1     Running   3          6d2h    192.168.100.11  kubemaster-1
kube-scheduler-kubemaster-1           1/1     Running   3          6d2h    192.168.100.11  kubemaster-1
```

To delete a static pod in Kubernetes, you can either delete its corresponding manifest file from StaticPodPath or move the manifest file file to another path. Use the following command to remove the manifest file:

```
sudo rm /etc/kubernetes/manifests/<static-pod-manifest.yaml>
```

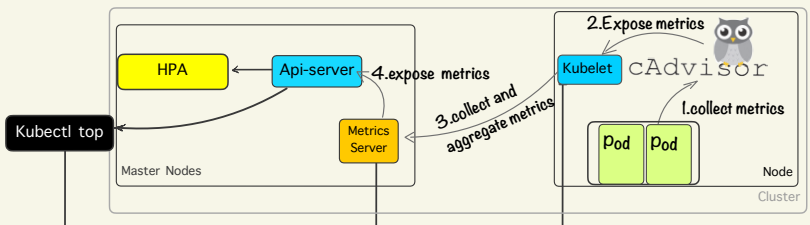
Replace <static-pod-manifest.yaml> with the filename of the manifest associated with the static pod you want to delete.

After performing either of these operations , the kubelet running on the node will detect the change in the static pod directory. It will stop managing the static pod associated with the deleted or moved YAML file, and Kubernetes will initiate the termination process for that pod.

Static PODs	DaemonSets
Created by the kubelet	Created by Kube-API server
Deploy Control Plane components as static pods	Deploy MonitoringAgents, logging Agents on nodes
ignored by the kube-scheduler	

Metrics Server

The Metrics Server is a component of Kubernetes that provides container resource metrics for built-in autoscaling pipelines. It collects resource metrics from Kubelets and exposes them through the Metrics API in the Kubernetes API server. These metrics can be used by the Horizontal Pod Autoscaler and Vertical Pod Autoscaler for autoscaling purposes



cAdvisor (short for "Container Advisor") is a component of the kubelet that is responsible for collecting and monitoring performance metrics for containers and pods running on a node in a Kubernetes cluster

API provided by the kubelet for discovering and retrieving per-node summarized stats available through the /metrics/resource endpoint

The Metrics Server aggregates metrics such as CPU and memory usage and stores in memory

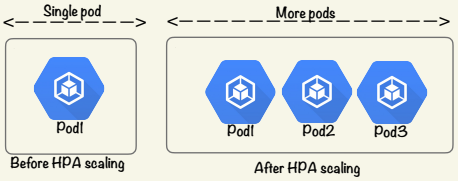
```
kubectl top nodes
NAME      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
node1     50m          5%     983Mi           49%
node2     47m          4%     1043Mi          52%
```

```
$ kubectl top pod
NAME                CPU(cores)   MEMORY(bytes)
pod1                0m           10Mi
pod2                1m           100Mi
```

Autoscaling

Autoscaling refers to the ability of the Kubernetes cluster to automatically adjust the number of running instances of a specific workload or application based on the current demand or load. Autoscaling helps to ensure that there are enough resources available to handle the workload while also optimizing resource utilization.
Kubernetes provides two types of autoscaling mechanisms:

Horizontal Pod Autoscaling (HPA) allows you to automatically scale out your application by adding or removing replicas based on resource utilization metrics such as CPU utilization or custom metrics. This ensures that you have the necessary resources to handle increased traffic or load without over-provisioning resources and incurring additional costs

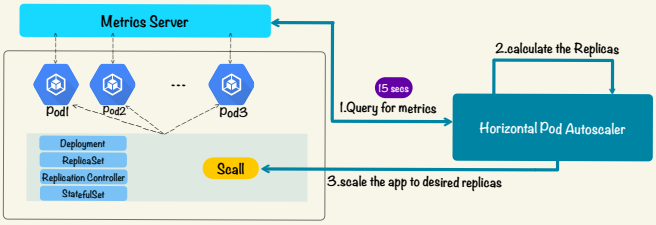


Scaling out, also known as horizontal scaling, is the process of adding more replicas of a Deployment or ReplicaSet to handle an increase in traffic or load

```yaml
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2
metadata:
  name: php-apache
  namespace: dev
spec:
  scaleTargetRef:
    kind: Deployment
    name: php-apache
    apiVersion: apps/v1
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 40
  behavior:
    scaleUp:
      policies:
        - type: Pods
          value: 5
          periodSeconds: 30
        - type: Percent
          value: 100
          periodSeconds: 30
      selectPolicy: Max
      stabilizationWindowSeconds: 40
    scaleDown:
      policies:
        - type: Pods
          value: 4
          periodSeconds: 10
        - type: Percent
          value: 10
          periodSeconds: 10
      selectPolicy: Min
      stabilizationWindowSeconds: 5
```

This specifies the target Deployment for the HPA

minimum and maximum number of replicas for the Deployment



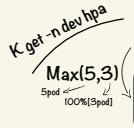
HPA uses the metrics server to collect the metrics data and then uses the scaling algorithm to calculate the new number of replicas needed based on the current load

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
  namespace: dev
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  namespace: dev
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

The metrics section defines the metrics that the HPA uses to scale the Deployment. In this case, two metrics are specified: CPU utilization and memory utilization. For each metric, the HPA calculates the average utilization across all pods over a certain period of time and compares it to the target utilization. If the actual utilization exceeds the target utilization, the HPA increases the number of replicas. If the actual utilization falls below the target utilization, the HPA decreases the number of replicas. By using multiple metrics, the HPA can make more informed scaling decisions

The behavior section defines the scaling behavior for the HPA. In this case, the HPA uses two policies for scaling up and two policies for scaling down. The Pods policy specifies the number of replicas to add or remove, while the Percent policy specifies the percentage of replicas to add or remove. By using both policies, the HPA can scale up or down more quickly or slowly, depending on the workload. The selectPolicy field specifies how the HPA should choose between the Pods and Percent policies. In this case, it's set to Max, which means that the highest value of the two policies will be used for scaling up, and the lowest value will be used for scaling down. The stabilizationWindowSeconds field specifies the number of seconds that the HPA should wait before it starts scaling again after a scaling event. This helps to prevent rapid scaling, which can cause instability in the cluster

The Pods policy specifies that the HPA should add 5 replicas every 30 seconds, while the Percent policy specifies that the HPA should add 100% replicas every 30 seconds




```
NAME         REFERENCE                TARGETS           MINPODS  MAXPODS  REPLICAS  AGE
php-apache   Deployment/php-apache    40%/40%, 20%/50%  3        20       3         1d

NAME         REFERENCE                TARGETS           MINPODS  MAXPODS  REPLICAS  AGE
php-apache   Deployment/php-apache    30%/40%, 15%/50%  3        20       8         1d
```

The Pods policy specifies that the HPA should remove 4 replicas every 10 seconds, while the Percent policy specifies that the HPA should remove 10% replicas every 10 seconds

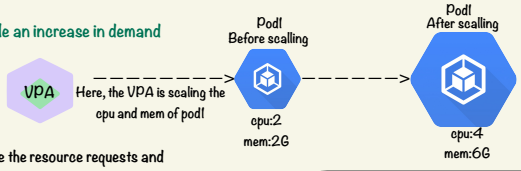
HPA is designed to automatically scale the number of replicas of a deployment or a replica set based on observed CPU utilization, memory utilization, or custom metrics. This makes it well-suited for stateless workloads that can be easily scaled horizontally by adding more replicas

Vertical Pod Autoscaling (VPA) allows you to automatically scale up or down the resource requests and limits of containersin a Pod based on actual resource usage. This ensures that each Pod has the necessary resources to handle the workload efficiently without wasting resources

Scaling up, also known as vertical scaling, is the process of increasing the resources available to each replica of a Deployment or ReplicaSet to handle an increase in demand

```yaml
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: php-apache
  namespace: dev
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: php-apache
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        mode: "Auto"
        controlledValues: "RequestsAndLimits"
        minAllowed:
          cpu: 10m
          memory: 5Mi
        maxAllowed:
          cpu: 200m
          memory: 500Mi
        controlledResources: ["cpu", "memory"]
```

there are two ways to trigger VPA in k8s: automatic and manual,you can set the updatePolicy field to Auto for automatic scaling or Off for manual scaling



! Because of the updateMode field in is set to "Auto", the VPA Updater component will automatically update the resource requests and limits of the containers in the pods.

targetRef: The reference to the target workload object that the VPA should adjust. In this case, it's a deployment with the name php-apache

updatePolicy determines how frequently the pod resource requests and limits should be updated. In this case, it's set to "Auto", which means the VPA will automatically update the resource requests and limits based on the pod's usage

resourcePolicy defines the resource requests and limits for the containers in the target workload. In this case, there is one container policy defined

containerName: The name of the container to apply the policy to. In this case, it's set to *, which means the policy applies to all containers in the target workload

mode determines how the resource requests and limits are set. In this case, it's set to "Auto", which means the VPA will automatically adjust the resource requests and limits based on the pod's usage.

controlledValues: The values that the VPA is allowed to set for the resource requests and limits. In this case, it's set to "RequestsAndLimits", which means the VPA can adjust both the resource requests and limits.

The minimum resource request and limit values allowed for the container are set to 10 milliCPU and 5 MiB of memory. The maximum resource request and limit values allowed for the container are set to 200 milliCPU and 500 MiB of memory

controlledResources: The resources that the VPA is allowed to adjust. In this case, it's set to both CPU and memory.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
  namespace: dev
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: "20m"
            memory: "200Mi"
          limits:
            cpu: "500m"
            memory: "1Gi"
```

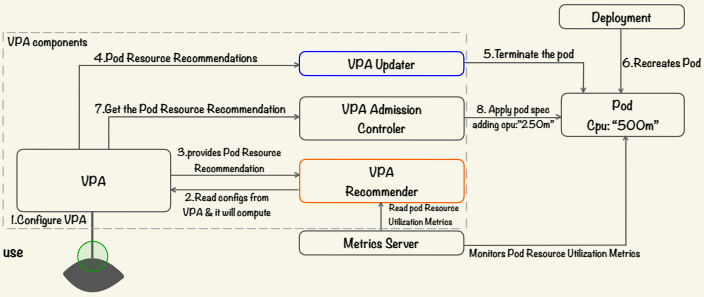
```yaml
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: php-apache
  namespace: dev
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: php-apache
  updatePolicy:
    updateMode: "off"
```

2 Because of the updateMode field in is set to "Off", the VPA Updater component will not automatically update the resource requests and limits of the containers in the pods. In this case, you will need to manually update the resource requests and limits of the pods when necessary
To manually adjust the resource requests and limits, you can update the deployment or statefulset object that the VPA is targeting

kubectl describe vpa command can provide recommendations for the resource requests and limits of containers based on the resource usage metrics collected by the VPA controller

VPA is well-suited for stateful workloads

```
kubectl -n dev get vpa
NAME         MODE  CPU   MEM     PROVIDED  AGE
php-apache   off   163m  262144k True      2m7s
```




```
kubectl describe vpa -n kube-system
...
container Recommendations.
  Container Name: php-apache
  Lower Bound:
    Cpu:     25m
    Memory:  262144k
  Target:
    Cpu:     163m
    Memory:  262144k
  Upper Bound:
    Cpu:     10173m
    Memory:  2770366988
```

Lower Bound: The minimum amount of CPU and memory that the container should have to meet the resource utilization targets

Target: The target amount of CPU and memory that the container should have to achieve the desired resource utilization levels

Upper Bound: The maximum amount of CPU and memory that the container can use

Configure application

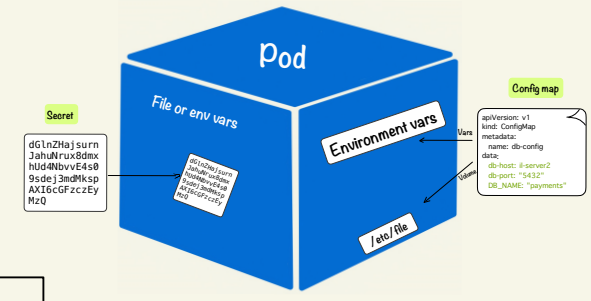
Kubernetes provides several ways to configure applications, including using ConfigMaps, environment variables, and Secrets.

ConfigMaps are Kubernetes resources that can be used to store configuration data as key-value pairs. You can create a ConfigMap with the desired configuration data, and then reference it in your Deployment or Pod specification using the 'configMapKeyRef' field or mount it directly to the pod.

You can create a ConfigMap using the 'kubectl create configmap' command, or by defining a YAML file.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-configmap
data:
  DB_HOST: "mydbhost"
  DB_PORT: "5432"
  DB_NAME: "mydb"
```

```
K get cm
k describe cm db-config
```



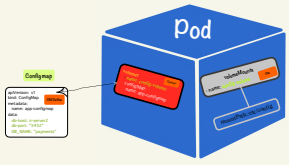
How you can use a ConfigMap to store configuration data

Environment Variables

You can store environment variables in a ConfigMap and use them to configure your application

Configuration Files

You can store configuration files in a ConfigMap and mount them as volumes in your container




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: myimage
    volumeMounts:
    - name: config-volume
      mountPath:/etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-configmap
```

we define a volume named 'config-volume' that maps to the 'app-configmap' ConfigMap using the 'configMap' field. We then mount this volume into the container using the 'volumeMounts' field, which specifies that the volume should be mounted at the path '/etc/config' in the container

Now, any configuration files that are stored in the 'app-configmap' ConfigMap can be accessed by the application running in the container at the '/etc/config' path

Command-Line Arguments

You can store command-line arguments in a ConfigMap and use them to configure your application

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: myimage
    command: ["/bin/myapp"]
    args: ["--config", "/etc/myapp/config.yaml"]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/myapp/
  volumes:
  - name: config-volume
    configMap:
      name: my-configmap
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  config.yaml: |
    setting1: value1
    setting2: value2
```

Environment variables can be used to pass configuration information to the container, such as database connection strings or API keys. You can define environment variables in the Deployment or Pod specification using the 'env' field, the 'envFrom' field, and the 'valueFrom' field

The 'env' field is used to define individual environment variables for a container. You can define the name and value of each environment variable using the 'name' and 'value' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
...
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_HOST
          value: "il-server2"
        - name: DB_PORT
          value: "5432"
        - name: DB_NAME
          value: "payments"
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: "il-server2"
  DB_PORT: "5432"
  DB_NAME: "payments"
```

The 'envFrom' field is used to define environment variables for a container based on a ConfigMap or Secret. You can specify the name of the ConfigMap or Secret using the 'configMapRef' or 'secretRef' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: db-config
        -secretRef:
            name: db-secrets
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
type: Opaque
data:
  DB_USER: dXNlcg==
  DB_PASSWORD: cGFzc3dvcmQ=
```

the 'valueFrom' field is used to define environment variables for a container based on a field in another resource, such as a ConfigMap or Secret. You can specify the name of the resource and the field using the 'configMapKeyRef' or 'secretKeyRef' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
...
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-host
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-port
```

The 'name' field in the 'configMapKeyRef' field specifies the name of the ConfigMap, and the 'key' field specifies the name of the key within the ConfigMap to use as the value for the environment variable

Because we only want to add specific variables from a ConfigMap to a container, we use the 'valueFrom' field

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db-host: il-server2
  db-port: "5432"
  DB_NAME: "payments"
```

Secrets are similar to ConfigMaps, but are used to store sensitive information such as passwords, tokens or API keys. You can create a Secret with the desired sensitive information, and then reference it in your Deployment or Pod specification using the 'secretKeyRef' field.

you can also create a secret by running the kubectl create secret command

```
kubectl create secret generic db-secret --from-literal=username=myuser --from-literal=password=mypassword
```

This command will create a secret named db-secret with two key-value pairs: username and password

```
arye@dev: kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-abc12   kubernetes.io/service-account-token   3      4d
db-secret             Opaque                                2      2h
```

The DATA column shows the number of data items (key-value pairs) in each secret

To use a secret in a pod, you can mount it as a volume or use it as an environment variable

```
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/myapp/secret
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: db-secret
```

To update a secret, you can use the kubectl edit secret command or edit the yaml file directly and apply the changes

```
kubectl edit secret db-secret
```

```
arye@dev: kubectl describe secret db-secret
Name:         db-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  16 bytes
username:  6 bytes
```

By default, the values of the key-value pairs in a secret are base64-encoded to provide a basic level of obfuscation. To decode the values, you can use the base64 command

```
kubectl get secret db-secret -o jsonpath='{.data.password}' | base64 --decode
```

There are several types of secrets in Kubernetes, including:

1. Opaque: This is the default secret type in Kubernetes. It can be used to store any arbitrary data and is encoded in base64.
2. TLS: This type of secret is used to store TLS certificates and keys. It contains two keys: tls.crt and tls.key.
3. Docker-registry: This type of secret is used to authenticate with a Docker registry. It contains the username and password for the registry.
4. SSH: This type of secret is used to store SSH keys. It contains the private key and the public key.
5. Service account: This type of secret is automatically created by Kubernetes when a service account is created. It contains a token that can be used to authenticate the service account.

Warning

Kubernetes Secrets use base64 encoding to obfuscate the sensitive data, it is important to note that base64 encoding is not a form of encryption and can be easily decoded

two solutions to solve this problem

Using external encryption tools or key management systems to secure sensitive data before storing it in Kubernetes Secrets can enhance security.(HashiCorp Vault, Azure Key Vault, and AWS Key Management Service)

you can use access controls to limit who can access the sensitive data. K8s provides various mechanisms for controlling access, such as RBAC and network policies, that can be used to limit access to sensitive data to only authorized users and applications

Application Lifecycle Management

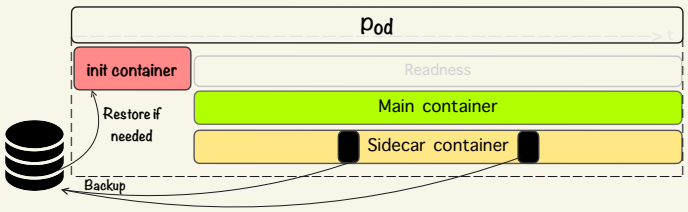
initContainer

An init container is a special type of container that runs before the main container(s) in a pod. The purpose of an init container is to perform some initialization or setup tasks that are required before the main container(s) can start running. Init containers are defined in the same YAML file as the pod specification, alongside the main container(s). They can be used to perform tasks such as setting up a database schema, downloading necessary files, or waiting for a specific service to become available

Init containers have their own lifecycle, and they are considered successful if they complete their tasks without error. If an init container fails, Kubernetes will attempt to restart it until it succeeds, which ensures that the main container(s) in the pod are not started until the initialization tasks are complete

Example 1

```
apiVersion: v1
kind: Pod
metadata:
  name: my-webapp-pod
spec:
  initContainers:
  - name: redis-setup
    image: redis:latest
    command: ["sh", "-c"]
    args:
    - |
      redis-cli ping || exit 1
      redis-cli config set maxmemory 1gb
      redis-cli config set maxmemory-policy allkeys-lru
      redis-cli config set save ""
  containers:
  - name: webapp
    image: my-webapp-image
    ports:
    - containerPort: 80
    env:
    - name: REDIS_HOST
      value: redis-service
    - name: REDIS_PORT
      value: "6379"
```



The init container uses the Redis image and runs a shell command that performs the following tasks:

- Check if the Redis server is running by pinging it
- Set the maximum memory limit to 1 gigabyte
- Set the eviction policy to "allkeys-lru"
- Disable automatic snapshots by setting the save policy to an empty string

After completing its tasks, the init container exits and is terminated. The main container then starts running and serves the web application for the duration of the Pod's lifecycle

Example 2

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-db
spec:
  containers:
  - name: mysql
    image: mysql:5.7
    env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: password
  initContainers:
  - name: migrate-db
    image: mysql:5.7
    command: ['sh', '-c', 'mysql -h ${DB_HOST} -u root -p${DB_PASSWORD} ${DB_NAME} < /migrations/migrate.sql']
    env:
      - name: DB_HOST
        value: 127.0.0.1
      - name: DB_NAME
        value: mydb
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: password
    volumeMounts:
      - name:migrations
        mountPath: /migrations
  volumes:
  - name: migrations
    configMap:
      name: db-migrations
```

When the Pod is started, the migrate-db container runs first and performs the database migration. Once the migration is complete, the mysql-db container starts and runs the application, which now uses the migrated database

Pod lifecycle

Here are the key phases in the lifecycle of a Pod in Kubernetes:

Once a pod's containers are scheduled to run on a node, the pod enters the running phase

When a pod is created, it enters the pending phase. During this phase, the Kubernetes scheduler assigns the pod to a node and the container images are pulled from the container registry. The pod remains in the pending phase until all of its containers are ready and scheduled to run on a node

When a container in a pod completes its task successfully, the container enters the Succeeded phase and the pod is considered to have completed its task

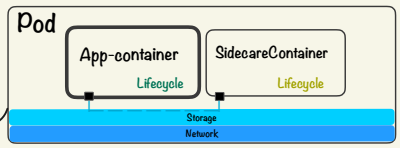
When a container in a pod fails or crashes, the container enters the Failed phase

MULTI-CONTAINER PODs: Sidecar Container | Adapter | Ambassador

Sidecar container is a container that is deployed alongside a main container in a pod . The main container is typically an application that performs some specific function, while the sidecar container provides support or complementary functionality to the main container

The idea behind the sidecar pattern is to keep the main container focused on a specific task or functionality, while delegating other tasks to the sidecar container. This allows for more modular and flexible deployment architectures, as the sidecar container can be updated or replaced independently of the main container

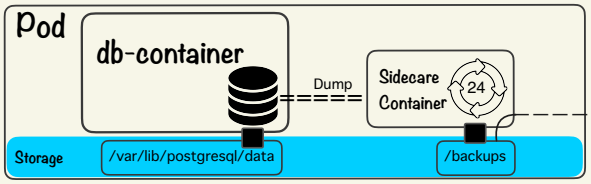
Although containers inside a pod share a common network and storage, they have independent lifecycles and can be created, updated, and deleted individually



important use cases

- Logging and Monitoring: A side containers can be used to collect and forward logs and metrics from the main application container to a central monitoring system
- Backup and Recovery: A side containers can be used to perform backup and recovery operations on the main application container
- Service mesh: A sidecar container can be used to implement a service mesh such as Istio or Linkerd. A service mesh provides additional functionality for managing and securing communications between services running in Kubernetes

One example of how a sidecar container can be used with a database service in a Kubernetes deployment:
The main container is running a database service and is exposing port 5432 for incoming database connections.
The sidecar container is configured to perform backups of the database




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: db-pod
spec:
  containers:
  - name: db-container
    image: my-database-image
    env:
      - name: DATABASE_URL
        value: "postgresql://my-database-hostname:5432/my-database"
    ports:
      - containerPort: 5432
    volumeMounts:
      - name: db-data
        mountPath: /var/lib/postgresql/data
  - name: sidecar-container
    image: my-sidecar-image
    env:
      - name: BACKUP_LOCATION
        value: "s3://my-bucket/my-backups"
      - name: DATABASE_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: database-password
    volumeMounts:
      - name: backup-data
        mountPath: /backups
      - name: db-secrets
        mountPath: /secrets
    command: ["/bin/sh", "-c"]
    args:
      - |
        while true; do
          pg_dump -U postgres -h localhost my-database | gzip > /backups/my-database-$(date +%Y-%m-%d-%H%M%S).sql.gz; s3cmd put /backups/my-database-*.sql.gz "$BACKUP_LOCATION";
          sleep 86400;
          s3cmd put /backups/my-database-*.sql.gz "$BACKUP_LOCATION";
        done
  volumes:
    - name: db-data
      emptyDir: {}
    - name: backup-data
      emptyDir: {}
    - name: db-secrets
      secret:
        secretName: db-secrets
```

The sidecar container can periodically backup the database to a remote location to ensure data resiliency

The sidecar container is running a script that periodically backs up the database and stores the backup files in the "/backups" directory.
The script is also using the "pg_dump" command to perform the backup and gzip to compress the backup file. The backup location is specified in the environment variable "BACKUP_LOCATION", which is set to an S3 bucket. The script is running in an infinite loop and sleeps for 24 hours between each backup.

The two containers are communicating using shared volumes and environment variables. The main container is using a volume mount called "db-data" to store its data files, while the sidecar container is using a volume mount called "backup-data" to store its backup files

Job & CronJobs

Job is a type of resource that allows you to create and manage a finite or batch process in your cluster. Jobs are commonly used for tasks that need to be run once or a few times, such as data processing, backups, or migrations

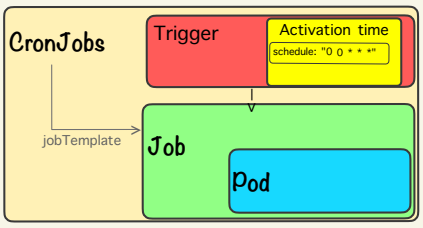
A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-processing-job
spec:
  backoffLimit: 3
  template:
    spec:
      containers:
      - name: data-processor
        image: data-processor:v1.4
        command: ["python", "process_data.py"]
      restartPolicy: Never
```

The backoffLimit specifies the number of times k8s should retry the Job if it fails before giving up

CronJobs in Kubernetes are a way to schedule and automate the execution of Jobs on a recurring basis. A Job is a Kubernetes object that creates one or more Pods to perform a specific task, and a CronJob is a higher-level abstraction that allows Jobs to be scheduled according to a specific time or interval, similar to the Unix cron utility.

