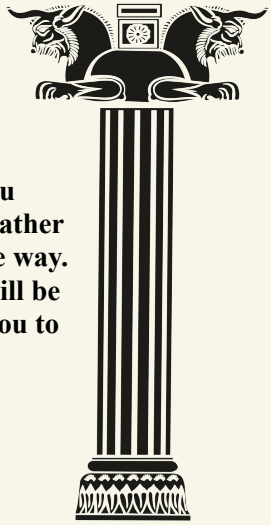
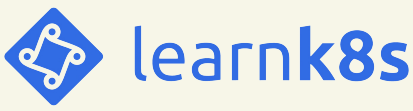
Kubernetes pocket guide

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

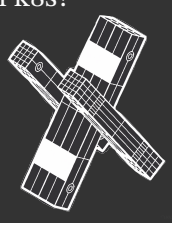
Scheduling

Lifecycle Management

Security

Storage

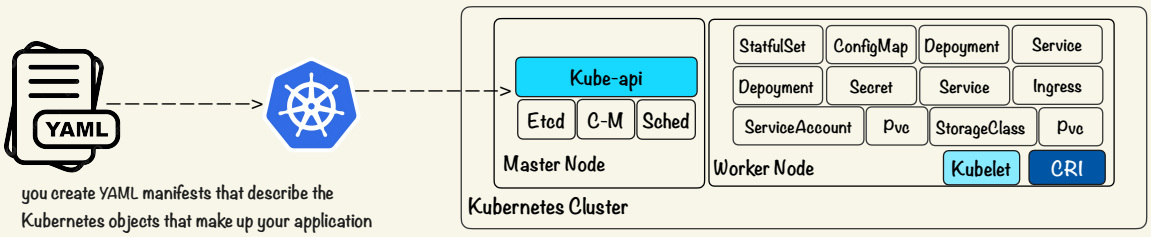
Addons



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

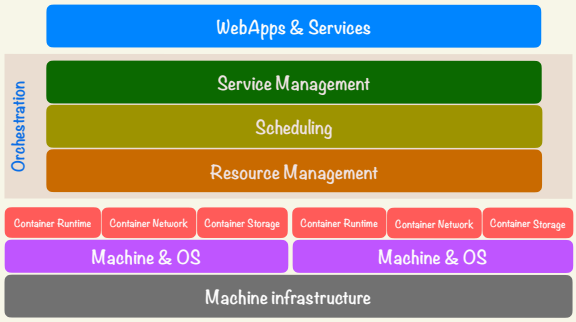
Increase automation: Kubernetes automates many of the tasks involved in deploying and managing containerized applications, such as rolling updates, scaling, and load balancing. This can help reduce the burden on operations teams and improve efficiency

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



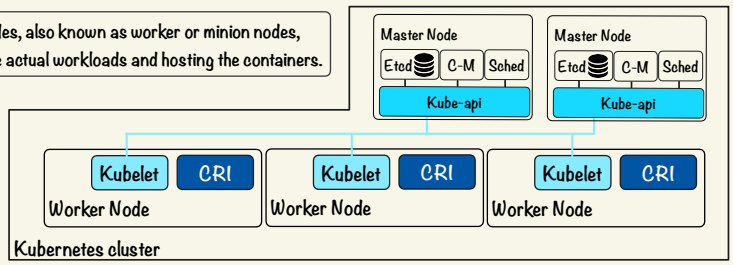
The CSI is a standard for exposing storage systems to container orchestrators like Kubernetes. Kubernetes uses a CSI driver to interact with the underlying storage system. The CSI driver is responsible for managing the lifecycle of the storage volumes used by your applications, including creating, deleting, 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:

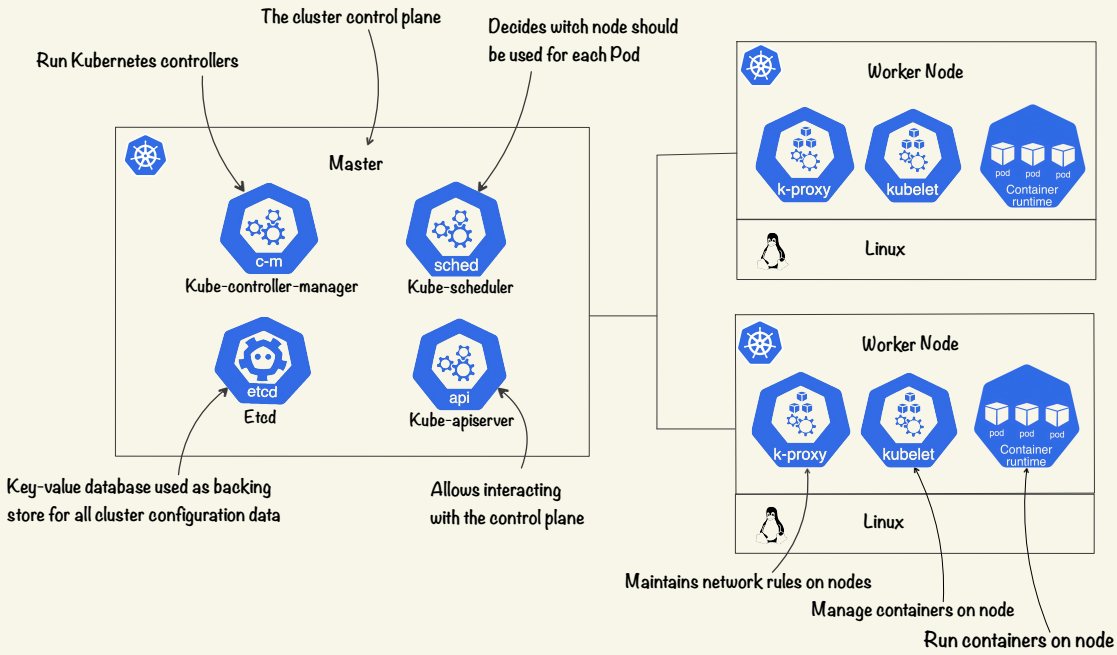
Master Node(s): The master node is responsible for managing the overall state and control of the cluster.

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

```
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
```

An example kubeconfig file

```
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
```

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

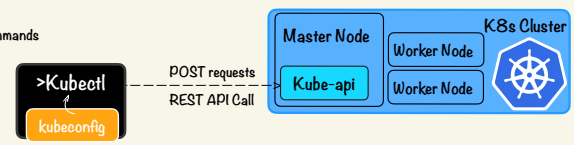
provides information about a user that can authenticate to a k8s cluster. Each user configuration includes the user name and any necessary authentication information such as a client certificate and key

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

`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

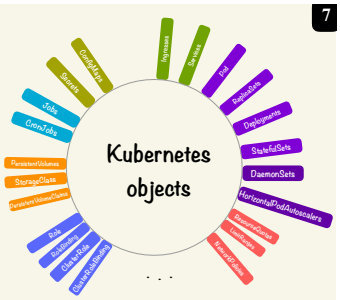
```
...
plugins=(
  git
  kubectl
)
```

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,...

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>
```

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
```

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
```