Let's say you have a web application that periodically needs to generate reports based on user data. You could create a CronJob that runs a script to generate the report and then terminates when the report is complete.



CronJobs create Jobs which in turn create Pods to run the task

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: report-generation-cronjob
spec:
  schedule: "0 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          ttlSecondsAfterFinished: 100
          containers:
          - name: report-generator
            image: my-django-app:v1
            env:
              - name: DJANGO_SETTINGS_MODULE
                value: myapp.settings
            command: ["python", "manage.py", "generate_report"]
          restartPolicy: Never
```

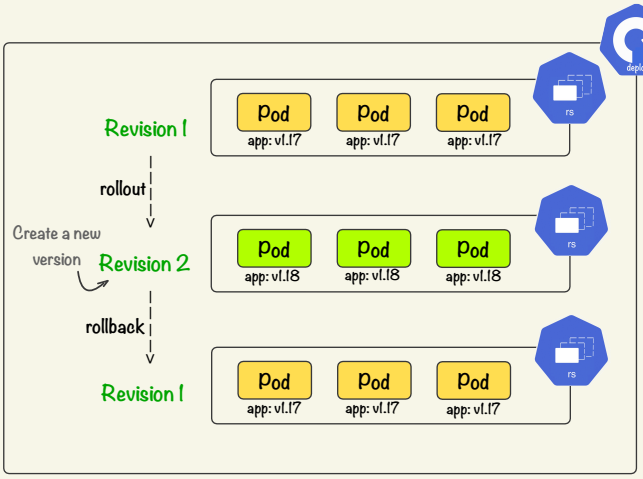
Job will run at the top of every hour

This will delete the Pod 100 seconds after it finishes

Notice: By default, completed Jobs and Pods are retained after running. To automatically clean up completed Jobs, you can set `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` on the CronJob

Rollout & Rollback

Rollout is the process of updating a Deployment or ReplicaSet to a new version of your application
Rollback is performed by updating container with the previous version of the container image



Deployment → Create → ReplicaSet-1
rollback / rollout
ReplicaSet-2

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-imp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.17
        ports:
        - containerPort: 80
```

After adding the 'strategy' section to the YAML file and applying it to the Kubernetes cluster using the 'kubectl apply' command, Kubernetes will start a RollingUpdate for the Deployment

```
kubectl apply -f deployment.yaml
```

Update image to nginx:1.2

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-imp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.18
        ports:
        - containerPort: 80
```

When you perform an upgrade to a deployment, Kubernetes creates a new replica set with the updated container image and configuration, and gradually replaces the pods managed by the old replica set with the pods managed by the new replica set

During a Rolling update , the 'maxUnavailable' and 'maxSurge' settings determine the rate at which replicas are replaced, ensuring that the application remains available and stable throughout the update process

'maxUnavailable' specifies the maximum number of replicas that can be unavailable during the update process. This parameter ensures that the application always has a minimum number of replicas available, even during the update process. For example, if you set 'maxUnavailable' to 1, Kubernetes will not terminate more than one replica at a time during the update process, ensuring that the application always has at least one replica available.

'maxSurge' specifies the maximum number of new replicas that can be created during the update process. This parameter ensures that the update process is efficient and does not overload the system with too many new replicas at once. For example, if you set 'maxSurge' to 1, Kubernetes will not create more than one new replica at a time during the update process, ensuring that the application remains stable and functional throughout the update.

you can also run a rolling update in Kubernetes using the 'kubectl' command

- To perform a rolling update using the 'kubectl' command, you need to have a Deployment defined in Kubernetes

```
kubectl create deployment nginx-imp —image nginx:1.17 —replicas 3
```

create a Deployment with the 'nginx:1.17' image and three replicas

- use the 'set' command in 'kubectl' to update the image used by the Deployment

```
kubectl set image deployment/nginx-imp nginx=nginx:1.18
```

After executing the 'set' command, Kubernetes will start a rolling update for the 'nginx-imp' Deployment

pod name-Replicaset id-pod id

```
kubectl get pod,rs
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-imp-f32gt99mnj-ihd7t      1/1     Running   0          10m
...
pod/nginx-imp-f32gt99mnj-ki34f      1/1     Running   0          10m

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-imp-f32gt99mnj      1         1         1       10m
replicaset.apps/nginx-imp-ht5g34kpz2      0         0         0       12m
```

Old Replicaset — New Replicaset

- You can monitor the progress of the rolling update by running the following command

```
kubectl rollout status deployment/nginx-imp
```

If you want to pause the rolling update at any time, you can use this command:

```
kubectl rollout pause deployment/nginx-imp
```

```
kubectl get pod,rs
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-imp-ht5g34kpz2-ihd7t      1/1     Running   0          10m
...
pod/nginx-imp-ht5g34kpz2-ki34f      1/1     Running   0          10m

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-imp-f32gt99mnj      0         0         0       14m
replicaset.apps/nginx-imp-ht5g34kpz2      1         1         1       16m
```

- If you want to undo the update and roll back to the previous version, you can use the following command:

```
kubectl rollout undo deployment/nginx-imp
```

- You can use 'kubectl rollout history' command to view the revision history of a Deployment, including the rollout status, the version of the Deployment, and the date and time of the revision

```
kubectl rollout history deployment nginx-imp
deployment.apps/nginx-imp
REVISION  CHANGE-CAUSE
1         kubectl create deployment nginx-imp --image=nginx:1.17 --replicas=5
2         kubectl set image deployment/nginx-imp nginx=nginx:1.18
```

The 'CHANGE-CAUSE' field in the 'kubectl rollout history' output is an annotation that is added to the Deployment when it is updated using the 'kubectl set' command
This annotation can be useful for tracking changes and providing additional information about the update process

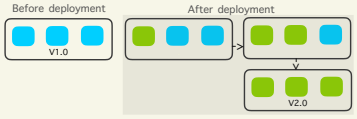
To change the 'CHANGE-CAUSE' annotation for a Deployment in Kubernetes, you can use the 'kubectl annotate' command

```
kubectl rollout undo deployment/nginx-imp --to-revision=2
kubectl annotate deployment nginx-imp kubernetes.io/change-cause="updated to nginx 1.19" --overwrite
```

```
kubectl rollout history deployment nginx-imp
deployment.apps/nginx-imp
REVISION  CHANGE-CAUSE
1         kubectl create deployment nginx-imp --image=nginx:1.17 --replicas=3
2         updated to nginx 1.18
```

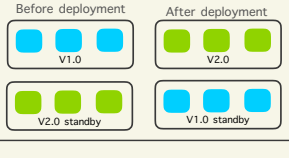
Some of Deployment strategies to perform rollouts

- **Rolling updates** are performed by gradually replacing instances of an old version of a container with instances of a new version. (default deployment strategy)

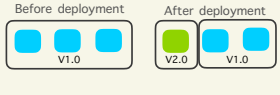


- **Recreate** strategy deletes all the old Pods before creating new ones. This can result in some downtime for your application

- **Blue/Green** strategy creates a new set of Pods running the updated version of your application alongside the old set of Pods running the previous version



- **Canary** strategy updates a small percentage of Pods with the new version of your application, while the rest of the Pods continue to run the previous version



Self-Healing Application

Self-healing applications in Kubernetes are applications that can detect and recover from failures automatically without human intervention. Kubernetes provides several mechanisms to enable self-healing, including probes, replica sets, and deployments. These components together ensure that the desired state of the application is maintained, even in the face of failures, updates, or changes in the environment.

Probes play a vital role in ensuring the health and availability of pods and containers running in a Kubernetes cluster. By periodically checking the health of containers, Kubernetes can take appropriate actions such as restarting containers, marking pods as ready to receive traffic, or delaying traffic until an application inside a container has started successfully

The main idea behind ReplicationControllers and Deployments in Kubernetes is to maintain a desired number of pod replicas running at any given time. In other words, they ensure that a particular pod (or set of pods) always remains up and running.

Kubernetes provides three main types of probes to check the health of Pods

The kubelet is responsible for running probes on containers to check their health

Liveness probes

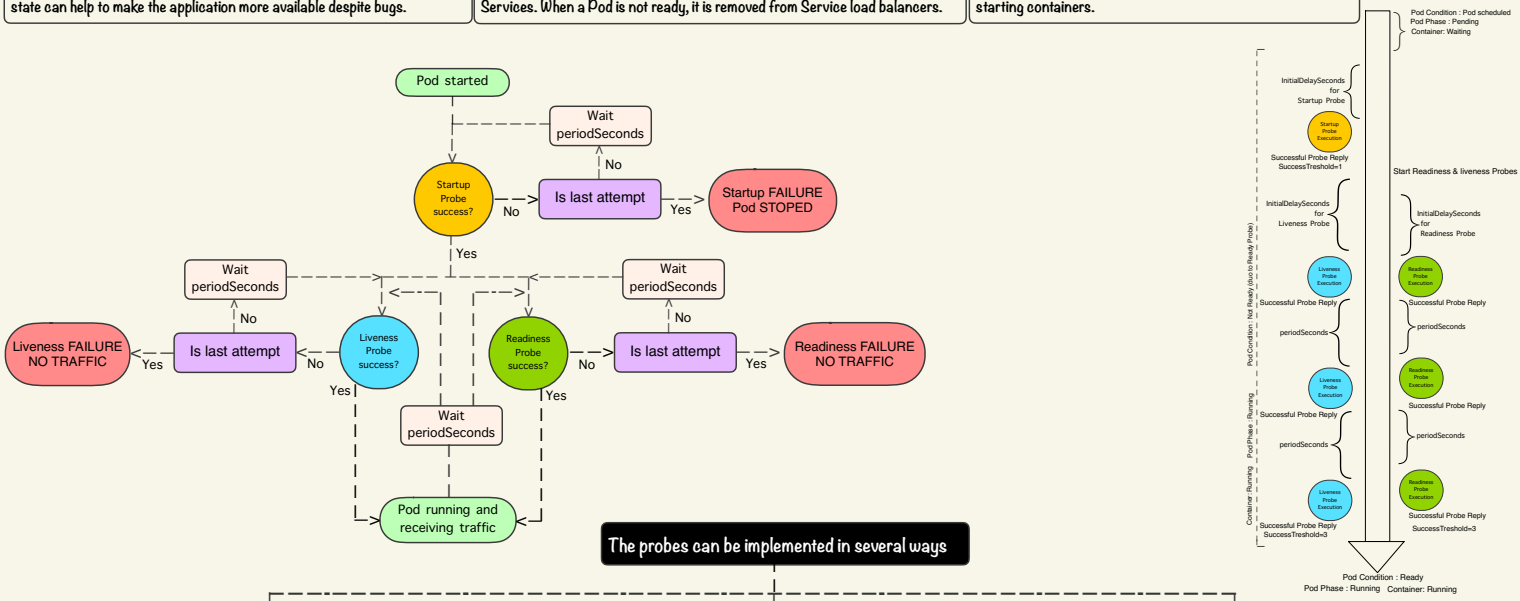
Kubernetes uses liveness probes to know when to restart a Container. For example, a liveness probe could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

Readiness probes

Kubernetes uses readiness probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

Startup Probes

These probes let Kubernetes know when your application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making it useful for slow-starting containers.



The probes can be implemented in several ways

HTTP checks: Kubernetes sends an HTTP request to the specified path of your application. If the application responds with a success status code (200 - 399), the probe is successful. Otherwise, it's considered a failure

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10
```

The "initialDelaySeconds" field indicates that k8s should wait 30 seconds before checking the container's health for the first time

The "periodSeconds" field indicates that Kubernetes should check the container's health every 10 seconds thereafter

The Liveness Probe is configured to use an HTTP GET request to check the container's health. The request is sent to the path "/healthz" on port 8080, which is where the container exposes its health check endpoint

TCP checks: Kubernetes tries to establish a TCP connection to your application on the specified port. If it can establish a connection, the probe is successful. Otherwise, it's failed.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    ports:
    - containerPort: 8080
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 10
      failureThreshold: 3
```

We use the tcpSocket handler to check the container's health by trying to open a TCP connection to port 8080. If the connection is successful, the Liveness Probe is considered successful

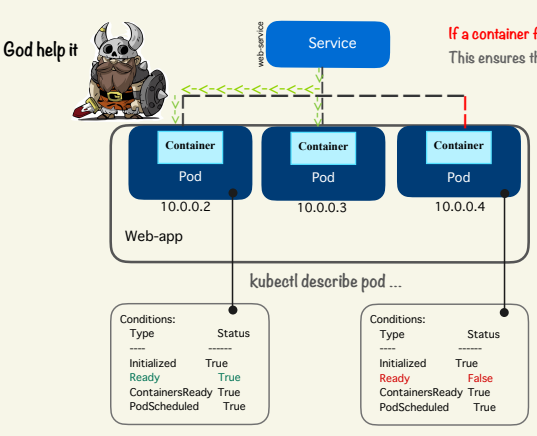
Exec checks: Kubernetes executes the specified command within your container. If the command returns an exit status of 0, the probe is successful. Otherwise, it's considered a failure.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    livenessProbe:
      exec:
        command:
        - /bin/sh
        - -c
        - /usr/bin/custom-script.sh
      initialDelaySeconds: 30
      periodSeconds: 10
```

This probe runs a script inside the container. If the script terminates with 0 as its exit code, it means the container is running as expected

Example: a Web Application with Readiness Probe

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
        - name: web-container
          image: my-web-image
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 5
            failureThreshold: 3
```



If a container fails the Readiness Probe check, it will be removed from the list of endpoints used by the service as a backend. This ensures that the service does not send requests to the container until it becomes ready to receive them again.

```
kubectl describe svc web-service

Name:              web-service
Namespace:         default
Labels:            <none>
Annotations:       Selector:  app=web-app
Type:              ClusterIP
IP:                10.0.0.1
Port:              http  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.0.0.2:8080, 10.0.0.3:8080, 10.0.0.4:8080
Session Affinity:  None
Events:            <none>
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
  type: ClusterIP
```

failureThreshold is a parameter that can specify how many consecutive failures are allowed before the container is considered to have failed the probe.

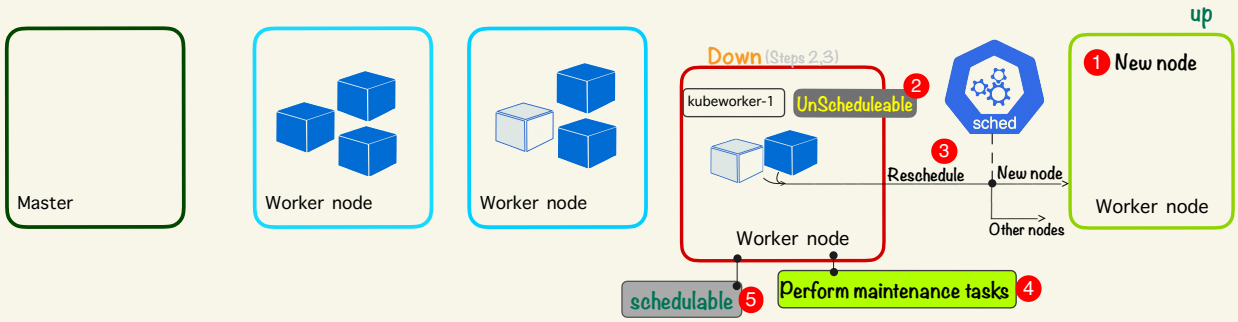
If the deployment fails the probe check three times in a row, the kubelet will restart the pod

```
K describe deploy web-app
Events
      readiness probe failed
```

Cluster maintenance

Node maintenance

Node maintenance in Kubernetes refers to the process of temporarily taking a node out of the cluster to perform maintenance tasks such as upgrading the operating system, applying security patches, replacing hardware or performing other tasks that require the node to be offline. During this time, any workloads running on the node will be evicted and rescheduled onto other nodes in the cluster to ensure high availability and minimal disruption to users.



Nodes need to be regularly updated and maintained to keep the cluster healthy

Steps to perform maintenance on a node in Kubernetes

To avoid any service disruptions during node maintenance, it's important to ensure that your Kubernetes cluster has sufficient resources and capacity to handle the workload of the evicted pods. If a node is added to the cluster, it increases the overall resources available for scheduling pods, reducing the chances of service disruptions. (optional)

Once the drain command completes and all the pods have been successfully rescheduled onto other nodes, you can perform the required maintenance tasks on the drained node. This may include updating the operating system, performing security patches, or any other necessary maintenance activities

Not adding a replacement node may cause the cluster to become unready, especially when there are a large number of pods running on the node being taken down and insufficient resources available on the remaining nodes to allocate to those pods

Node maintenance steps

Add a Node -> Cordon -> Drain -> Perform maintenance tasks -> UnCordon -> Remove added Node [after updating all nodes]

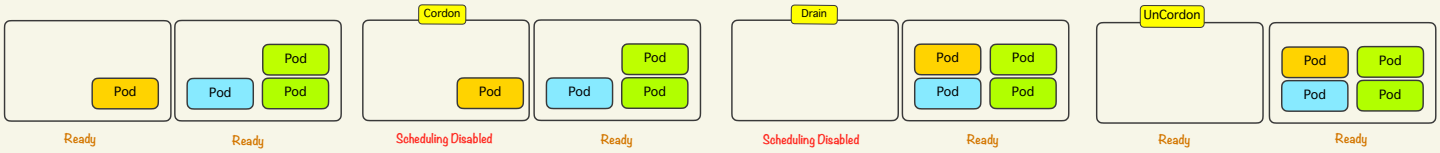
1. kubectl cordon kubeworker-1
2. kubectl drain kubeworker-1
3. Start node updating .....
4. kubectl uncordon kubeworker-1

When maintenance needs to be performed on a node, it should be cordoned as the first step

The cordon command marks a node as unschedulable. It prevents new pods from being scheduled on the node while allowing existing pods to continue running. By running kubectl cordon <node-name>, you indicate that the node is entering maintenance and should not receive any additional workload.

The drain command is used to gracefully evict pods from the node that is undergoing maintenance. It triggers the rescheduling of active pods onto other available nodes in the cluster. Running kubectl drain <node-name> initiates the process of moving pods off the node, ensuring that they are not abruptly terminated.

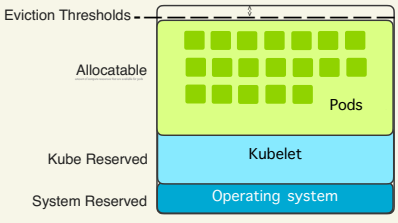
After the maintenance tasks are completed, the node can be brought back online and added back to the cluster. Kubernetes will automatically detect the new node and begin scheduling pods on it again. It's important to note that when a node is added back to the cluster, Kubernetes will not automatically move all the evicted pods back to the node. Instead, the scheduler will treat the node like a new node and schedule new pods onto it based on the available resources and workload requirements



Reserving resources for the operating system and the kubelet in Kubernetes is crucial for maintaining stability

Kubernetes nodes can encounter resource starvation issues when pods consume all available capacity on a node, resulting in an insufficient allocation of resources for critical system daemons and processes that drive the functioning of the operating system and Kubernetes infrastructure. This imbalance can subsequently lead to cluster instability and performance degradation. configuring kubelet resource reserves is a good way to prevent resource starvation issues on Kubernetes nodes.

Here are some ways kube and system resource reserves can help:



kube-reserved This reserves resources for Kubernetes system daemons like kubelet, container runtime, node problem detector, etc. Prevents starvation of critical components.

system-reserved Reserves resources for the underlying node's kernel and system services. Leaves room for OS processes.

eviction-hard The kubelet will evict pods when available resources drop below this threshold to maintain reserves

To configure these reserves, you can set flags on the kubelet service like:

```
--kube-reserved=cpu=500m,memory=1Gi
--system-reserved=cpu=1,memory=2Gi
--eviction-hard=memory.available<500Mi
```

SPOF

Single Point of Failure (SPOF) refers to a component or resource that, if it fails, can cause a complete or partial outage of the entire system. This means that the failure of a single component can result in the unavailability or degraded performance of the overall Kubernetes cluster. Identifying and mitigating SPOFs is crucial for ensuring high availability and reliability in a Kubernetes environment. Here are some recommendations for ensuring the minimum amount of SPOFs for critical Kubernetes components:

Kubernetes Control Plane - Need at least 3 master nodes spread across availability zones. This ensures high availability of API server and controller manager.	etcd - For production, need at least 3 etcd instances, 5 for better redundancy. Should be co-located with control plane nodes.	Worker Nodes - No specific minimum, but have at least 3 nodes in a cluster and spread them across zones.
Load Balancers - Front load balancers with at least 2 instances or use external LB services.	Ingress Controllers - Need 2+ ingress controllers like Nginx for redundancy. Configure with a load balancer.	Data Storage - Use cluster-wide storage like GlusterFS, Rook, OpenEBS with replication.
Cluster Networking - Should have high availability at the network level - multiple switches, routers etc. Avoid SPOF in networking.		

Cluster upgrade

It's important to keep k8s components up-to-date with the latest stable version to ensure that the cluster is secure and stable. Here are the several methods for upgrading a k8s cluster:

Kubeadm: Kubeadm is a popular tool for bootstrapping and managing Kubernetes clusters, particularly for self-provisioned clusters. Kubeadm provides commands like `kubeadm upgrade plan` and `kubeadm upgrade apply` to systematically upgrade the control plane and worker nodes. It simplifies the process of upgrading kubeadm-provisioned clusters.

Kubernetes Tools: Various Kubernetes deployment tools such as Kops, Kubespray, Rancher, and others provide their own mechanisms for cluster upgrades. These tools typically offer automation and specific commands for upgrading the cluster. For example, Kops provides the `kops upgrade cluster` and `kops rolling-update` commands to handle the upgrade process.

Cloud Provider Upgrades: Managed Kubernetes services offered by cloud providers, such as Amazon EKS, Azure AKS, and Google GKE, often handle control plane upgrades transparently. The cloud provider automatically manages the upgrade process, including the control plane components. As a user, you only need to update the node machine images to the desired version.

Blue-Green Deployment: The blue-green deployment approach involves creating a parallel "green" cluster with the desired version while the existing "blue" cluster is still running. Once the green cluster is ready, you switch traffic over to it, ensuring minimal downtime. After verifying the green cluster's stability, you can delete the old blue cluster. This method allows for a smooth transition and rollback option if any issues arise.

Kubernetes does support the last three minor versions for 9 months and provides patches for security and bug fixes during that time

V 1.22 — un-Supported | V 1.23 — Supported | V 1.24 — Supported | V 1.25 — Supported | V 1.26 — Latest

when updating Kubernetes it is generally recommended to update only one minor version at a time, Minor version updates are meant to be backwards compatible. So going from 1.x to 1.x+1 should work smoothly

For production Kubernetes clusters, the general recommendation is to stay within 1 minor version of the latest stable Kubernetes release.

The maximum amount of difference that can exist between k8s components

- Control plane components: 0 versions (identical)
- kubelet/kubectl: Up to 2 minor versions behind
- etcd: Up to 1 minor version behind API server

Kubernetes releases its versions based on semantic versioning

v 1.25.3

MAJOR MINOR PATCH

Features Functionalities — Bug fixes

How to Upgrade Kubernetes Cluster Using Kubeadm?

Step 1: Prepare for the Upgrade

Before upgrading, it is important to review the release notes and documentation for the target Kubernetes version. Check for any specific requirements or considerations.

https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/

recommended to perform upgrades on a test cluster before upgrading a production cluster to ensure that the process goes smoothly and without any issues

Back up any critical data and configurations, including etcd data, you maybe need to roll back the upgrade.

```
ETCDCTL_API=3 etcdctl snapshot save snapshot.db \
  --endpoints=$ENDPOINTS \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

Step1: Determine which version to upgrade to

My current version is 1.25.3 and we will be upgrading it to one higher version, ie, 1.26.7

```
# Find the latest 1.26 version in the list.
# It should look like 1.26.x-00, where x is the latest patch
```

Step 2: Upgrade Control Plane Nodes

Upgrade the control plane components (API server, controller manager, and scheduler) and etcd (if applicable) on each control plane node one by on

Typically, this involves running a series of commands with `kubeadm` to upgrade the control plane components.

C:Analyzes the current state of the cluster and generates a plan for upgrading the control plane components to a newer version of Kubernetes.

A. Drain the control plane node

```
kubectl drain <control-plane-node-name> --ignore-daemonsets
```

B.Upgrade kubeadm ("Skip this step if you want to update to the latest patch in this minor version.")

```
sudo apt-mark unhold kubeadm
sudo apt-get upgrade -y kubeadm=1.26.7-00
sudo apt-mark hold kubeadm
```

The `apt-mark` command in the operating system can be used to label packages and update only the operating system without changing their version.

C. Plan the upgrade

```
sudo kubeadm upgrade plan
```

Upgrade to the latest version in the v1. series:

COMPONENT	CURRENT	TARGET
kube-apiserver	v1.25.3	v1.26.7
kube-controller-manager	v1.25.3	v1.26.7
kube-scheduler	v1.25.3	v1.26.7
kube-proxy	v1.25.3	v1.26.7
CoreDNS	v1.9.1	v1.9.3
etcd	3.5.4-0	3.5.6-0

You can now apply the upgrade by executing the following command:
kubeadm upgrade apply v1.26.7

D. Perform the upgrade

```
sudo kubeadm upgrade apply v1.26.7
```

E.Upgrade kubelet and kubectl

```
apt-mark unhold kubelet kubectl
sudo apt-get upgrade -y kubelet=1.26.7-00
sudo apt-get upgrade -y kubectl=1.26.7-00
sudo apt-mark hold kubelet kubectl
```

F. Restart kubelet and Uncordon the node

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
kubectl uncordon <control-plane-node-name>
```

Step3: Upgrade Worker Nodes

Upgrade the worker nodes one by one. This can be done by draining and cordoning each node, upgrading the necessary components, and then uncordoning the node.

The upgrade process for worker nodes typically involves upgrading the kubelet, kube-proxy, and any other relevant components.

A. Drain the control plane node

```
kubectl drain <worker-node-name> --ignore-daemonsets
```

B. Upgrade kubeadm, kubelet

```
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.26.7-00 && \
apt-mark hold kubeadm
```

C.Upgrade the k8s configuration

```
sudo kubeadm upgrade node
```

D.Upgrade kubelet and kubectl

```
apt-mark unhold kubelet kubectl && \
apt-get update && \
apt-get install -y kubelet=1.27.x-00 && \
apt-get install -y kubectl=1.27.x-00 && \
apt-mark hold kubelet kubectl
```

E. Restart kubelet and Uncordon the node

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
kubectl uncordon <worker-node-name>
```

Worker nodes upgrade strategies

- "All at once": In this strategy, all worker nodes are upgraded at the same time. This approach can be faster than other strategies, but it also carries the highest risk of causing downtime if something goes wrong during the upgrade
- "+1/-1": This strategy involves upgrading one worker node at a time, starting with adding a new node with the updated Kubernetes version, followed by removing an old node with the old version. This process is repeated until all worker nodes have been upgraded. This strategy minimizes the risk of downtime while still allowing for a relatively quick upgrade.
- "-1/+1": This strategy is similar to "+1/-1", but it involves removing an old node first and then adding a new node with the updated Kubernetes version. This strategy carries a slightly higher risk of downtime because there may be fewer worker nodes available during the upgrade process, which could result in an overload on the remaining nodes and potentially cause them to become not ready

Step 4: Verify Cluster Health

After upgrading all the control plane and worker nodes, you should verify the health of the cluster.

Check the status of the control plane components using commands like `kubectl get nodes` and `kubectl get pods -n kube-system`.

Step 5: Update Kubernetes Objects

Some Kubernetes objects, such as Deployments or StatefulSets, may need to be updated to take advantage of new features or changes in the upgraded version

Backup & Restore Methods

It's important to regularly back up to ensure that your k8s cluster can be easily restored in the event of a failure or data loss. Additionally, it's important to test your backup and restore processes to ensure that they are working properly and that you can recover from any issues that may arise.

When designing a backup strategy for a Kubernetes cluster, it's crucial to back up both the application data and the cluster configuration.

Cluster configuration

Cluster configuration includes all the Kubernetes objects and resources that configure your cluster and applications

etcd data: The cluster state and metadata in Kubernetes are stored in etcd. To ensure cluster recovery, it's crucial to back up the etcd data. This can be achieved either by taking periodic snapshots of the etcd database or by implementing a backup solution specifically designed for etcd, such as etcdctl or Velero.

Kubernetes manifests: includes all the Kubernetes objects and resources that configure your cluster and applications. This includes things like deployments, services, configmaps, and etc.These resources are usually defined as code, for example in YAML or JSON files. Because they are code, a good practice is to store them in a version control system like Git. This gives you a history of changes and allows you to revert to a previous state if something goes wrong.

You can backup Kubernetes resources using etcdctl command-line

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: etcd-backup
spec:
  schedule: "0 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: etcd-backup
            image: quay.io/coreos/etcd:v3.5.0
            command:
            - /bin/sh
            - -c
            - |
              ETCDCTL_API=3 etcdctl snapshot save /backup/k8s/etcd-snapshot.db \
              --endpoints=<ETCD-endpoints> \
              --cacert=/etc/kubernetes/pki/etcd/ca.crt \
              --cert=/etc/kubernetes/pki/etcd/server.crt \
              --key=/etc/kubernetes/pki/etcd/server.key
            volumeMounts:
            - name: etcd-certs
              mountPath: /etc/kubernetes/pki/etcd
            - name: backup
              mountPath: /backup
          volumes:
          - name: etcd-certs
            secret:
              secretName: etcd-certs
          - name: backup
            persistentVolumeClaim:
              claimName: backup-pvc
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: backup-pvc
spec:
  storageClassName: <storage-class>
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

This manifest sets up a CronJob that runs an etcd backup job every hour, using the etcdctl command-line tool inside a container to create a snapshot of the etcd database and save it to the specified path. It mounts the etcd certificates and a PersistentVolumeClaim for storing the backup.

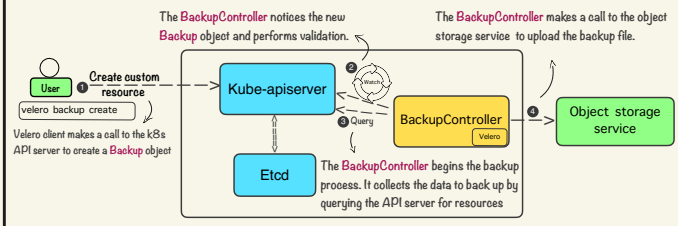
You can backup Kubernetes resources using Velero

Once Velero is installed, you can create a backup by running the following command:

```
velero backup create <backup-name>
```

By default, Velero will back up all resources in all namespaces. If you want to back up only certain namespaces or resources, you can specify them with the --include-namespaces and --include-resources flags, respectively

```
velero backup create <backup-name> --include-namespaces my-namespace \
--include-resources deployments,pods
```



To restore a backup, run the following command:

```
velero restore create --from-backup <backup-name>
```

By default, Velero will restore all resources in the backup to their original namespaces. If you want to restore only certain namespaces or resources, you can specify them with the --include-namespaces and --include-resources flags, respectively

```
velero restore create --from-backup <backup-name> --include-namespaces my-namespace \
--include-resources deployments,pods
```

you can also automate the backup process with Velero

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: velero-backup
spec:
  schedule: "0 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: velero
            image: velero/velero:v1.7.0
            command:
            - /velero
            args:
            - backup
            - create
            - my-backup
            volumeMounts:
            - name: cloud-credentials
              mountPath: /credentials
            env:
            - name: AWS_ACCESS_KEY_ID
              valueFrom:
                secretKeyRef:
                  name: cloud-credentials
                  key: aws_access_key_id
            - name: AWS_SECRET_ACCESS_KEY
              valueFrom:
                secretKeyRef:
                  name: cloud-credentials
                  key: aws_secret_access_key
          volumes:
          - name: cloud-credentials
            secret:
              secretName: cloud-credentials
```

To restore an etcd backup using etcdctl

1. Ensure that the Kubernetes API server is not active or stopped

```
sudo mv /etc/kubernetes/manifests/kube-api.yaml another-path
```

Or

```
sudo systemctl stop kube-apiserver
```

2. change the path of the ETCD data directory to var /var/lib/etcd-from-backup/, you need to edit the manifest file and update the relevant volume and hostPath specifications

```
/etc/kubernetes/manifests/etcd.yaml
volumes:
- name: etcd-data
  hostPath:
    path: /var/lib/etcd-from-backup/
```

3. Use the etcdctl to restore the backup

```
etcdctl snapshot restore /backup/k8s/etcd-snapshot.db \
--data-dir=/var/lib/etcd-from-backup \
--initial-cluster= etcd01= kubemaster-1=https://192.168.100.11:2380 \
--initial-advertise-peer-urls https://192.168.100.11:2380 \
--initial-cluster-token=<ETCD-initial-cluster-token> \
--name=kubemaster-1
```

Application data

Refers to the actual data produced and managed by the applications running on your k8s cluster. This could include databases, user-generated content, logs, and anything else that your applications are producing or manipulating

There are several strategies you can follow to backup this data:

Database Backups: If you're using a database in your application, it's likely that the database itself has backup functionality. For example, you can create a dump of a MySQL database or a snapshot of a MongoDB database.

Backup Sidecars: Another approach is to use a sidecar container in your pods specifically for managing backups. This container would be responsible for regularly creating backups and sending them to a remote location

Volume Snapshots: Kubernetes volume snapshots provide a standardized way to create copies of the content of persistent volumes at a point in time, without creating new volumes.

To create a VolumeSnapshot in Kubernetes, follow the steps below

Ensure that you have the necessary prerequisites in place

Cluster must have volume snapshot CRDs, and snapshot controller deployed on it for this to work

```
kubectl get crds | grep snapshot.storage.k8s.io
```

CSI driver and storage class that support volume snapshots

Create a VolumeSnapshot of the desired PVC

Define a VolumeSnapshot object that references the PVC you want to snapshot

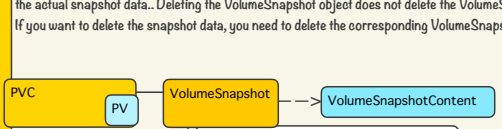
Verify VolumeSnapshotContent was created

Check the status of the VolumeSnapshot to ensure it is created successfully

```
kubectl describe volumesnapshot <snapshot-name>
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-hostpath-sc
```

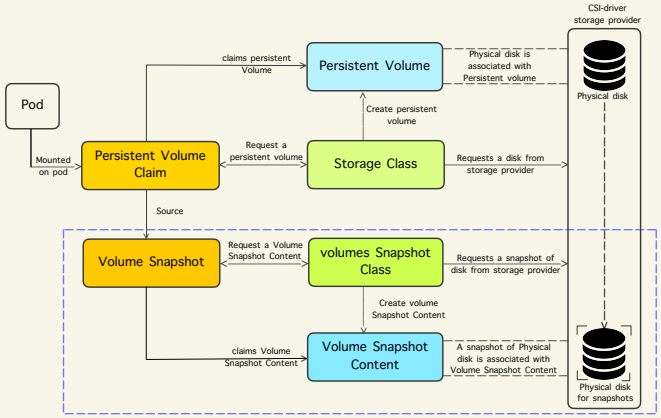
When a VolumeSnapshot object is created, the VolumeSnapshotClass provisions a VolumeSnapshotContent to hold the actual snapshot data.. Deleting the VolumeSnapshot object does not delete the VolumeSnapshotContent object. If you want to delete the snapshot data, you need to delete the corresponding VolumeSnapshotContent object




```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-sc
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
```

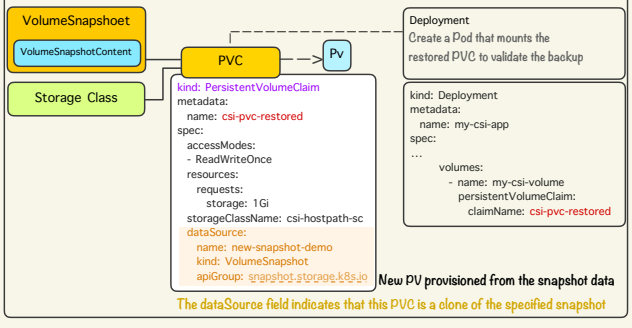
```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-demo
spec:
  volumeSnapshotClassName: csi-hostpath-sc
  source:
    persistentVolumeClaimName: csi-pvc
```

The VolumeSnapshotClass defines the snapshotter/provisioner that will be used to take snapshots and parameters like retention policy, etc. it enables dynamic provisioning of snapshots, just like a StorageClass allows dynamic provisioning of volumes



When you create a VolumeSnapshot object, it triggers the storage provider to create a snapshot of the underlying storage volume. The snapshot is represented by the VolumeSnapshotContent object

To restore a snapshot, create a new PVC based on a VolumeSnapshotContent. This results in a new PV with data populated from the snapshot



The dataSource field indicates that this PVC is a clone of the specified snapshot

Security

Kubernetes uses a combination of secure network channels, authentication and authorization mechanisms, network policies, and container security features to ensure that all communication within the cluster is authenticated, encrypted, and secure. These mechanisms help to protect the cluster against unauthorized access, data breaches, and other security threats, and provide a reliable and secure platform for deploying and managing containerized applications.

Secure network channels: Kubernetes uses secure network channels to ensure that all communication within the cluster is encrypted and secure. These channels are established using Transport Layer Security (TLS) certificates, which provide a secure way to authenticate the identity of different components and encrypt all data that is transmitted between them.

In Kubernetes, many of the components use mutual TLS (Transport Layer Security) authentication for secure communication between each other. This method involves each component having its own certificate (crt) and private key (key) that are used to authenticate and encrypt communication when communicating with other components.

The cluster's certificate authority (CA) is responsible for issuing and managing certificates used for authentication and encryption within the cluster. the CA is typically implemented as a component within the Kubernetes control plane, and is responsible for generating and managing the cluster's root certificate and private key. These are used to sign and issue certificates for different components within the cluster, such as nodes, API servers, and users.

/etc/kubernetes/ is a directory that contains Kubernetes configuration files. usually used for defining settings related to the k8s components

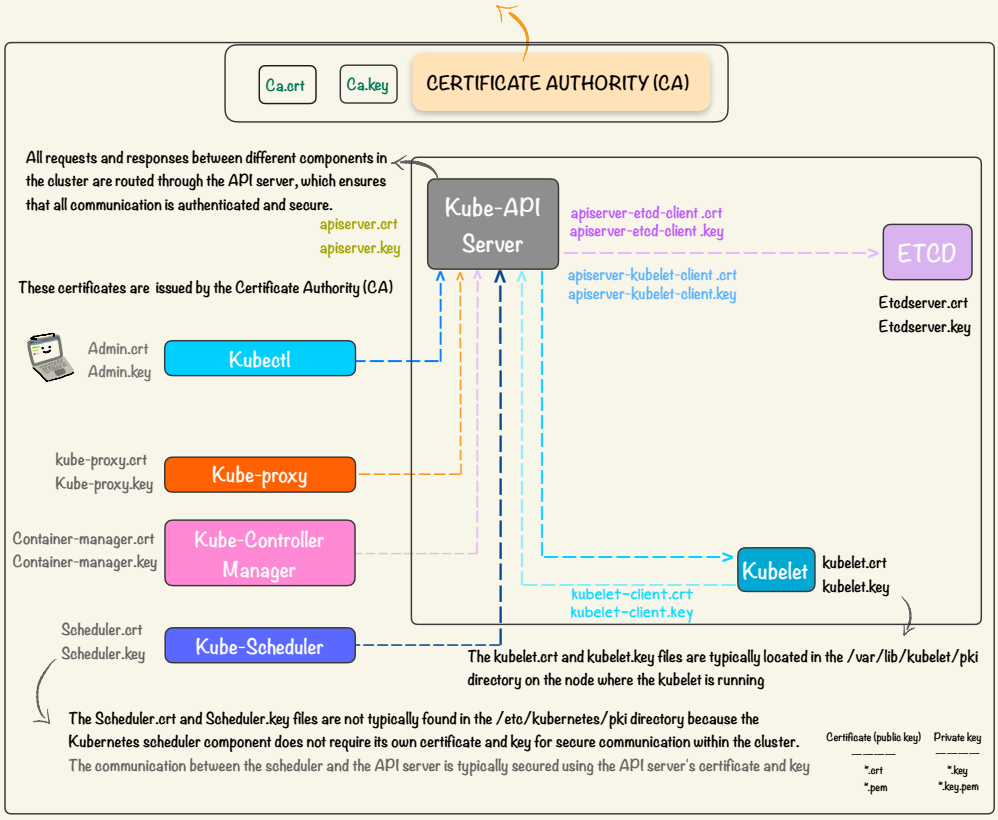
```
arye@kubemaster-1:/etc/kubernetes$ ll
-rw-------  1 root root 5450 May 18 09:34 admin.conf
-rw-------  1 root root 5486 May 18 09:34 controller-manager.conf
-rw-------  1 root root 1886 May 18 09:35 kubelet.conf
drwxr-xr-x  2 root root 4096 May 18 10:30 manifests/
drwxr-xr-x  3 root root 4096 May 18 09:34 pki/
-rw-------  1 root root 5438 May 18 09:34 scheduler.conf
```

These configuration files are essential for the proper functioning of the various k8s components. They contain settings such as the API server address, authentication and authorization information, and other component-specific configurations

admin.conf: This file contains the configuration for the Kubernetes cluster administrator, holding the necessary credentials and cluster information to interact with the cluster using the kubectl command-line tool

The /etc/kubernetes/pki directory is a directory used by Kubernetes to store the public key infrastructure (PKI) materials, such as certificates and keys, that are used to secure communication between the different components of the Kubernetes cluster.

```
arye@kubemaster-1:/etc/kubernetes/pki$ ll
-rw-r--r-- 1 root root 1090 May 18 09:34 apiserver-etcd-client.crt
-rw------- 1 root root 1679 May 18 09:34 apiserver-etcd-client.key
-rw-r--r-- 1 root root 1099 May 18 09:34 apiserver-kubelet-client.crt
-rw------- 1 root root 1675 May 18 09:34 apiserver-kubelet-client.key
-rw-r--r-- 1 root root 1229 May 18 09:34 apiserver.crt
-rw------- 1 root root 1679 May 18 09:34 apiserver.key
-rw-r--r-- 1 root root 1025 May 18 09:34 ca.crt
-rw------- 1 root root 1679 May 18 09:34 ca.key
drwxr-xr-x 2 root root 4096 May 18 09:34 etcd/
-rw-r--r-- 1 root root 1038 May 18 09:34 front-proxy-ca.crt
-rw------- 1 root root 1679 May 18 09:34 front-proxy-ca.key
-rw-r--r-- 1 root root 1058 May 18 09:34 front-proxy-client.crt
-rw------- 1 root root 1675 May 18 09:34 front-proxy-client.key
-rw------- 1 root root 1675 May 18 09:34 sa.key
-rw------- 1 root root  451 May 18 09:34 sa.pub
```



Certificates generated by kubeadm expire after 1 year and will need to be renewed. kubeadm provides a simple command to renew all certificates

To renew all the certificates in a k8s cluster with kubeadm, you can use the kubeadm certs renew command with the all option

```
root@kubemaster-1 ( /etc/kubernetes):
kubeadm certs renew all
```

This will renew the following certificates:
- etcd server and peer certificates
- API server certificate
- Front proxy client certificate
- Controller manager client certificate
- Scheduler client certificate

Note: kubelet.conf is not included in the list above

you can check the expiration dates of the certificates

```
arye@kubemaster-1:~$ sudo kubeadm certs check-expiration
[check-expiration] Reading configuration from the cluster...
[check-expiration] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
CERTIFICATE                EXPIRES                  RESIDUAL TIME   CERTIFICATE AUTHORITY   EXTERNALLY MANAGED
admin.conf                 May 18, 2023 09:34 UTC   341d                                    no
apiserver                  May 18, 2023 09:34 UTC   341d            ca                      no
apiserver-etcd-client      May 18, 2023 09:34 UTC   341d            etcd-ca                 no
apiserver-kubelet-client   May 18, 2023 09:34 UTC   341d            ca                      no
controller-manager.conf    May 18, 2023 09:34 UTC   341d                                    no
etcd-healthcheck-client    May 18, 2023 09:34 UTC   341d            etcd-ca                 no
etcd-peer                  May 18, 2023 09:34 UTC   341d            etcd-ca                 no
etcd-server                May 18, 2023 09:34 UTC   341d            etcd-ca                 no
front-proxy-client         May 18, 2023 09:34 UTC   341d            front-proxy-ca          no
scheduler.conf             May 18, 2023 09:34 UTC   341d                                    no

CERTIFICATE AUTHORITY.     EXPIRES                  RESIDUAL TIME   EXTERNALLY MANAGED
ca                         May 15, 2032 09:34 UTC   9y              no
etcd-ca                    May 15, 2032 09:34 UTC   9y              no
front-proxy-ca             May 15, 2032 09:34 UTC   9y              no
```

you can use the following command to display the details of a certificate file in a human-readable format:

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

It is advisable to backup your certificates and configuration files before executing the command

/etc/kubernetes/pki/*.* | /etc/kubernetes/*.conf | ~/.kube/config

After running the command you should restart the control plane Pods

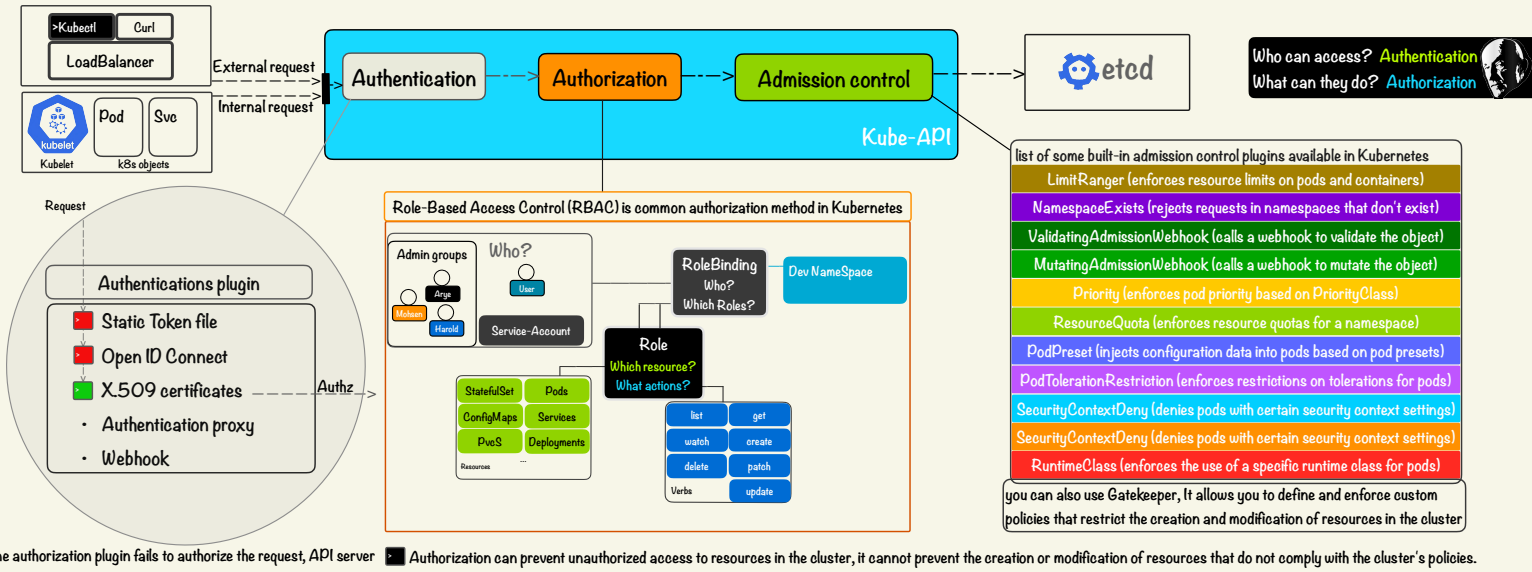
Static Pods are managed by the local kubelet and not by the API Server, thus kubectl cannot be used to delete and restart them. To restart a static Pod you can temporarily remove its manifest file from /etc/kubernetes/manifests/

kubeadm can renews all the certificates during control plane upgrade.

This feature is intended to address straightforward scenarios. If you don't have specific requirements regarding certificate renewal and regularly perform Kubernetes version upgrades (with less than a year between each upgrade), kubeadm will handle the process to ensure your cluster remains up to date and reasonably secure.

Authentication & Authorization

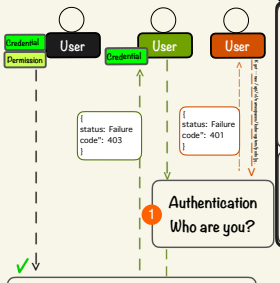
When a client (such as kubectl or a custom application) sends an API request to Kubernetes, the request goes through several steps before it is processed and a response is sent back to the client



if one authorization plugin fails to authorize the request, API server try another plugin until it finds one that can authorize the request

Authorization can prevent unauthorized access to resources in the cluster, it cannot prevent the creation or modification of resources that do not comply with the cluster's policies. Admission control helps ensure that only valid and compliant resources are created or updated in the cluster, which can prevent misconfigurations, security vulnerabilities, and other issues

Authentication: Kubernetes uses authentication mechanisms to verify the identity of users and components trying to access the cluster. This ensures that only authorized users and components can access the cluster.



The most significant authentication methods in k8s

Service Account authentication

Service Account authentication is a method that uses Kubernetes Service Accounts to authenticate clients. Each Service Account in the cluster has its own token that is used to authenticate clients. This method is commonly used when a cluster has a large number of clients or when automated processes need to access the Kubernetes API

X.509 client certificates authentication

X.509 client certificates authentication is a method that uses digital certificates to authenticate clients. Each client in the cluster has its own certificate and private key that are used to authenticate and encrypt communication when communicating with other components. This method is commonly used when a cluster has a small number of clients or when strong authentication is required

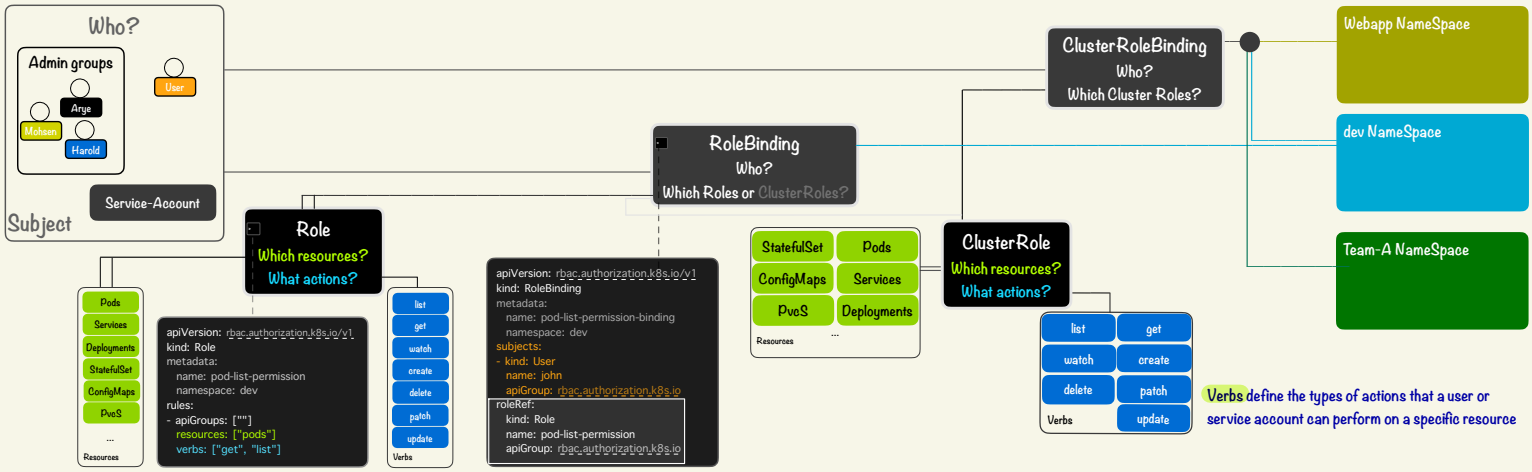
Webhook authentication

Webhook authentication is a method that uses an external HTTP service to authenticate clients. This method is commonly used when a cluster needs to integrate with acustom authentication system. The external HTTP service receives authentication requests from the Kubernetes API server and returns a response indicating whether the client is authenticated or not

2 **Authorization**
an authenticated user can execute
What action on which resources

Authorization: Kubernetes uses authorization mechanisms to determine what actions a user or component can perform within the cluster. This ensures that users and components have access only to the resources they are authorized to access.

Role-Based Access Control (RBAC): RBAC is a security mechanism in Kubernetes that allows you to control access to resources based on the user's role and permissions. in RBAC, you define roles and cluster roles that specify a set of permissions, such as read, write, or delete, for a particular set of resources. You then create role bindings and cluster role bindings that associate roles and cluster roles with users, groups, or service accounts.



Role is a set of permissions that define what actions are allowed on specific resources within a namespace.

RoleBinding is a mechanism for binding a role to a ServiceAccount,a user or group of users within a namespace. Role bindings are used to grant specific permissions to users or groups of users by assigning them to a particular role

ClusterRoles: A ClusterRole is similar to a Role, but it applies to the entire cluster instead of a single namespace. ClusterRoles can be used to grant permissions for cluster-scoped resources (e.g. Nodes) or for resources in all namespaces.

ClusterRoleBinding is cluster-scoped and apply to all namespaces

Roles and ClusterRoles can be either custom or built-in

K get clusterrole

- Built-in Roles and ClusterRoles are predefined by k8s. These built-in roles are designed to provide a set of default permissions for managing Kubernetes resources
- Custom Roles and ClusterRoles are created by users to define their own set of permissions for managing Kubernetes resources

some built-in ClusterRole

- ClusterAdmin: This role is intended to be used by administrators who need full access to all resources in the cluster. It grants permissions to perform any action on any resource in any namespace.
- admin: This ClusterRole provides full access to manage resources in a specific namespace, including the ability to create, update, and delete resources
- system:controller:*, system:node:*: These ClusterRoles provide permissions for k8s controllers and nodes to manage resources in the cluster

```
kubectl describe clusterrole cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  *.*        []                 []              [*]
             [*]                []              [*]
```

The 'system:node' ClusterRole is used to define the set of permissions for nodes in the cluster. This ClusterRole is typically used to grant permissions to the kubelet, to perform actions on various resources related to nodes, including nodes themselves, pods, and service accounts

Admission control: Kubernetes uses a Admission control mechanism for enforcing rules and policies on k8s resources before they are created or updated in the cluster. This can include validating the structure and content of resource manifests, applying default values, and enforcing constraints on resource usage. There are two types of admission control plugins:

Validating admission plugins: These plugins validate the request object without modifying it. They can reject the request if it doesn't meet the required criteria.

Mutating admission plugins: These plugins can modify the request object, as well as validate it. They are executed before the validating admission plugins

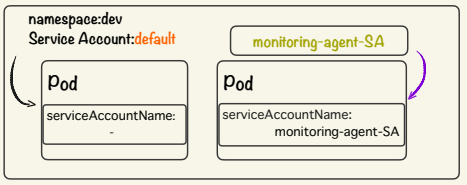
Service Account

Service accounts in Kubernetes are non-human accounts that provide a unique identity for system components and application pods. These accounts are namespace-specific objects managed within the Kubernetes API server. By default, each Kubernetes namespace includes a service account called "default" which has no special roles or privileges assigned to it. In earlier versions of Kubernetes prior to 1.24, when a service account was created, an associated token would be automatically generated and mounted within the pod's file system. However, from Kubernetes 1.24 onwards, the automatic token generation has been discontinued, and tokens must be acquired through the TokenRequest API or by creating a Secret API object, allowing the token controller to populate it with a service account token.

`kubectl create namespace <namespace_name>` → automatically create a default service account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: < namespace_name >
```

Every kubernetes namespace has a default service account named default once being created



If a pod is created without specifying a service account, it will use the default ServiceAccount, default Service Account has limited permissions, but If you need to grant your pod more permissions, you can create a custom Service Account with the necessary roles and assign it to the pod.

Creating and Using a Service Account in a Kubernetes Pod.

1 Create a service account

`kubectl create serviceaccount monitoring-agent-SA`

In Kubernetes v1.24 and earlier, when a Service Account is created, a token secret is automatically generated and stored in the same namespace. This token secret is used for authentication and authorization purposes, However, in Kubernetes v1.25 and later, this automatic token creation has been removed. Instead, there are alternative methods for token creation and management. Here are some options:

Secret API object When you create a Secret with the annotation kubernetes.io/service-account.name and specify a ServiceAccount name, the token controller in k8s will automatically populate the Secret with a service account token associated with the referenced ServiceAccount. so you don't need to manually generate or provide the token for the Secret.The token controller takes care of generating and populating the token for you

```
apiVersion: v1
kind: Secret
metadata:
  name: monitoring-agent-SA-token
  annotations:
    kubernetes.io/service-account.name: monitoring-agent-SA
type: kubernetes.io/service-account-token
```

TokenRequest TokenRequest is an API resource in k8s that allows you to request a token for a specific Service Account. It offers a way to dynamically generate short-lived tokens for authentication and authorization purposes. The TokenRequest object has several important fields, with the audience field being one of them. The `audience` field specifies the intended recipient(s) or target audience for the requested token, defining the authorized users or services for token usage. Here are some examples of audiences that can be specified in the audience field

- rbac.authorization.k8s.io: For Role and ClusterRole operations
- metrics.k8s.io: For Metrics API access
- authentication.k8s.io: This specifies use by authentication methods and operators like kubelet
- storage.k8s.io: For Storage operations
- api: This specifies that the token is intended for use against the Kubernetes API server. API access given to service accounts is enforced by this audience

```
apiVersion: authentication.k8s.io/v1
kind: TokenRequest
metadata:
  name: monitoring-agent-SA-token
spec:
  audiences:
  - rbac.authorization.k8s.io
  serviceAccountName: monitoring-agent-SA
```

2 Grant permissions to the ServiceAccount

Create a Cluster Role that grants the necessary permissions for SA and Create a Role Binding that associates the Service Account with the Cluster Role
Code creates a ClusterRole that grants permissions to retrieve and list information about pods and nodes in the k8s cluster using the "get", "list", and "watch" verbs

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring-agent-role
rules:
- apiGroups: [""]
  resources: ["pods", "nodes"]
  verbs: ["get", "list", "watch"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: monitoring-agent-role-binding
subjects:
- kind: ServiceAccount
  name: monitoring-agent-SA
roleRef:
  kind: ClusterRole
  name: monitoring-agent-role
  apiGroup: rbac.authorization.k8s.io
```

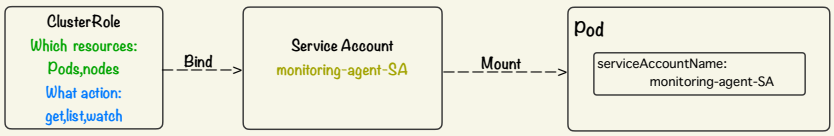
To check the permissions of a service account in Kubernetes, execute the following command to list all the available permissions granted to the monitoring-agent-SA Service Account in the default namespace.

```
kubectl auth can-i --list --as=system:serviceaccount:default:monitoring-agent-SA
```

The --list flag is used to list all the actions and resources that the service account has access to, and the --as flag is used to specify the service account to check the permissions for.

3 Mount the service account token into a pod

When you specify the serviceAccountName field in the Pod spec, Kubernetes mounts the secret containing the Service Account token as a volume in the Pod.
The volume is mounted at /var/run/secrets/kubernetes.io/serviceaccount, and the Service Account token is stored in the token file inside this volume




```
apiVersion: v1
kind: Pod
metadata:
  name: monitoring-agent
spec:
  serviceAccountName: monitoring-agent-SA
  containers:
  - name: monitoring-agent
    image: monitoring-agent-image
    args: ["--kubeconfig=/var/run/secrets/kubernetes.io/serviceaccount/token"]
    volumeMounts:
    - name: sa-token
      mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      readOnly: true
  volumes:
  - name: sa-token
    secret:
      secretName: monitoring-agent-SA-token-xxxxx
```

Using a Service Account to Access a Kubernetes Cluster with kubectl

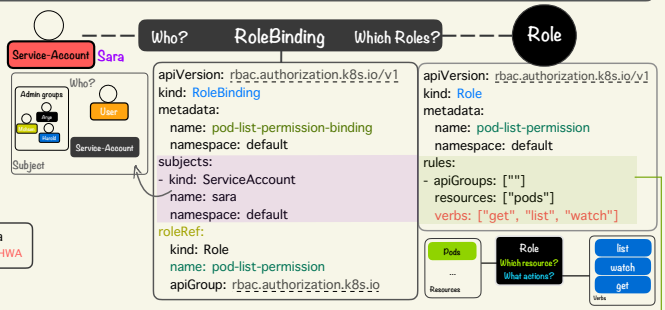
1 Create & Grant permissions to the service account

`kubectl create serviceaccount sara`

`kubectl create role pod-list-permission --verb=get,list,watch --resource=pods --namespace=default`

`kubectl create rolebinding pod-list-permission-binding --role=pod-list-permission --user=sara --namespace=default`

↘ this command creates a role binding in the default namespace that binds the pod-list-permission role to the user sara




```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-list-permission-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: sara
  namespace: default
roleRef:
  kind: Role
  name: pod-list-permission
  apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-list-permission
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

2 Retrieve the Service Account token

You can retrieve the Service Account token or recreate it by running the following commands:

```
Kubectl -n default create token sara
eyJhbGciOiJSUzI1NiIsIm…Lz9APOb2rsWHr9HWA
```

3 Set the token as a credential in kubectl

To add the user sara to this .kube/config file, you would need to add the following code configuration under the users section

```
- name: sara
  user:
    token: eyJhbGciOiJSUzI1NiIsIm…Lz9APOb2rsWHr9HWA
```

After adding this configuration, you would then need to create a new context that uses the sara user and the k8s-cluster-1 cluster

```
- context:
    cluster: k8s-cluster-1
    user: sara
  name: sara-k8s-cluster-1
```

Finally, set the current-context field to the newly created context name

```
current-context: sara-k8s-cluster-1
```

This configuration sets the authentication method for the user sara to use a bearer token (token field) instead of the client certificate and client key used by the arye user

.kube/config file

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBD…RVURS0tLS0tCg==
    server: https://127.0.0.1:42995
  name: k8s-cluster-1
contexts:
- context:
    cluster: k8s-cluster-1
    user: sara
  name: sara-k8s-cluster-1
- context:
    cluster: k8s-cluster-1
    user: arye
  name: arye@k8s-cluster-1
current-context: sara-k8s-cluster-1
kind: Config
preferences: {}
users:
- name: arye
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRQUR…S0tLS0tCg==
    client-key-data: LS0tLS1CRUdJTiBdONibHFxS0t…LS0tCg==
- name: sara
  user:
    token: eyJhbGciOiJSUzI1NiIsIm…Lz9APOb2rsWHr9HWA
```

The "rules" section in a role specifies the permissions granted by the role. The "rules" section is an array of rules, where each rule specifies the resources and operations that are allowed

The "subjects" section specifies the user or group of users to which the role should be bound. A subject can be a user, a group, or a service account.

How to bind a role to multiple users?

```
...
subjects:
- kind: User
  name: mohsen
  apiGroup: rbac.authorization.k8s.io
- kind: User
  name: arye
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: developers
  apiGroup: rbac.authorization.k8s.io
...
```

you can bind a role to multiple users by creating a role binding that specifies multiple users in the "subjects" section

How to specify multiple rules in a Role?

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-viewer
  namespace: my-namespace
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["get"]
```

You can also specify multiple rules in the "rules" section of a role

Using Service Accounts for authentication can be more secure than using user accounts because Service Accounts are automatically created and managed by Kubernetes

API Groups

In k8s, API groups are a way of organizing related resources and operations together. This allows for easier discovery and usage, and also helps to avoid naming conflicts between different resources. When k8s was first introduced, all the resources like Pod, Service, ReplicationController, etc., were all part of a single group, the "core" group, and were accessed at the path /api/v1.As k8s evolved and more resources were added, it became clear that this single group was not scalable. So, the concept of API groups was introduced

Kubernetes uses a versioning scheme to facilitate the evolution of its API. There are three types of versioning in Kubernetes:

Alpha: This is the first stage of the development of a new API. Alpha APIs may be unstable, change significantly after the initial release, and may not even be enabled in your clusters.

Beta: This is the second stage. Beta APIs are well-tested and are enabled by default in your clusters. However, they may still undergo changes, such as in the form of bug fixes or feature enhancements.

Stable: This is the final stage. Stable APIs appear in released software for many subsequent versions

The version of an API group is represented by vXalphaX (e.g., v1alpha1), vXbetaX (e.g., v2beta2), and vX (e.g., v1) for alpha, beta, and stable versions, respectively.

Kubernetes API groups are divided into two categories

Named API Groups

Named API groups are additional API groups introduced to extend the functionality of Kubernetes beyond the core resources. Each named API group focuses on specific features or functionalities and manages specialized resources related to those features. The Named API group is accessed using the /apis endpoint

Core API Group

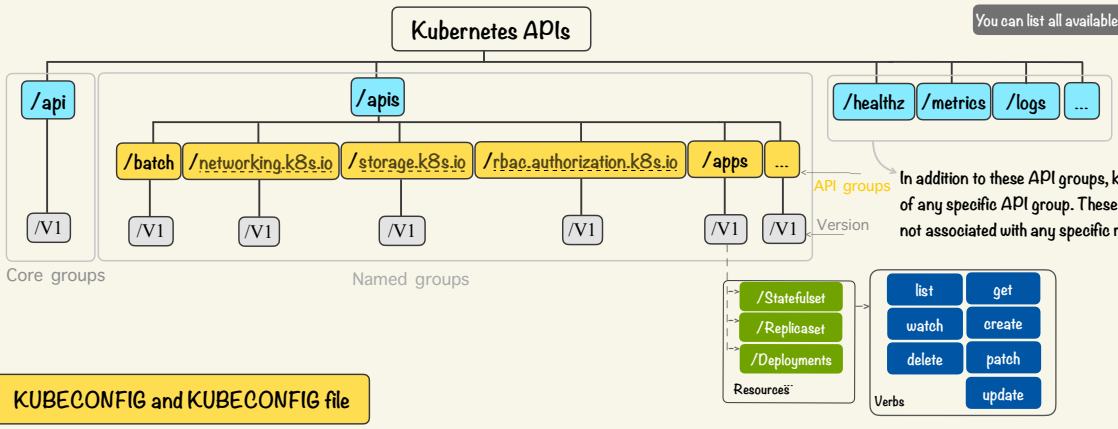
The core API group, also referred to as the "v1" group, contains the essential resources that are fundamental to the functioning of a Kubernetes cluster. It includes resources such as Pods, Services, Namespaces, ConfigMaps, Secrets, and more. The core API group is accessed using the /api/v1 endpoint

Pods · services · configmaps · secrets · namespaces · PV · PVC · ns · nodes · endpoint · events · binding

- **apps**: This group contains resources related to running applications on Kubernetes. It includes Deployment, ReplicaSet, StatefulSet, and DaemonSet.
- **batch**: This group includes resources for batch processing and job-like tasks. It includes Job and CronJob.
- **rbac.authorization.k8s.io**: This group contains the Role, ClusterRole, RoleBinding, and ClusterRoleBinding resources for handling role-based access control (RBAC) in Kubernetes.
- **networking.k8s.io**: This group contains resources related to networking in k8s, such as NetworkPolicy and Ingress.
- **storage.k8s.io**: This group contains resources related to storage, such as StorageClass, VolumeAttachment, and the CSINode driver

New resources are accessed at the path /apis/{group}/{version}.For example, to access the Deployment resource, which is part of the apps group, you would use the path /apis/apps/v1/deployments.

You can list all available API groups and versions in your cluster by running kubectl api-versions



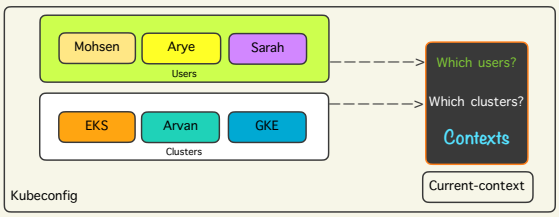
In addition to these API groups, k8s also provides several non-resource endpoints that are not part of any specific API group. These endpoints provide access to information and functionality that are not associated with any specific resource, such as the /healthz, /metrics, and /logs endpoints

KUBECONFIG and KUBECONFIG file

The KUBECONFIG environment variable is used to specify the path to the Kubernetes configuration file, which contains information about the cluster, user, and context used by kubectl and other Kubernetes command-line tools. The KUBECONFIG file can contain multiple contexts, each representing a different cluster and namespace. The KUBECONFIG file is typically stored in the user's home directory at the path ~/.kube/config on Unix-based systems

```
kubectl --kubeconfig=/path/to/my-kubeconfig/my-kubeconfig.yaml get pods
```

This command uses the specified KUBECONFIG file instead of the default ~/.kube/config file



Contexts section defines the mapping between the Kubernetes cluster(s) and the users who can access them. A context includes the cluster and user information, as well as a reference to a default namespace and a name for the context

```
apiVersion: v1
Kind: Config
Clusters:
- name: GKE
  cluster:
    certificate-authority: ca.cert
    servrer: https://k8s-endpoint:6443
contexts:
- name: Arye@GKE
  context:
    cluster: GKE
    user: Arye
    namespace: dev
users:
- name: Arye
  user:
    client-certificate: arye.crt
    client-key: arye.key

current-context: Arye@GKE
```

new.kubeconfig

The default namespace to use for this context.

To create a kubeconfig file using kubectl, you can follow these steps:

Set the cluster details | **Set the user credentials** | **Set the context** | **Use the context**

```
kubectl config set-cluster GKE --server=https://k8s-endpoint:6443 --certificate-authority=ca.crt --embed-certs --kubeconfig new.kubeconfig
```

If embed-certs=false

- Set a cluster entry in kubeconfig
- Cluster name
- Server Address
- K8s CA certificate for TLS verification
- Embeds the certificate data directly in the kubeconfig instead of linking to a file
- kubeconfig file that will be created with this new entry.

```
kubectl config set-credentials Arye --client-key=/path/to/arye.key --client-certificate=/path/to/arye.crt --embed-certs --kubeconfig new.kubeconfig
```

If you use a token

- User name
- user key file
- user certificate file

```
kubectl config set-credentials Arye --token= eyJhbGciOiJSUzIlNiIsIm...Lz9APOb2rsWHr9HWA --kubeconfig new.kubeconfig
```

If you use a client certificate

```
kubectl config set-context Arye@GKE --cluster=GKE --user=Arye namespace=dev --kubeconfig new.kubeconfig
```

- context name
- default namespace

```
kubectl config use-context Arye@GKE
```

you can specify a different kubeconfig file by setting the KUBECONFIG environment variable.

```
export KUBECONFIG=new.kubeconfig
```

Cluster Scope in Kubernetes

In Kubernetes, resources are divided into two categories based on their scope: Namespaced and Cluster-scoped

Namespaced resources: These resources exist and operate within a namespace. They can have different configurations and states in different namespaces

configmaps · Role · PVC · Rolebinding · Deployment · Replicaset · Pods · Services · Jobs

```
kubectl api-resources —namespace=true
```

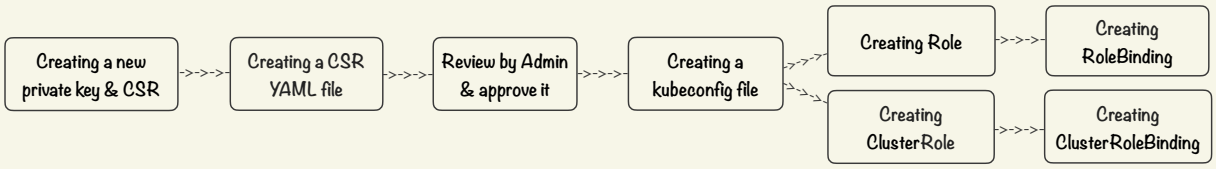
Cluster-scoped resources: These resources exist and operate across the entire cluster. They are not confined to any particular namespace

nodes · PV · Clusterroles · Clusterrolebinding · Namespace

```
kubectl api-resources —namespace=false
```

```
kubectl api-resources
NAME.                SHORTNAMES.   APIVERSION.              NAMESPACED.   KIND
...
endpoint             no            v1                       true          Endpoints
Pods                 po            v1                       false         Pod
Service              svc           v1                       true          Service
deployment           deploy        apps/v1                  true          Deployment
Ingresses            ing           extensions/v1beta1       true          Ingress
...
```

How to create a new admin or developer user account for accessing to a k8s cluster with X.509 ?



1 Creating a new private key & a csr file by new user

Generate a private key for the user using OpenSSL. The private key is used as part of the user's credentials to authenticate with the Kubernetes API server

```
openssl genrsa -out mojtaba.key 2048
```

Create a CSR for the user using the private key

```
openssl req -new -key mojtaba.key -subj "/CN=mojtaba" -out mojtaba.csr
```

The CSR includes the user's identifying information and the public key associated with the private key

2 Creating a new CSR yaml file and Sign the CSR using the Kubernetes CA

Create a CertificateSigningRequest object in Kubernetes that includes the user's CSR and submit the CSR to the Kubernetes cluster

csr-mojtaba.yml

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: mojtaba
spec:
  groups:
    - system:authenticated
  request: LS0tLS1CRUdJTjkKNUlEdC9BWT.....0KLS0tLS1FTkQ...
gQ0VSVEIGSUNBVEUgUkVRVUVTVC0tLS0tCg==
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - client auth
```

```
request: $(cat user-name.csr | base64 -w 0)
```

Or

```
cat mojtaba.csr | base64 -w 0
```

The signerName specifies the Kubernetes CA that will sign the certificate.
The usages field specifies that the certificate will be used for client authentication.

```
kuectl  apply -f csr-mojtaba.yml
```

3 submit the CSR to the Kubernetes cluster and approve it

Once the CSR is submitted, it needs to be approved by a cluster administrator.

```
k get csr
NAME       AGE   SIGNERNAME                            REQUESTOR           CONDITION
mojtaba    33m   kubernetes.io/kube-apiserver-client   kubernetes-admin    Pending
```

```
k describe csr mojtaba
Name:           mojtaba
Labels:         <none>
Annotations:    kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"certificates.k8s.io/
v1","kind":"CertificateSigningRequest","metadata":{"annotations":{},"name":"mojtaba"},"spec":{"groups":
["system:authenticated"],"request":"LS0tLS1CRUdJTjkKNUlEdC9BWTOKLS0tLS1FTkQgQ0VSVEIGSUNBVEUgUkVRVU
VTVC0tLS0tCg==","signerName":"kubernetes.io/kube-apiserver-client","usages":["client auth"]}}

CreationTimestamp:  Sat, 11 Jun 2022 17:51:33 +0000
Requesting User:    kubernetes-admin
Signer:             kubernetes.io/kube-apiserver-client
Status:             Pending
Subject:
        Common Name:          mojtaba
        Serial Number:
        Organization:         StarkWare
        Organizational Unit:  blockchain
        Country:              IL
        Locality:             haifa
        Province:             haifa
Events:  <none>
```

```
k certificate approve mojtaba
```

```
k get csr
NAME       AGE   SIGNERNAME                            REQUESTOR           CONDITION
mojtaba    39m   kubernetes.io/kube-apiserver-client   kubernetes-admin    Approved,Issued
```

This command notifies the Kubernetes CA that the CSR has been approved and requests a signed certificate for the user. The signed certificate is then stored in the status.certificate field of the CertificateSigningRequest object

4 Export the issued certificate from the CertificateSigningRequest.

you retrieve the signed certificate for the user

```
kubectl get csr mojtaba -o jsonpath='{.status.certificate}' | base64 -d >mojtaba.crt
```

5 Create a kubeconfig File for the User

Create a kubeconfig file for the user that includes the cluster details, user credentials, and context. The certificate-authority-data field contains the base64-encoded CA certificate for the Kubernetes cluster.

```
apiVersion: v1
kind: Config

current-context: mojtaba@cka

clusters:
  - name: cka
    cluster:
      server: https://kubemaster-1:6443
      certificate-authority: ca.crt

users:
  - name: mojtaba
    user:
      client-certificate: mojtaba.crt
      client-key: mojtaba.key

contexts:
  - name: mojtaba@cka
    context:
      cluster: cka
      user: mojtaba
      namespace: dev
```

```
apiVersion: v1
kind: Config

current-context: mojtaba@cka

clusters:
  - name: cka
    cluster:
      server: https://kubemaster-1:6443
      certificate-authority-data: <base64-encoded CA certificate data>

users:
  - name: mojtaba
    user:
      client-certificate-data: <base64-encoded client certificate data>
      client-key-data: <base64-encoded client key data>

contexts:
  - name: mojtaba@cka
    context:
      cluster: cka
      user: mojtaba
      namespace: dev
```

To become independent from external files in the configuration, you can use the data field directly within the configuration file

certificate-authority-data

```
cat /etc/kubernetes/pki/ca.crt | base64 -w 0
```

client-certificate-data

```
cat mojtaba.csr | base64 -w 0
```

client-key-data

```
cat mojtaba.csr | base64 -w 0
```

5.1 If you don't want to create a kubeconfig manually, you can create a kubeconfig using kubectl

```
kubectl config set-cluster cka --server=https://kubemaster-1:6443 --certificate-authority=ca.crt --embed-certs --kubeconfig devuser.kubeconfig
kubectl config set-credentials mojtaba --client-key=/path/to/mojtaba.key --client-certificate=/path/to/mojtaba.crt  --embed-certs --kubeconfig devuser.kubeconfig
kubectl config set-context mojtaba@cka --cluster=cka --user=mojtaba  namespace=dev  --kubeconfig devuser.kubeconfig
kubectl config use-context mojtaba@cka
```

6 Set Up Role-Based Access Control (RBAC) for the User

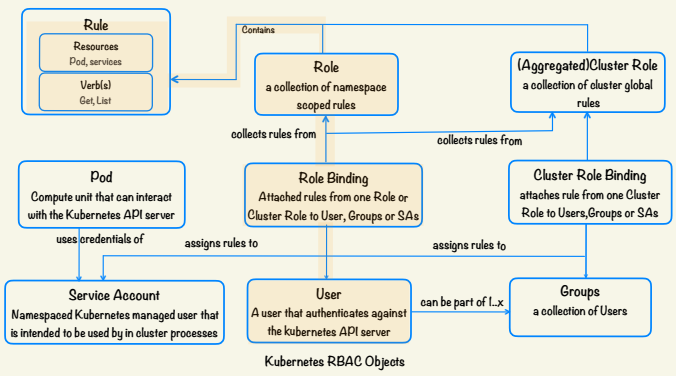
In this final step, you create a role and role binding to grant the user permissions in the Kubernetes cluster

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
  namespace: dev
rules:
  - apiGroups: [ "" ]
    resources: ["pods"]
    verbs: ["list" , "get" , "create" , "update" , "delete"]

  - apiGroups: [ "" ]
    resources: ["configMap"]
    verbs: ["create"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: mojtaba-developer
  namespace: dev
roleRef:
  apiGroup: "rbac.authorization.k8s.io"
  kind: "Role"
  name: "developer"
subjects:
  - apiGroup: "rbac.authorization.k8s.io"
    kind: "User"
    name: "mojtaba"
```

The reason for having two separate rules in the Role definition is that the two resources, "pods" and "configMap", have different permissions requirements



https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/#normal-user

Auditing

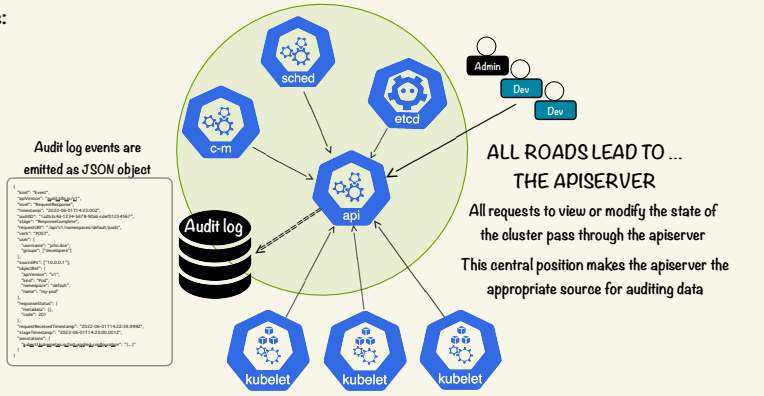
Auditing in Kubernetes refers to the process of recording and analyzing activities that occur on the cluster. This can include actions taken by users, API requests, and changes to objects in the cluster. Auditing provides visibility into the behavior of the cluster and can be used for security, compliance, and troubleshooting purposes

Audit levels in Kubernetes define the verbosity of the recorded events. There are four audit levels:

- **None**: Do not log any events.
- **Metadata**: Log request metadata only (e.g., who, what, where, when).
- **Request**: Log event metadata and request content (excluding the response).
- **RequestResponse**: Log event metadata, request content, and response content.

Memory consumption depends on the audit logging policy

Audit logging increases the memory consumption of the API Server



Audit log events are emitted as JSON object

ALL ROADS LEAD TO ... THE APISERVER

All requests to view or modify the state of the cluster pass through the apiserver

This central position makes the apiserver the appropriate source for auditing data

To enable audit logging in the Kubernetes API server, you need to configure the API server to use a specific audit policy and write audit logs to a file or other destination

1. Edit the API server configuration file, Add the following flags to the spec.containers.command section

```
- --audit-policy-file=/etc/kubernetes/audit/policy.yaml
- --audit-log-path= /var/log/kubernetes/audit/audit.log
- --audit-log-format=json
- --audit-log-maxsize=500
- --audit-log-maxbackup=3
```

The apiserver has some audit logging options:
audit-policy-file: sets the policy file to use
audit-log-*: setting configure log files
audit-webhook-*: settings configure log network endpoints

2. Add a volume and volumeMount to the spec section

```
volumes:
- name: audit-config
  hostPath:
    path: /etc/kubernetes/audit/policy.yaml
    type: File
- name: audit-logs
  hostPath:
    path: /var/log/kubernetes/audit
    type: DirectoryOrCreate
```

3. Add the corresponding volume mounts to the spec.containers.volumeMounts section

```
- name: audit-config
  mountPath: /etc/kubernetes/audit/policy.yaml
  readOnly: true
  subPath: audit-policy.yaml
- name: audit-logs
  mountPath: /var/log/kubernetes/audit
```

4. After the API server restarts and applies the policy.yaml file, you can tail the logs to see the events being recorded

```
tail -f /var/log/kubernetes/audit/audit.log | jq
```

Audit policy is a configuration that defines the rules for what events should be recorded and at what level

/etc/kubernetes/audit/policy.yaml

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
  resources:
  - group: ""
    resources: ["pods", "services"]
- level: Request
  users: ["system:serviceaccount:my-namespace:my-serviceaccount"]
  resources:
  - group: ""
    resources: ["configmaps"]
```

This policy logs metadata for all pod and service operations and logs request content for configmap operations performed by the specified service account

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  - level: Request
    resources:
    - group: ""
      resources: ["pods", "services"]
    verbs: ["create", "update", "delete"]
  - level: Metadata
    resources:
    - group: ""
      resources: ["namespaces", "configmaps", "secrets"]
    verbs: ["create", "update", "delete"]
  - level: None
    resources:
    - group: ""
      resources: ["persistentvolumes", "persistentvolumeclaims"]
```

This policy will log request-level for pod and service creation, update, and deletion, also will log metadata-level events for namespace, configmap, and secret creation, update, and deletion. It will not log events related to persistent volume and persistent volume claim resources.

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:

- level: None
  verbs: ["get", "watch", "list"]

- level: None
  resources:
  - group: "" # core
    resources: ["events"]

- level: None
  users:
  - "system:kube-scheduler"
  - "system:kube-proxy"
  - "system:apiserver"
  - "system:kube-controller-manager"
  - "system:serviceaccount:gatekeeper-system:gatekeeper-admin"

- level: None
  userGroups: ["system:nodes"]

- level: RequestResponse
```

This policy has five rules, each specifying a different level of audit logging:

-> specifies that all "get", "watch", and "list" operations should not be audited at all.

-> specifies that events should not be audited at all.

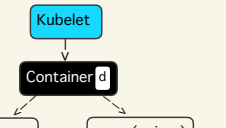
-> specifies these certain system users should not be audited

-> specifies that all users belonging to the "system:nodes" group should not be audited.

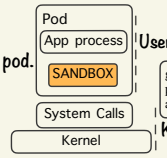
-> specifies that all other operations, including requests and responses, should be audited at the RequestResponse levels

RuntimeClass

RuntimeClass is a Kubernetes feature that allows users to specify different runtime configurations for their containers. One common use case for RuntimeClass is to run containers with different levels of isolation.For example, a user may want to run some containers with a higher level of isolation, while others may not require the same level of security. By defining multiple RuntimeClasses with different runtime configurations, the user can choose the appropriate class for each container.



gVisor is a user-space kernel that provides isolation for containers by intercepting and handling system calls. It can be used with k8s to provide an extra layer of security for your pods. To restrict syscalls for a pod running in k8s, you can use gVisor as the runtime for that pod.



gVisor is an sandboxed container runtime developed by Google. It provides an additional layer of isolation between containerized applications and the host kernel using a technique called "sandboxing"

How to use gVisor

Install gVisor

Configure containerd

Create a RuntimeClass resource

Use the gVisor RuntimeClass in your pod

First, you need to install gVisor on your Kubernetes nodes. You can do this using the runsc binary, which is the gVisor runtime. Download and install the runsc binary on each node:

```
wget https://storage.googleapis.com/gvisor/releases/nightly/latest/runsc
chmod +x runsc
sudo mv runsc /usr/local/bin
```

To use gVisor with Kubernetes, you need to configure the container runtime (e.g., containerd) to use gVisor. Create a configuration file for containerd:

```
sudo mkdir -p /etc/containerd
sudo nano /etc/containerd/config.toml
```

Add this configuration to the `config.toml` file:

```toml
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]
  runtime_type = "io.containerd.runsc.v1"
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc.options]
    BinaryName = "/usr/local/bin/runsc"
    Root = ""
    LogLevel = "info"
    Debug = false
    DebugLogFile = ""
    NoSandbox = false
```

Restart containerd to apply the new configuration:

```
sudo systemctl restart containerd
```

create a `RuntimeClass` resource in your Kubernetes cluster that specifies gVisor as the runtime. Save the following YAML file as `gvisor-runtime-class.yaml`:

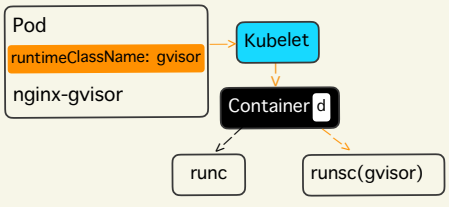
```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

```
kubectl apply pply -f gvisor-runtime-class.yaml
```

To use gVisor for a specific pod, set the `runtimeClassName` field to `gvisor` in the pod spec. Here's an example of a simple Nginx pod that uses gVisor:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-gvisor
spec:
  runtimeClassName: gvisor
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

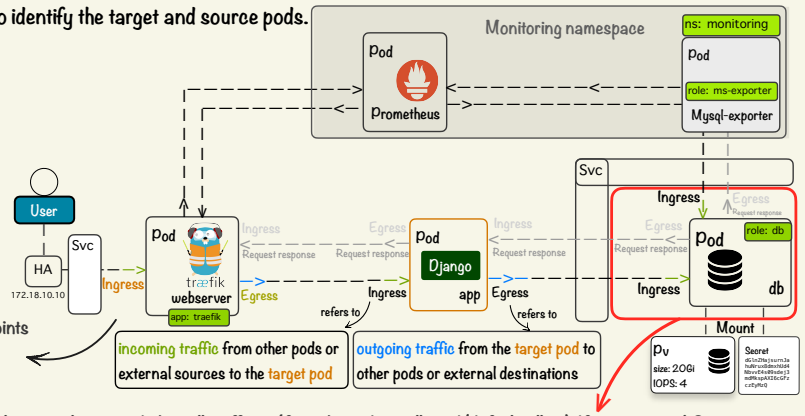
```
kubectl apply pply -f nginx-gvisor-pod.yaml
```



Network policy

Network Policy is a Kubernetes feature that allows you to define rules for ingress and egress traffic between pods inside a cluster. It's a way to implement security and access control at the network level by specifying which pods can communicate with each other, using labels to identify the target and source pods.

features
- Policies are namespace scoped
- Policies are applied to pods using label selectors
- Policy rules can specify the traffic that is allowed to/from pods, namespaces, or CIDRs
- Policy rules can specify protocols (TCP, UDP, SCTP), named ports or port numbers



Network policies are applied to pods rather than services because pods are the network endpoints that actually receive the traffic. Services are not network endpoints and do not receive traffic directly. Instead, they route traffic to the appropriate pods based on their labels.

incoming traffic from other pods or external sources to the target pod

outgoing traffic from the target pod to other pods or external destinations

If no Kubernetes network policies apply to a pod, then all traffic to/from the pod are allowed (default-allow). If one or more k8s network policies apply to a pod, then only the traffic specifically defined in that network policy are allowed (default-deny)

Network policies are like firewall rules for your Kubernetes pods. By default, pods are non-isolated and can accept traffic from any source. When you apply a NetworkPolicy to a pod, that pod becomes isolated and only allows traffic that is permitted by the policy. There are several types of Network Policy rules that can be defined in Kubernetes:

PodSelector: This rule selects a specific set of pods to apply the policy to based on their labels.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: django
    ports:
    - protocol: TCP
      port: 3306
```

Allow ingress traffic from pods in the same namespace

This selects the pods to which the NetworkPolicy applies. In this case, it matches all pods with the label role: db

specifies that the policy only applies to ingress traffic

The "ingress" section specifies the traffic rules that govern inbound traffic to the selected pods. Specifically, it permits traffic from pods labeled with "role: django" to access the selected pods on TCP port 3306.

Ports filed allows you to specify the ports and protocols that are allowed for incoming or outgoing traffic.

NamespaceSelector: This rule selects all the pods in a specific namespace to apply the policy to.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy-namespace
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: ms-exporter
      namespaceSelector:
        matchLabels:
          ns: monitoring
    ports:
    - port: 3306
```

Allow ingress traffic from pods in a different namespace

This rule only allows inbound traffic from pods labeled "role: ms-exporter" in the "ns: monitoring" namespace. Incoming traffic is limited to port number 3306

ExternalEntities: This rule allows you to define specific IP addresses or IP ranges that are allowed to communicate with the selected pods

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: traefik-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: traefik
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.18.0.0/24
  egress:
  - to:
    - podSelector:
        matchLabels:
          role: django
```

specifies that the policy applies to both Ingress and Egress traffic

specifies that pods with the label "app: traefik" can only receive traffic from the IP block "172.18.0.0/24" and can only send traffic to pods with the label "role: django".

Please note that in order to use Network Policies, you must have a CNI (Container Network Interface) that supports them, such as Calico or Weave Net.

Security Context

SecurityContext is a configuration object that defines the security settings for a Pod or a specific container within a Pod. It allows you to set the access control and security-related properties for the containers, including their file system, users, and groups, as well as the capabilities and privileges of the processes running inside the containers.

SecurityContext object can be defined at the Pod level or at the container level, using the securityContext field in the Pod or container specification

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      runAsUser: 1000
      runAsGroup: 2000
      fsGroup: 3000
      readOnlyRootFilesystem: true
```

container will run as the user ID 1000, the group ID 2000, and have its filesystem owned by the group ID 3000. Additionally, the container's root filesystem will be read-only, which can help to improve security by preventing changes to critical system files.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      privileged: true
```

the privileged field is set to true, which means that the container will run in privileged mode

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        drop:
        - CHOWN
      allowPrivilegeEscalation: false
```

the capabilities field is used to specify the Linux capabilities that the container is allowed to use. Here, the container is allowed to use the NET_ADMIN capability, but is not allowed to use the CHOWN capability.

Additionally, the allowPrivilegeEscalation field is set to false, which means that the container is not allowed to escalate privileges beyond what is specified in the SecurityContext object

Image security

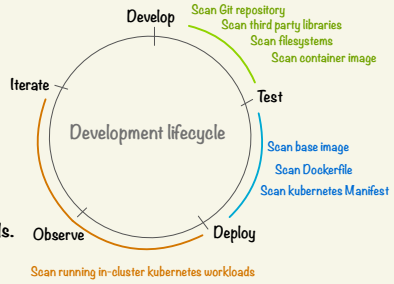
Trivy is a simple and comprehensive vulnerability scanner for containers. It's used to identify vulnerabilities in operating system packages (Alpine, Red Hat Universal Base Image, CentOS, etc.) & application dependencies (Bundler, Composer, npm, yarn, etc.). It's especially useful in the Kubernetes (k8s) environment for scanning container images and ensuring your workloads are secure. Here's how Trivy can be integrated into different stages of Kubernetes deployment:

Pre-deployment Scanning:

Before deploying your workloads, you can use Trivy to scan various resources for vulnerabilities and misconfigurations. Here are some common use cases:

- **Third-party Libraries**: Scan your application's dependencies and libraries for known vulnerabilities.
- **Container Images**: Scan container images for vulnerabilities in the underlying operating system packages and application dependencies
- **Git Repositories**: Analyze your code repositories for secrets, sensitive information, or other security issues.

You can use the Trivy CLI on your local machine or integrate Trivy into your CI/CD pipeline to perform these pre-deployment scans. Trivy will provide you with a list of vulnerabilities and misconfigurations to address before deploying your workloads



Continuous Scanning of Running Workloads:

After deploying your workloads to Kubernetes, it's essential to set up automated and continuous scanning to detect vulnerabilities in your running workloads. Here are the recommended features for this stage:

- **Trivy K8s Command**: Use the trivy kubernetes command to scan Kubernetes Deployments or Namespaces. Trivy will scan the container images used by the running Pods and provide vulnerability reports
- **Trivy Operator**: Deploy the Trivy Operator in your Kubernetes cluster. The Trivy Operator automates the scanning of running workloads by continuously monitoring and scanning container images within the cluster

```
trivy k8s --namespace=kube-system --report=summary deploy
Summary Report for minikube

Workload Assessment
```

Namespace	Resource	Vulnerabilities					Misconfigurations					Secrets				
		C	H	M	L	U	C	H	M	L	U	C	H	M	L	U
kube-system	Deployment/metrics-server								2	8						
kube-system	Deployment/coredns		1					1	3	5						
kube-system	Deployment/logviewer	2							4	11						

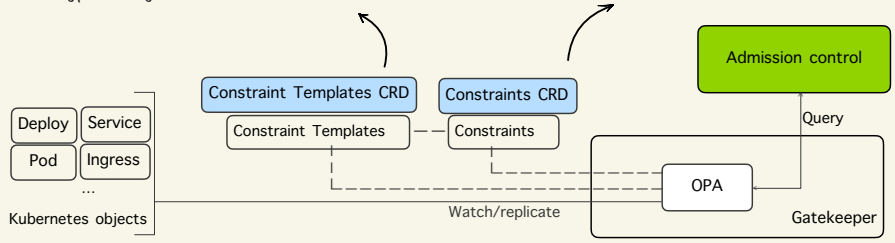
Severities: C=CRITICAL H=HIGH M=MEDIUM L=LOW U=UNKNOWN

Gatekeeper

Kubernetes provides admission controller webhooks as a mechanism to decouple policy decisions from the API server. These webhooks intercept admission requests before they are persisted as objects in k8s, allowing custom logic and policies to be enforced. Gatekeeper was specifically designed to facilitate customizable admission control through configuration, rather than requiring code changes. It brings awareness to the overall state of the cluster, going beyond evaluating a single object during admission. Gatekeeper integrates with Kubernetes as a customizable admission webhook. It leverages the Open Policy Agent (OPA), which is a policy engine hosted by the Cloud Native Computing Foundation (CNCF), to execute policies in cloud-native environments.

Constraint Templates are Kubernetes Custom Resource Definitions (CRDs) that define a set of constraints or policies that can be applied to Kubernetes objects. They act as a template or blueprint for creating individual Constraints. A Constraint Template defines the structure, parameters, and validation rules for a specific type of constraint that can be applied to Kubernetes resources.Constraint Templates allow you to define reusable policies that can be applied to multiple resources across your cluster. They provide a way to centralize and standardize the enforcement of constraints

Constraints are instances of Constraint Templates. They are created based on the defined template and applied to specific Kubernetes resources. Constraints enforce policies by validating the resources against the defined rules and conditions in the Constraint Template. If a resource violates any of the defined constraints, it is considered non-compliant



When a user tries to create/update a resource in the cluster, the request first goes to the gatekeeper (as an admission webhook). Gatekeeper checks if the resource satisfies all the defined constraints and rejects the request if any policy is violated

Enforcing Resource Limits and Requests for Pods using Gatekeeper

To enforce a policy where all Pods must have resource limits and requests set using Gatekeeper, you would create a ConstraintTemplate and then a Constraint using that template. Here's how you can do it:

1. Create a ConstraintTemplate, which defines the schema and the Rego logic for the policy. The ConstraintTemplate specifies that the Pods must have resource limits and requests

```yaml
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredresources
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredResources
      validation:
        openAPIV3Schema:
          properties:
            resources:
              type: array
              items: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredresources

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.limits.memory
          msg := sprintf("container <%v> has no memory limit", [container.name])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.limits.cpu
          msg := sprintf("container <%v> has no CPU limit", [container.name])
        }
```

2. Create a Constraint based on the ConstraintTemplate you defined. The Constraint specifies the name and the kind of resources to which the policy applies

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredResources
metadata:
  name: pod-must-have-limits
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

3. After applying the Constraint, any new Pods that do not have resource limits and requests will be rejected by the Gatekeeper admission webhook. Existing pods will not be affected by this policy

this ConstraintTemplate is defining a constraint that requires all containers in Kubernetes resources to have resource limits defined. If any container violates this constraint, Gatekeeper will prevent the resource from being created or modified.

The first violation rule checks whether a container in the input resource's specification (spec) has defined memory resource limits. If there are no memory resource limits defined, it generates a violation with a message indicating that the container lacks memory resource limits.

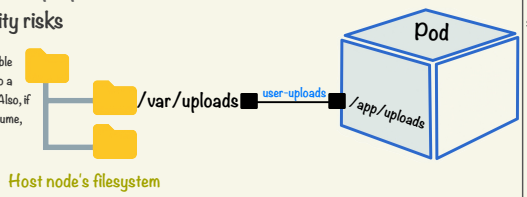
Storage

In Kubernetes, containers are typically considered to be ephemeral and immutable, meaning that they are designed to be short-lived and replaceable. This approach is well-suited for stateless applications that don't store or modify persistent data, but it can be challenging for stateful applications that require persistent storage.
To address this challenge, Kubernetes provides various ways to persist data, ranging from simple to complex solutions. Here are some of the approaches to persist data in k8s.

HostPath volumes:

HostPath volumes allow you to mount a directory from the host node's filesystem into a Pod. This approach is useful for testing and development purposes, but it is not recommended for production environments as it can create security risks

One important thing to note about HostPath volumes is that they are only accessible from the node where the pod is running. This means that if the pod is rescheduled to a different node, it will not have access to the files on the original node's filesystem. Also, if multiple pods are scheduled on the same node and they use the same HostPath volume, they will be able to read and write to the same files on the host node's filesystem




```yaml
apiVersion: v1
kind: Pod
metadata:
 name: web-app
 namespace: dev
spec:
 containers:
   - name: web-app
     image: my-web-app-image
     volumeMounts:
       - mountPath: /app/uploads
         name: user-uploads
 volumes:
   - name: user-uploads
     hostPath:
       path: var/uploads
       type: DirectoryOrCreate
```

2 We then mount the user-uploads volume to the container's /app/uploads directory using the volumeMounts field in the container specification. This allows the web application to access the user-uploaded files stored in the /var/uploads directory on the host node's filesystem

1 We are creating a HostPath volume named user-uploads that maps to the /var/uploads directory on the host node's filesystem

type field specifies that the directory should be created if it doesn't already exist

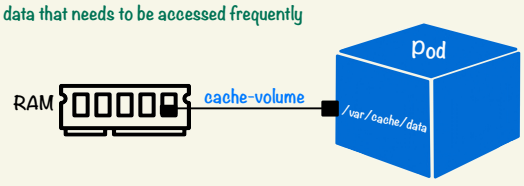
EmptyDir volumes:

EmptyDir volumes are a type of temporary storage volume that are created and attached to a Pod when the Pod is created. The data stored in an EmptyDir volume exists only for the lifetime of the Pod and is deleted when the Pod is deleted. These volumes are commonly used for storing temporary data that is needed by a Pod, such as cache files or temporary log

When you define an EmptyDir volume, you can specify a size limit for the volume. If you don't specify a limit, the application running in the pod can generate any amount of data, which can cause the disk to become full and potentially cause the node to become unavailable

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: monitoring-pod
spec:
  containers:
  - name: monitoring-container
    image: monitoring-image
    volumeMounts:
    - name: logs-volume
      mountPath: /var/log/monitoring-app
  volumes:
  - name: logs-volume
    emptyDir:
      sizeLimit: 1Gi
```

EmptyDir volume can be configured to store its data in memory instead of on disk. This provides faster access to the data in the volume, which can make it useful for caching data that needs to be accessed frequently



The medium field is used to indicate the underlying storage medium for a volume. By setting the medium to "Memory", the cache-volume volume will be created using the host node's RAM as the storage medium.

/var/cache/data directory inside the container is mounted to an EmptyDir volume named cache-volume. The cache-volume volume is configured with a sizeLimit of 1 gigabyte, which means that it can store up to 1 gigabyte of data in memory during the lifetime of the Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ML-app
spec:
  containers:
  - name: video-conv
    image: video-conv
    volumeMounts:
    - name: cache-volume
      mountPath: /var/cache/data
  volumes:
  - name: cache-volume
    emptyDir:
      medium: Memory
      sizeLimit: 1Gi
```

ConfigMaps and Secrets:

ConfigMaps and Secrets are Kubernetes objects that allow you to store configuration data and sensitive information such as credentials and keys, respectively. They can be mounted as volumes in a Pod, allowing the Pod to access the data as files Go to page 18

Persistent Volumes (PVs) and Persistent Volume Claims (PVCs):

PVs are independent storage volumes that can be provisioned from different storage providers such as cloud storage or on-premise storage systems, and PVCs are used to request storage resources from the PVs. The PVs and PVCs allow you to abstract the underlying storage infrastructure from your application, providing a layer of indirection. You can use PVs and PVCs to store data persistently, even if a Pod is deleted or restarted. PVs and PVCs can be used with different storage backends like NFS, iSCSI, Ceph, etc

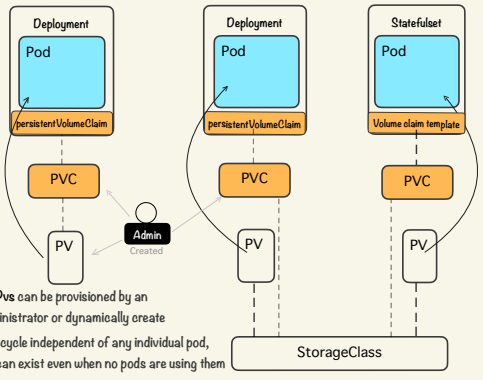
To connect PVs and PVCs to pods, you need to follow these steps:

- Provision a Persistent Volume (PV): As an administrator, you'll define and create a PV object, specifying the storage capacity, access modes, and other properties. This involves interacting with your underlying storage infrastructure, whether it's local disks, network storage, or cloud storage.
- Create a Persistent Volume Claim (PVC): A user or developer creates a PVC object, specifying their desired storage capacity, access modes, and any additional requirements. The PVC will be used by the pod to request storage.
- Binding PVC to PV: Once the PVC is created, Kubernetes matches it with an available PV that meets the requested criteria. The binding process ensures that the PVC and PV are associated with each other.
- Mounting the PV to a pod: In the pod's specification, you specify the PVC as a volume source. When the pod is scheduled and runs, Kubernetes mounts the PV associated with the PVC to a specified path within the pod's filesystem.

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-40g-rw
spec:
  capacity:
    storage: 40Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv1-40g-rw
  persistentVolumeReclaimPolicy: Retain
```

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-pvc-20g
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-image
      volumeMounts:
        - name: my-volume
          mountPath: /data
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: nfs-pvc-20g
```



Pvs can be provisioned by an administrator or dynamically create

PVs have a lifecycle independent of any individual pod, meaning they can exist even when no pods are using them

PVC is a request for storage by a pod. It is a way for pods to dynamically request a specific amount and type of storage without having to know the details of the underlying storage infrastructure

accessModes is a field that is used to specify how the volume can be mounted and accessed by a pod

- ReadOnlyMany (ROX): This access mode allows the volume to be mounted as read-only by multiple nodes in a cluster. This means that the volume can be mounted by multiple pods at the same time, but cannot be modified. This mode is typically used for shared read-only storage resources, such as configuration files or static data
- ReadWriteMany (RWX): This access mode allows the volume to be mounted as read-write by multiple nodes in a cluster. This means that the volume can be mounted by multiple pods at the same time, and can be modified. This mode is typically used for shared read-write storage resources, such as file shares or databases
- ReadWriteOnce (RWO): This access mode allows the volume to be mounted as read-write by a single node in a cluster. This means that the volume can be mounted by only one pod at a time, and is typically used for storage resources that can only be accessed by one node or pod at a time, such as local storage or block storage.

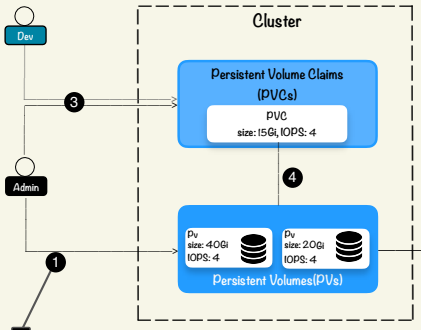
The **persistentVolumeReclaimPolicy** determines what happens to the contents of a Persistent Volume (PV) when it is released, specifying whether the contents should be retained or deleted

- Retain: The PV's contents are retained even after the PV is released. This means that the PV can be reused by creating a new PVC that requests the same storage capacity and access modes as the original PVC that used the PV
- Delete: The PV's contents are deleted when the PV is released. This means that the PV cannot be reused by creating a new PVC that requests the same storage capacity and access modes as the original PVC that used the PV
- Recycle (deprecated): The PV's contents are deleted when the PV is released, but the PV is made available for reuse. However, this value is deprecated and should not be used in newer versions of Kubernetes

PVs can be provisioned statically or dynamically

Static provisioning involves manually creating PVs and configuring their properties, such as storage capacity, access modes, and claimPolicy.(provisioned by an Administrator)

Dynamic provisioning allows Kubernetes to automatically create PV when a PVC is created. Dynamic provisioning can be implemented using StorageClasses



When you create a PV that uses an NFS volume, the PV connects to the NFS server and uses it as the backend storage for the PV

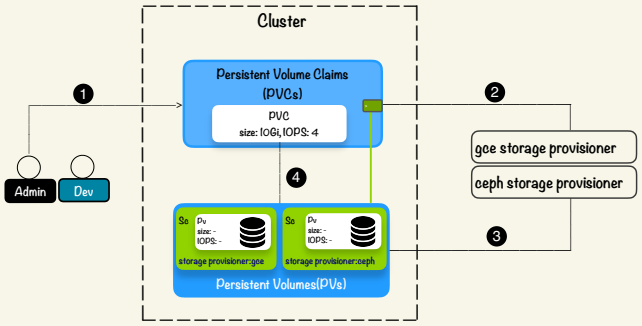
File storage

To create a Persistent Volume (PV) in Kubernetes from an existing NFS volume, you first need to create an NFS export on the NFS server that will make the volume available to the Kubernetes cluster

```
exportfs -o rw,sync,no_subtree_check,no_root_squash,size=40g,fsid=0,
sec=sys,anonuid=65534,anongid=65534 k8s-cluster-ip:/mnt/nfs_share/pv1-40g-rw
```

```
exportfs -o rw,sync,no_subtree_check,no_root_squash,size=20g,fsid=0,
sec=sys,anonuid=65534,anongid=65534 k8s-cluster-ip:/mnt/nfs_share/pv2-20g-rw
```

Persistent Volumes (PVs) that fit NFS volumes are created by the cluster administrator on the Kubernetes cluster.



storage classes act as an abstraction layer on top of PVs, allowing you to define a set of default parameters and policies that are used when dynamically provisioning new PVs based on PVC requests

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-40g-rw
spec:
  capacity:
    storage: 40Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv1-40g-rw
  persistentVolumeReclaimPolicy: Retain
```

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-20g-rw
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv2-20g-rw
  persistentVolumeReclaimPolicy: Retain
```

1 When a PVC is created, it specify a StorageClass to use, which will dictate how the PV is provisioned. If no StorageClass is specified, the default StorageClass will be used (if one is defined)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: gce-pd-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gce-pd-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

The provisioner field in a StorageClass specifies the name of the provisioner that should be used to provision the storage. There are many different provisioners available for different types of storage, including those for cloud providers like GCE, AWS, and Azure.

The storageClassName field in the PVC specification is used to specify the name of the StorageClass that should be used to provision the requested storage

This accessMode don't allow multiple pods to read and write to the same PVC simultaneously.

3 To use persistent storage in their Pod, the user can run the kubectl get pv command to view the available PV

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
nfs-pv1-20g-rw	20Gi	RWX	Retain	Available		nfs		1d
nfs-pv1-40g-rw	40Gi	RWX	Retain	Available		nfs		1d

In order to use one of these PVs for persistent storage in a Pod, we can create a Persistent Volume Claim (PVC) that requests storage from the desired PV

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-pvc-20g
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
```

4 Once the PVC is bound to the PV, we can mount the PV to the Pod by including it as a volume in the Pod definition file.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-image
      volumeMounts:
        - name: my-volume
          mountPath: /data
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: nfs-pvc-20g
```

Each PV can be bound to only one PVC at a time , because when a PVC is created, Kubernetes will try to find an available PV that matches the PVC's requirements based on capacity, access mode, and storage class. If a suitable PV is found, the PVC is bound to that PV, and the PV becomes unavailable for other PVCs to

When the Pod is created, Kubernetes will use the bound PVC named nfs-pvc-20g as a volume mount point to access the persistent storage associated with the bound PV. The volumeMounts section in the Pod specification specifies that the volume should be mounted at the path /data within the container, so any data written to that path will be stored persistently in the PV.

2 Kubernetes first tries to find an existing PV that matches the criteria specified in the PVC. If no suitable PV is found , Kubernetes requests the provisioner mapped to the PVC's storage class to create a new volume. The storage provisioner can be a plugin or a driver that interfaces with the underlying storage system

3 The storage provisioner creates a new PV that matches the PVC's requirements, such as size, access mode, and storage class

4 Once the new PV is created, Kubernetes binds it to the PVC and the PVC is ready to be used by a Kubernetes pod. The pod can then mount the volume and use it to store and retrieve data

If Pod is deleted and then rebuilt with the same PVC, it will be connected to the same persistent volume, and any data that was previously stored in the volume will still be accessible However if the PVC is deleted, any data stored in the associated persistent volume will be lost and the Pod that was using that PVC will no longer be able to access the data. This is because deleting a PVC deletes the binding between the PVC and the PV, which causes the PV to be released and potentially recycled for use by other PVC

Finalizers

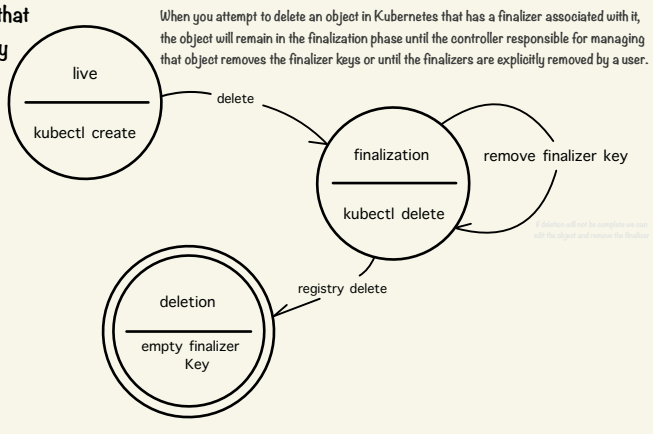
finalizers are markers attached to resources (such as pods, services, or deployments) to indicate that some additional cleanup or finalization steps need to be performed before the resource can be fully deleted. Finalizers are represented as strings and are stored in the metadata of the resource

Some common finalizers you've likely encountered are:

kubernetes.io/pv-protection

kubernetes.io/pvc-protection

The finalizers above are used on volumes to prevent accidental deletion

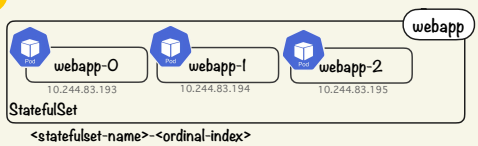


StatefulSet

StatefulSets are a type of workload object in Kubernetes that are used to manage stateful applications. They are designed to handle applications that require unique identities, stable network addresses, persistent storage ,ordered deployment and scaling, and graceful deletion. Such as databases, message queues, etc. StatefulSets maintain a sticky identity for each pod, so even if a pod gets rescheduled, it still maintains the same identity/name. The pods are created from the same spec, but are not interchangeable - each has a unique persistent ID.

Predictable pod name:

In a StatefulSet, each Pod is assigned a predictable name based on the name of the StatefulSet and its index. For example, if the StatefulSet is named "webapp" and has three replicas, the Pods will be named "webapp-0," "webapp-1," and "webapp-2." This allows for easy identification and reference to specific Pods within the Set



Each Pod has a stable hostname based on its ordinal index

Fixed individual DNS name:

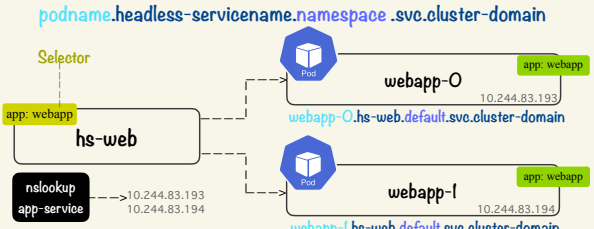
StatefulSets also provide a fixed individual DNS name for each Pod, based on the predictable name assigned to it. This allows applications to refer to each Pod by a consistent DNS name, even if the Pod is rescheduled to a different node. For example, if the StatefulSet is named "webapp," and the Pod is named "webapp-0," the DNS name for that Pod will be "webapp-0.webapp

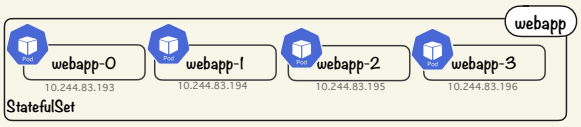
Headless service:

StatefulSets are accompanied by a headless service, which allows for direct communication with individual Pods rather than the Service as a whole. This is useful for stateful applications that require direct communication between Pods, such as database clusters.

Ordered Pod creation:

StatefulSets ensure that Pods are created in a specific order, with each Pod waiting for the previous one to be ready before starting. This is particularly important for stateful applications that require specific sequencing of events, such as database clusters





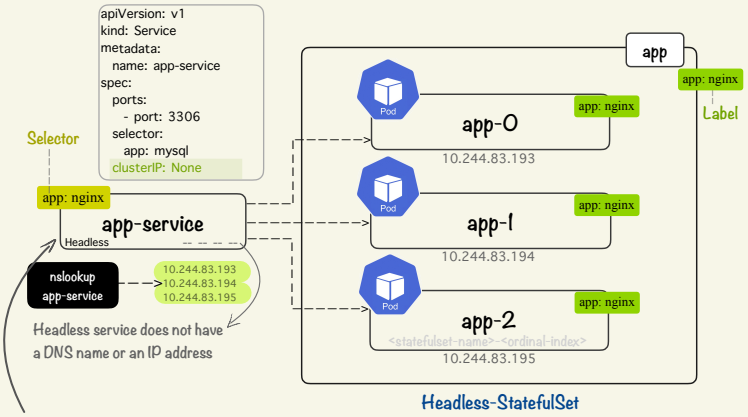
Pods are deployed in order from 0 to N-1, and terminated in reverse order from N-1 to 0.

Why do we need StatefulSets?

Consider an example of a stateful application - a database. Databases are typically stateful, meaning they require persistent storage to store their data. They also require stable network identities to ensure that client applications can consistently connect to the same instance of the database, If you deploy a database using a regular Deployment or RS, Kubernetes will create multiple replicas of the database, each with its own randomly assigned hostname and IP address. This can cause problems for the database, as the client applications may not be able to connect to the correct instance of the database, or data may be lost when pods are deleted or recreated. To solve these problems, you can use a StatefulSet to manage the deployment and scaling of the database.

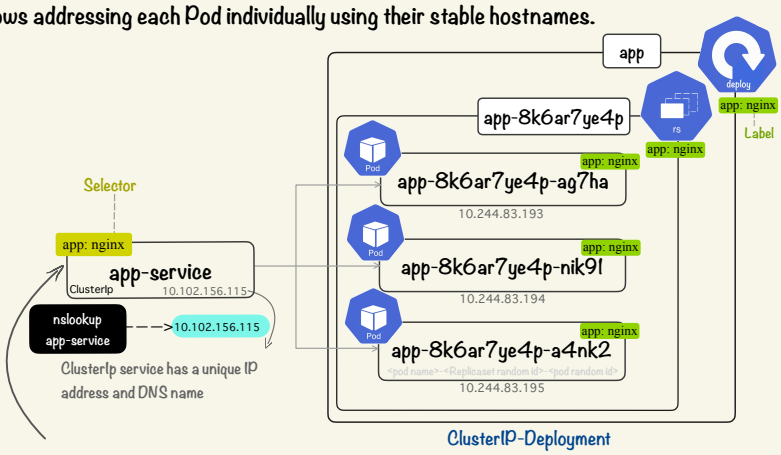
Headless service

A Headless Service is a type of Kubernetes service that does not have a ClusterIP assigned to it. Instead, it manages the Domain Name System (DNS) records directly. This means that when a client tries to connect to a Pod that is part of the Headless Service, it can use the DNS name associated with the Pod's IP address to directly communicate with the Pod. When used with StatefulSets, it allows addressing each Pod individually using their stable hostnames.



Headless-StatefulSet

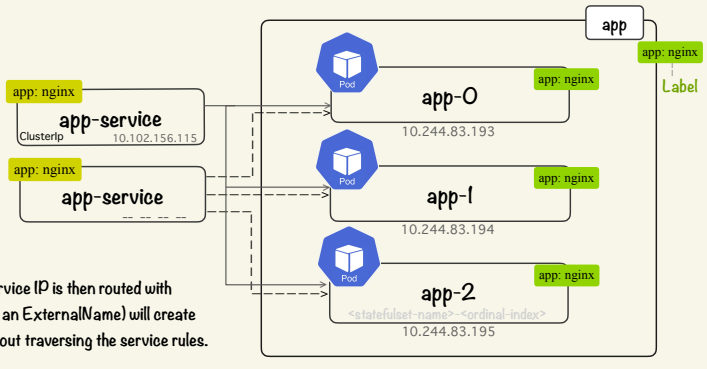
When a client sends traffic to a Headless Service, Kubernetes returns the IP addresses of all the Pods that are backing the service, regardless of their status. This means that the client may receive IP addresses for Pods that are not running or are in a failed state. The client is then responsible for load-balancing the traffic across the individual Pod IP addresses that are returned



ClusterIP-Deployment

When a client sends traffic to the service, Kubernetes chooses one of the Pods based on a load-balancing algorithm. Regular services use a ClusterIP address to load-balance traffic across the Pods that are backing the service

- Regular service provides a single IP address that represents a group of Pods, while a Headless Service provides individual DNS names and IP addresses for each Pod in the service
- Regular services are typically used for stateless applications that can handle traffic from multiple clients, while Headless Services are more commonly used for stateful applications that require direct access to individual Pods
- Headless services can be used in combination with regular services to provide both direct access to individual pods and load-balanced access to the service as a whole. For example, you might use a headless service to allow database nodes to communicate directly with each other, while also exposing a regular service for client applications to connect to
- Regular service has a virtual Service IP that exists as iptables or ipvs rules on each node. A new connection to this service IP is then routed with DNAT to one of the Pod endpoints, to support a form of load balancing across multiple pods.A headless service (that isn't an ExternalName) will create DNS A records for any endpoints with matching labels or name. Connections will go directly to a single pod/endpoint without traversing the service rules.



Headless-ClusterIP -StatefulSet

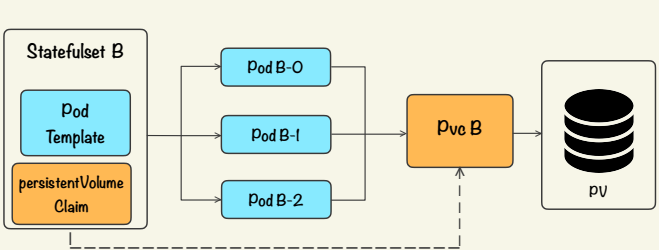
StatefulSets can use two types of storage

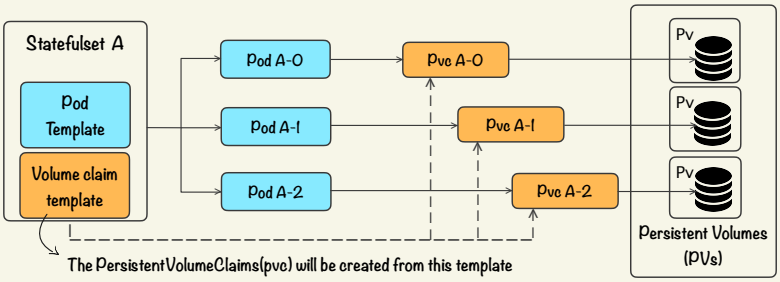
Shared Storage

All Pods in the StatefulSet share the same storage volume. Data is available to all Pods.
Good for things like caches, tmp files etc.
Specify a PersistentVolumeClaim template in the sfs spec. All Pods will get a clone of this PVC.
data can be corrupted if multiple Pods write to the same files

Dedicated Storage

Each Pod gets its own PersistentVolume. Data is isolated between Pods.
Good for databases, unique files etc.
Don't specify volumeClaimTemplates. StatefulSet will create a PVC for each Pod
Updating Pods is harder with dedicated storage, may need to coordinate Pod termination to avoid data loss






```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql-hs
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:latest
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: "yourpassword"
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pvc
```

The volumeMounts and volumes are defined in the pod template section of the StatefulSet manifest, which means that they will be shared by all pods created by the StatefulSet

serviceName field specifies the name of the Headless Service that controls the network identity of the StatefulSet's Pods and it is a mandatory filed

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-hs
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
  clusterIP: None
```

Pod template

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: manual
  resources:
    requests:
      storage: 1Gi
```

The PVC must be created beforehand either manually or through some automated process

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  nfs:
    path: /srv/nfs/kubedata/pv3
    server: 192.168.49.1
```

The PV and PVC are using the "manual" StorageClass. The PVC has requested a capacity of 1Gi and has been bound to the PV.

```
k get pv,pvc
NAME                                   CAPACITY   ACCESS   MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   AGE
persistentvolume/mysql-pv              1Gi        RWM              Retain           Bound    default/mysql-pvc     manual         5h27m

NAME                                        STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/mysql-pvc-sts3        Bound    mysql-pv   1Gi        RWM            manual         5h29m
```

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-hs
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:latest
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: root-password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-shared-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-shared-storage
    spec:
      accessModes: [ "ReadWriteMany" ]
      storageClassName: google-storage
      resources:
        requests:
          storage: 10Gi
```

volumeClaimTemplates is specified at the StatefulSet level, not in the pod template

The "volumeClaimTemplates" field in a StatefulSet is used to define persistent volume claims (PVCs) that will be used by the pods in the set for their storage needs. When a pod is created or rescheduled, it will automatically create/claim one of these PVCs and use it for its persistent storage

If you set the storageClassName to the name of a StorageClass that is configured with a dynamic provisioner, Kubernetes will automatically create a new PV based on the specifications defined in the volumeClaimTemplates section

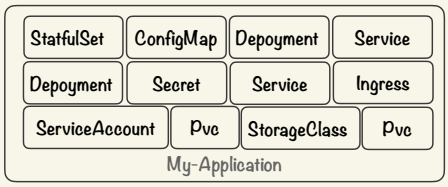
```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: google-storage
provisioner: kubernetes.io/gce-pd
....
```

How to deploy an application in k8s?

An application in Kubernetes typically consists of YAML files that define the k8s resources needed to run the application, such as Deployments, Services, ConfigMaps, and Secrets. You can deploy the application in Kubernetes manually by creating the YAML files and then using the `kubectl apply` command to create the Kubernetes resources on the cluster. Alternatively, you can use deployment tools like Kustomize, Helm, or the Helm Operator to automate the deployment process and simplify the creation of the YAML files. These tools provide a higher-level abstraction for managing Kubernetes resources and can make it easier to deploy and manage complex applications in Kubernetes.

Manual Deployment:

Manually deploying applications in Kubernetes involves creating YAML files that define the k8s resources needed to run the application, such as deployments, services, and config maps. You would then use the kubectl apply command to create those resources on the Kubernetes cluster.




```
kubectl apply -f deployment.yml
kubectl apply -f service.yml
kubectl apply -f statfulset.yml
...
kubectl apply -f serviceaccount.yml
```

This approach can be useful for simple applications or for users who prefer a more hands-on approach, but it can be time-consuming and error-prone for more complex applications

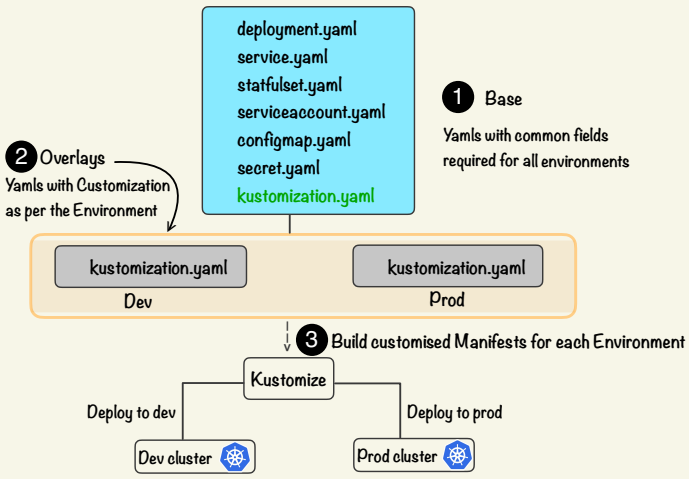
Kustomize

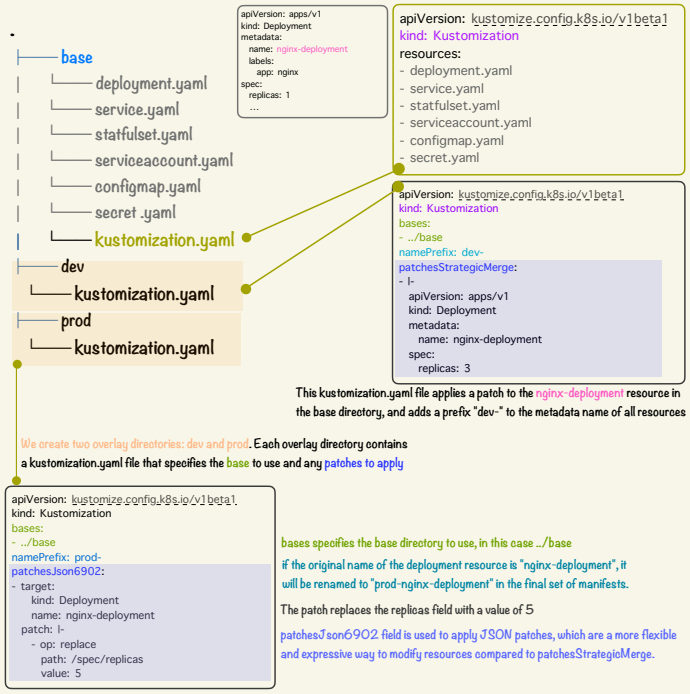
Kustomize is a tool for managing k8s manifest files using a declarative approach. It allows you to define a set of base manifests that define the desired state of your Kubernetes resources, and then apply changes using composition and customization. The basic workflow of Kustomize consists of the following steps:

Create a base directory containing your Kubernetes manifests. This directory represents the desired state of your application or environment

Define a kustomization.yaml file in the base directory. This file specifies the base resources to use, as well as any additional resources that should be added, modified, or removed

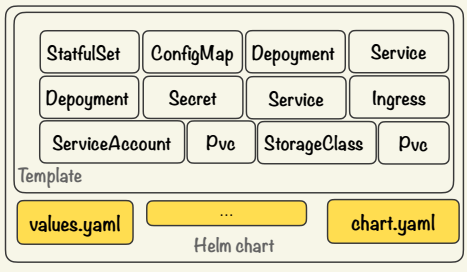
Create overlay directories for each environment or application variant, if needed. These overlay directories contain additional resources or modifications to apply on top of the base resources





Helm

Helm is a widely-used package manager for Kubernetes that simplifies the deployment and management of applications on a k8s cluster. It enables developers to package their applications as charts, which are reusable and shareable bundles that contain all the resources required to deploy an application on a Kubernetes cluster. With Helm, users can easily search for charts, install and upgrade applications, rollback changes, and manage dependencies through a straightforward command-line interface. Additionally, Helm supports versioning, which allows users to track changes to their applications over time and roll back to previous versions if necessary



Helm uses a packaging format called "charts". A chart is a collection of files that describe a related set of kubernetes resources.For example, instead of manually creating deployments, services, and other K8s objects, you can package these into a Helm chart. Then, anyone can easily deploy your application by installing the chart.

A Helm chart typically includes the following files:

Chart.yaml: This is the core file which includes the name, description, and version of the chart. This file is used by Helm to identify the chart and to provide information to the user when installing or upgrading the chart

```
apiVersion: v2
name: wordpress
description: A Helm chart for deploying WordPress on Kubernetes
version: 1.0.0
appVersion: 5.8.0
maintainers:
  - name: Your Name
    email: your@email.com
```

values.yaml : This file contains the default values for the chart's parameters. These parameters are used in the templates to generate the Kubernetes YAML files. The user can override these values during installation or upgrade using the --set flag or a values file. This file is used to allow users to customize the behavior of the chart without modifying the templates directly.

```
wordpress:
  image: wordpress:5.8.0-php7.4-apache
  imagePullPolicy: IfNotPresent
  replicaCount: 1
```

The values in this file are used by the templates in the templates/ directory to generate the k8s YAML files

To change a value, you can modify the values.yaml file and then run the helm upgrade command

The template syntax, enclosed in double curly braces ({{ }}), is used to reference the values specified in values.yaml

templates/ : This directory contains the Kubernetes YAML files that define the resources to be deployed. These files are usually written in a templating language like Go templating or Helm's own template language. The templates can include placeholders for the values defined in the values.yaml file. The templates can also include logic to conditionally include or exclude resources based on the values of the parameters

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-wordpress
  labels:
    app: wordpress
spec:
  replicas: {{ .Values.wordpress.replicaCount }}
...
    spec:
      containers:
        - name: wordpress
          image: {{ .Values.wordpress.image }}
          imagePullPolicy: {{ .Values.wordpress.imagePullPolicy }}
          ports:
            - name: http
              containerPort: 80
...
```

helpers.tpl : This file contains reusable snippets of code that can be used in the templates. These snippets can be used to simplify the templates and make them more readable. For example, a helper function might generate a random password or generate a unique name for a resource.

Do you want to deploy an application using Helm in Kubernetes? Here are the general steps to follow

Install Helm: You need to install Helm on your local machine or on the cluster where you will be deploying the application. You can follow the official Helm installation guide for your operating system to install Helm.

Add the Helm chart repository: Add the Helm chart repository that contains the application you want to deploy using the helm repo add command. You can specify a name for the repository and the URL of the repository.

Search for the Helm chart: Use the helm search command to search for the Helm chart that contains the application you want to deploy. You can specify the repository name or search all repositories.

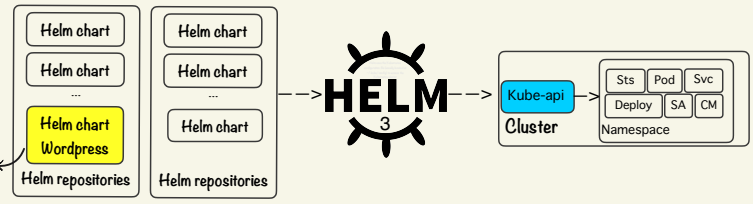
Configure the Helm chart: Create a values.yaml file to configure the Helm chart. This file contains the values that will be used to replace the placeholders in the Kubernetes resource files.

Install the Helm chart: Use the helm install command to install the Helm chart to the Kubernetes cluster. You can specify the release name, namespace, and any other required parameters using the command line or a YAML file

Verify the deployment: After the Helm chart has been installed, you can use kubectl commands to verify that the Kubernetes resources have been created and are running correctly

Upgrade or rollback the Helm chart: If you need to make changes to the application, you can use the helm upgrade command to upgrade the Helm chart. If there are issues with the new version, you can use the helm rollback command to revert to a previous version

How to deploy an application such as WordPress from a Helm repository using Helm?



1 **Add the WordPress Helm chart repository:**
you need to add the WordPress Helm chart repository to your local Helm installation. You can do this by running the following command:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

The Bitnami Helm repository contains a variety of charts for popular applications like WordPress, MySQL.

Update the Repository
This step ensures that Helm has the latest versions of all the charts from the Bitnami repository.

```
helm repo update
```

better to create a Kubernetes namespace for your WordPress installation in order to isolate the resources associated with your WordPress deployment from other resources running in the same Kubernetes cluster

```
kubectl create namespace wordpress
```

2 **Customize the WordPress deployment:**
Before deploying WordPress, let's customize some values. The default configuration can be obtained using the following command:

```
helm show values bitnami/wordpress
```

You can customize the values in a Helm chart by using the --set flag when you install the chart or by creating a values.yaml file that overrides the default values

- If you want to customize the installation, you can pass additional parameters to the Helm chart using the --set flag. For example, you can set a custom password for the WordPress administrator account by running the following command

```
helm install my-wordpress bitnami/wordpress --namespace wordpress --set wordpressEmail=admin@example.com
```

- WordPress Helm chart comes with a default values file (values.yaml) which contains all the configuration options. We'll create a custom values file (values.yaml) to override some of these defaults, and customize the settings according to ours needs

3 **Install the chart:**
Once you have customized the values, you can install the chart on your Kubernetes cluster using the helm install command. to install the WordPress chart with a release name of my-wordpress, run the following command:

```
helm install my-wordpress bitnami/wordpress --namespace wordpress --create-namespace -f values.yaml --set service.type=NodePort
```

The my-wordpress argument is the name of the release that Helm will use to track the installation, and the --namespace wordpress argument specifies the namespace in which to install WordPress

This command installs WordPress using the values.yaml file and sets the service.type value to NodePort

values.yaml

```
wordpressUsername: myusername
wordpressPassword: mypassword
wordpress:
  persistence:
    size: 20Gi
#mariadb.auth.rootPassword= ROOT_PASSWORD
mariadb:
  auth:
    rootPassword: ROOT_PASSWORD
```

Verify the chart:

```
helm list -n wordpress
NAME          NAMESPACE   REVISION   UPDATED                                   STATUS     CHART               APP VERSION
my-wordpress  wordpress   1          2023-07-28 20:40:28.214200542 +03 +03     deployed   wordpress-16.1.33   6.2.2
```

releaseName — A release is an instance of an application deployed by Helm from a chart

Upgrade or rollback the chart:

```
helm upgrade -f values2.yaml my-wordpress bitnami/wordpress -n wordpress
```

```
helm upgrade [RELEASE] [CHART] [flags]
```

```
helm list -n wordpress
NAME          NAMESPACE   REVISION   UPDATED                                   STATUS     CHART               APP VERSION
my-wordpress  wordpress   2          2023-09-26 11:42:00.841703412 +03 +03     deployed   wordpress-17.1.6    6.3.1
```

A revision is a versioned change to the release. Each time a release is installed or upgraded, a new revision is created incrementally (rev 1, 2, 3 etc).

How to get custom values for a helm release?

```
helm get values my-wordpress --revision=2 -n wordpress
USER-SUPPLIED VALUES:
wordpressPassword: "qazwsx"
wordpressUsername: daniele
```

```
helm rollback my-wordpress 1 -n wordpress
```

```
helm rollback RELEASE_NAME REVISION_NUMBER
```

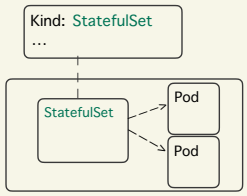
>> operator

operator is a method of packaging, deploying, and managing a specific application or workload on a Kubernetes cluster. Operators are essentially Kubernetes controllers that are designed to automate the deployment and management of complex applications or services

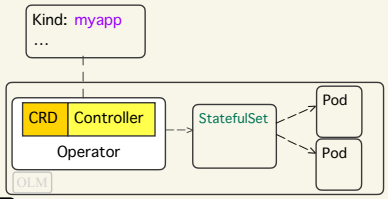
An operator typically consists of custom resources, custom controllers, and a set of Kubernetes objects that are defined to manage the application or workload. The custom resource is a Kubernetes object that represents the desired state of the application or workload, while the custom controller is responsible for ensuring that the actual state of the application or workload matches the desired state.

Managing stateful applications in Kubernetes can be challenging, but operators are particularly well-suited for this task. For example, an operator for a database application might automate tasks such as provisioning new database instances, scaling the database up or down, performing backups and restores, and handling failovers

Operators are typically implemented using the Kubernetes Operator SDK, which provides a set of tools and libraries for building, testing, and deploying operators



Dev — How to scale up a StatefulSet?
How to do leader election in Kubernetes?
How to migrate databases?
..



Dev — operators can definitely help to automate many routine tasks associated with managing complex applications in Kubernetes, & this can free up human operators to focus on more strategic tasks

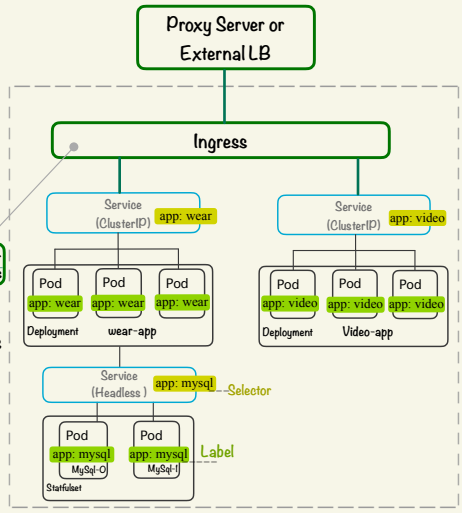
Ingress

Ingress is an API object in Kubernetes that allows access to your Kubernetes services from outside the Kubernetes cluster. It provides load balancing, SSL termination and name-based virtual hosting for your services, In other words, it's a way for your applications to expose URLs to the outside world.

Ingress provides external reachable URLs, SSL termination and name-based virtual hosting to services in the cluster. This means you can route requests to different services based on the request host or path.

Ingress provides layer 7 load balancing. It acts as a reverse proxy and load balances traffic to different services in your Kubernetes cluster

Ingress object allows you to expose multiple services through a single IP address



If you use the LoadBalancer service type, the service is made available to clients outside the cluster through a load balancer. This approach is fine if you only need to expose a single service externally, but it becomes problematic with large numbers of services, since each service needs its own public IP address.Fortunately, by exposing these services through an Ingress object instead, you only need a single IP address.

Ingress consists of two main components:

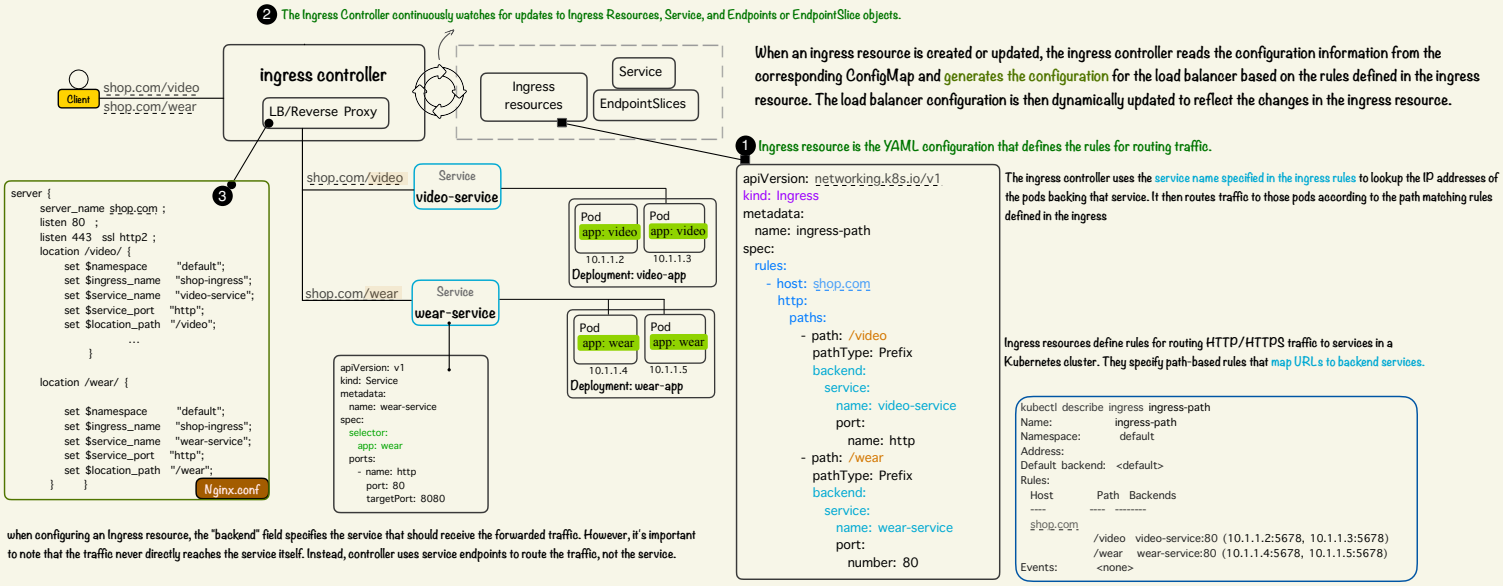
- **Ingress resource** is a Kubernetes API object that defines the rules for how external traffic should be directed to services within a cluster. The ingress resource specifies the rules for routing traffic based on the host name, path, and other criteria. It also specifies the backend services that should receive the traffic.
- **Ingress controller** is responsible for implementing the rules defined in the ingress resource and handling external traffic based on these rules. Ingress controllers like Nginx use ConfigMaps to store the configuration for the ingress resources and dynamically generate Nginx configuration based on the rules defined in the ingress resource

Kubernetes only provides the Ingress resource and needs a separate Ingress Controller to satisfy the Ingress. There are several options available, but for the purpose of this guide, we'll use the Nginx Ingress Controller.Install the Nginx Ingress Controller

In order for the Ingress resource to work, the cluster must have an ingress controller running.Unlike other types of controllers which run as part of the kube-controller-manager binary, Ingress controllers are not started automatically with a cluster. You have to select an Ingress Controller compatible with your setup and start it manually. (The actual implementation of Ingress is done by Ingress Controllers)

How Does an Ingress Controller Work?
Here's a simplified view of how an Ingress Controller works:

1. You define an Ingress Resource in your cluster, which has a set of routing rules associated with it.
2. The Ingress Controller continuously watches for updates to Ingress Resources , Service, and Endpoints or EndpointSlice objects. When it detects a new or modified these objects, the controller is notified. it reads the information in these objects to understand what traffic routing changes it needs to make.
3. The Ingress Controller configures the load balancer to implement the desired traffic routing.



How to customize Nginx Ingress Controller?

Helm Chart Values: If deploying the Ingress controller via Helm chart, you can customize settings by overriding chart values. The Helm chart exposes many config settings as values.
ConfigMap: using a ConfigMap to set global configuration in NGINX, For example, you can specify custom log formats, change timeout values, enable features like GeoIP, etc
Annotation: use this if you want a specific configuration for a particular ingress rule.

How to enable Basic Authentication for an ingress rule in Kubernetes?

This example shows how to add authentication in a Ingress rule using a secret that contains a file generated with htpasswd

1 Generate the base64 encoded user/pass combo:

```
htpasswd -nbm arye Heisenberg | base64
```

2 Convert htpasswd into a secret:

```
kubectl create secret generic shop-basic-auth --from-literal=auth=<base64 output>
```

Or

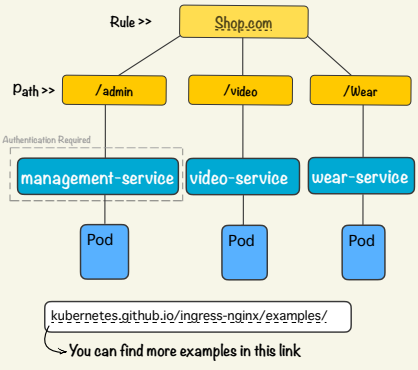
```
apiVersion: v1
kind: Secret
metadata:
  name: shop-basic-auth
  namespace: default
data:
  auth: YXJ5ZTokYXByMSQxbzAzWElTQiRJVFYudWh0dmcuVmV5d0t5a0s1cC4vCgo=
```

Or

```
kubectl create secret generic shop-basic-auth --from-literal=username=arye --from-literal=password=Heisenberg
```

3 Configure the Ingress rule to use the basic authentication secret

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: shop.com-admin
  namespace: default
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/auth-type: basic
    nginx.ingress.kubernetes.io/auth-secret: shop-basic-auth
    nginx.ingress.kubernetes.io/auth-realm: Authentication Required
spec:
  rules:
  - host: shop.com
    http:
      paths:
        - path: /admin
          pathType: Prefix
          backend:
            service:
              name: management-service
              port:
                name: http
```



How to enable TLS for an ingress rule in Kubernetes?

Method 1: Self-signed certificate

Generate a self-signed certificate and private key:

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=arye.ir"
```

Create a secret containing the key and certificate:

```
kubectl create secret tls tls-secret --key tls.key --cert tls.crt
```

Method 2: Use Certbot

Use Certbot to generate a TLS certificate for your domain.

```
certbot --manual --preferred-challenges dns certonly -d arye.ir
```

Create a Kubernetes secret that contains the private key and the certificate

```
kubectl create secret tls tls-secret --key privkey.pem --cert cert.pem
```

Configure Ingress to Use the Certificate

Reference tls-secret secret in your Ingress resource

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-path
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - arye.ir
    secretName: tls-secret
  rules:
    - host: arye.ir
      http:
        paths:
          - path: /booklet
            pathType: Prefix
            backend:
              service:
                name: book-service
                port:
                  name: http
...
```

This YAML manifest describes an Ingress resource that enables TLS for the host "arye.ir", redirects HTTP traffic to HTTPS, and defines a routing rule for the path "/booklet" to the backend service named "book-service" on the specified port

You can find more annotations in this link

https://github.com/kubernetes/ingress-nginx/blob/main/docs/user-guide/nginx-configuration/annotations.md

Method 3: Use Cert-manager

Cert-manager

Cert-manager is a certificate management controller for k8s. It helps with issuing and renewing certificates from various sources, such as Let's Encrypt, HashiCorp Vault, Venafi. cert-manager ensures certificates are valid and up to date, and will attempt to renew certificates at a configured time before expiry.

main features of cert-manager

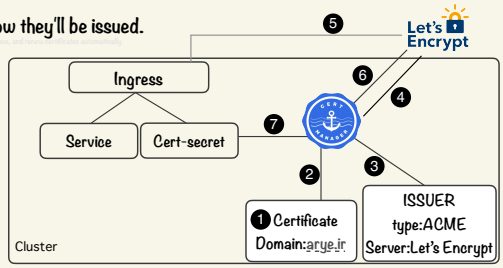
- Issuing and renewing certificates from a variety of sources
- DNS01 and HTTP01 ACME challenge solver support for Let's Encrypt certificates
- Issuing certificates for Certificate resources using CRDs
- Automated creation and updating of k8s Secrets with certificates
- Issuing certificates for Ingress resources with annotations

Cert-manager mainly uses two different custom Kubernetes resources (CRDs) to configure and control how it operates, as well as to store state. These resources are Issuers and Certificates.

Issuer is an object that represents a particular certificate authority or a specific method for issuing certificates. It defines the parameters and configurations required to request certificates. An Issuer can be used to issue certificates within a single namespace or cluster in Kubernetes. There are different types of issuers supported by CertManager, such as:

- ACME Issuer: This type of issuer integrates with the Automated Certificate Management Environment (ACME) protocol, which is commonly used by Let's Encrypt and other CAs. ACME issuers automate the process of obtaining and renewing certificates.
- CA Issuer: This type of issuer is used when you have an existing certificate authority (CA) that you want to use for issuing certificates. It requires you to provide the CA's certificate and private key.
- Self-Signed Issuer: This type of issuer is used when you want to generate self-signed certificates within the Kubernetes cluster. It is typically used for testing or development purposes

Certificates resources allow you to specify the details of the certificate you want to request. They reference an issuer to define how they'll be issued.



what happens when you create a Certificate resource in cert-manager:

1. You create a Certificate resource with details like the domain name, secret name to store certificate, and reference to the Issuer
2. The cert-manager controller sees the new Certificate and kicks off the issuance process
3. cert-manager first checks if the referenced Issuer exists and is valid. The Issuer has the details for the certificate authority
4. cert-manager requests the certificate authority (CA) like Let's Encrypt to issue a certificate for the requested domain
5. The CA validates that you own/control the domain name by performing a challenge. For example, with HTTP challenge, you need to have a temporary file served on the domain
6. Once domain ownership is validated, the CA issues the signed certificate. The certificate is returned to cert-manager
7. cert-manager takes the certificate and creates or updates the Kubernetes secret defined in the Certificate. This secret will contain the certificate and private key.

How can I issue a certificate for the domain arye.ir using cert-manager from Let's Encrypt?

Configure Let's Encrypt Issuer

Cert-manager uses 'Issuer' or 'ClusterIssuer' resources to represent certificate authorities. We'll create a 'ClusterIssuer' for Let's Encrypt

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: your-email@your-domain.com
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

privateKeySecretRef specifies the name of the Kubernetes secret that will be used to store the private key for the certificate.

solvers specifies the method for verifying ownership of the domain. In this case, we are using the HTTP-01 challenge, which involves creating a temporary file in the web root of the domain and responding to an HTTP request to that file. The ingress field specifies that we will use an Ingress resource to serve the challenge.

this ClusterIssuer can be referenced by other resources like Certificate or ingress to automatically generate and manage self-signed certificates.

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
```

Issue a Certificate

Create a certificate resource to obtain the certificate from Let's Encrypt for the specified domain.

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: arye-ir-cert
  namespace: default
spec:
  secretName: arye-ir-tls
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  commonName: arye.ir
  dnsNames:
  - arye.ir
  - *.arye.ir
  duration: 90d
```

After a few moments, cert-manager should issue a certificate for your domain and store it in the secret specified in the certificate resource. You can verify that the certificate has been issued by checking the contents of the secret:

```
kubectl describe secret arye-ir-tls
```

```
Data
====
tls.crt:  2316 bytes
tls.key:  1704 bytes
```

secretName specifies the name of the Kubernetes secret that will be used to store the TLS certificate and private key.

issuerRef specifies the name and kind of the Kubernetes resource that is used as the issuer for this certificate. In this case, we are using the previously defined letsencrypt-prod ClusterIssuer

commonName specifies the common name for the TLS certificate. In this case, it is set to arye.ir.

dnsNames specifies the list of DNS names for which the TLS certificate should be issued. In this case, we are issuing the certificate for arye.ir and all subdomains of the specified domain

You can configure TLS for Ingress using annotations instead of Certificate resources

- Use annotations for basic, single TLS certificate configuration. Simpler, but less flexible.
- Use Certificate resources for multiple certificates, automation, and advanced management. More complex, but more flexible and powerful

Configure Ingress to Use the Certificate

Your Ingress configuration should use the secret arye-ir-tls for its TLS configuration

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: arye.ir-ingress
spec:
  tls:
  - hosts:
    - arye.ir
    - *.arye.ir
    secretName: arye-ir-tls
  rules:
  - host: arye.ir
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: your-service-name
            port:
              number: 80
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: arye.ir-ingress
annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - arye.ir
    - *.arye.ir
    secretName: arye-ir-tls
  rules:
  - host: arye.ir
...
```

cert-manager.io/cluster-issuer: References the Issuer resource in cert-manager that will be used to obtain the certificate

Validate the Setup

Check if the certificate has been issued successfully

```
kubectl describe certificate arye-ir-cert
```

```
Name:         arye-ir-cert
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  cert-manager.io/v1
Kind:         Certificate
...
Spec:
  Common Name:  arye.ir
  DNS Names:
    arye.ir
  Issuer Ref:
    Group:      cert-manager.io
    Kind:       ClusterIssuer
    Name:       letsencrypt-prod
  Secret Name:  tls-secret
Status:
  Conditions:
    Last Transition Time:  2023-09-27T10:01:00Z
    Message:               Certificate is up to date and has not expired
...
  Not After:               2024-09-27T10:00:00Z
Events:                    <none>
```

Add-ons

Kubernetes has a rich ecosystem of add-ons and extensions that provide additional functionality and features to enhance and extend the capabilities of a Kubernetes cluster. So far, we have covered a few of these add-ons in the booklet. Now, let's introduce some additional add-ons that can further enhance your Kubernetes experience:

Argo CD is a powerful open-source tool designed for Kubernetes, enabling GitOps continuous delivery. It simplifies application deployment and management by utilizing a declarative approach. With Argo CD, you can define the desired state of your applications using Kubernetes manifests stored in a Git repository. It provides a user-friendly graphical interface to monitor application status, track changes, and roll back if needed. By following GitOps principles, Argo CD ensures that your cluster's configuration matches the desired state defined in the repository, automatically deploying and updating applications.

Service mesh add-ons, like Istio and Linkerd, are powerful tools that enhance the networking and observability capabilities of microservices within a Kubernetes cluster. By integrating transparently with the cluster, they offer advanced features for traffic management, security, and distributed tracing. These service mesh solutions enable fine-grained control over traffic routing, load balancing, and fault tolerance mechanisms, ensuring efficient and reliable communication between microservices. With built-in security features like mutual TLS authentication and encryption, they provide robust protection for service-to-service communication. Additionally, service mesh add-ons enable comprehensive observability with distributed tracing, metrics collection, and logging, allowing for deep insights into the behavior and performance of microservices.

Rook and Longhorn are two notable storage-related add-ons for Kubernetes. Rook is a cloud-native storage orchestrator that enables the deployment and management of various storage solutions as native Kubernetes resources. It automates the provisioning, scaling, and lifecycle management of distributed storage systems like Ceph, CockroachDB, and more. On the other hand, Longhorn is a lightweight, open-source distributed block storage system built for Kubernetes. It provides reliable, replicated block storage for stateful applications, offering features like snapshots, backups, and volume expansion. Together, Rook and Longhorn empower Kubernetes users to easily deploy and manage resilient, scalable, and persistent storage solutions within their clusters, enhancing the availability and data management capabilities of their applications.

Monitoring and logging add-ons, such as Prometheus, facilitate the collection and storage of time-series data and metrics from diverse Kubernetes components and applications, enabling comprehensive analysis and alerting capabilities. Additionally, Fluentd serves as a dependable log aggregation tool, simplifying the gathering, parsing, and forwarding of logs from multiple sources to ensure centralized and scalable log management. The ELK (Elasticsearch, Logstash, and Kibana) stack offers a comprehensive solution for monitoring and logging, utilizing Elasticsearch for efficient log indexing and searching, Logstash for data processing and filtering, and Kibana for visualizing and analyzing log data. Together, these add-ons provide Kubernetes users with powerful tools for monitoring performance, detecting issues, and gaining valuable insights to optimize their Kubernetes environments.

Additionally, The CNCF landscape is an excellent resource for discovering and exploring a vast array of add-ons and tools within the cloud-native ecosystem. It offers a visual representation of different projects and categories, allowing users to navigate through various technologies that can enhance their Kubernetes deployments. Whether you're looking for monitoring and observability tools, networking solutions, or storage options, the CNCF landscape provides a comprehensive overview of the available options. By exploring the CNCF landscape, you can expand your knowledge and make informed decisions about incorporating the right add-ons and tools into your Kubernetes and cloud-native environments. It's a valuable resource for staying up-to-date with the latest innovations and finding the best solutions to meet your specific needs.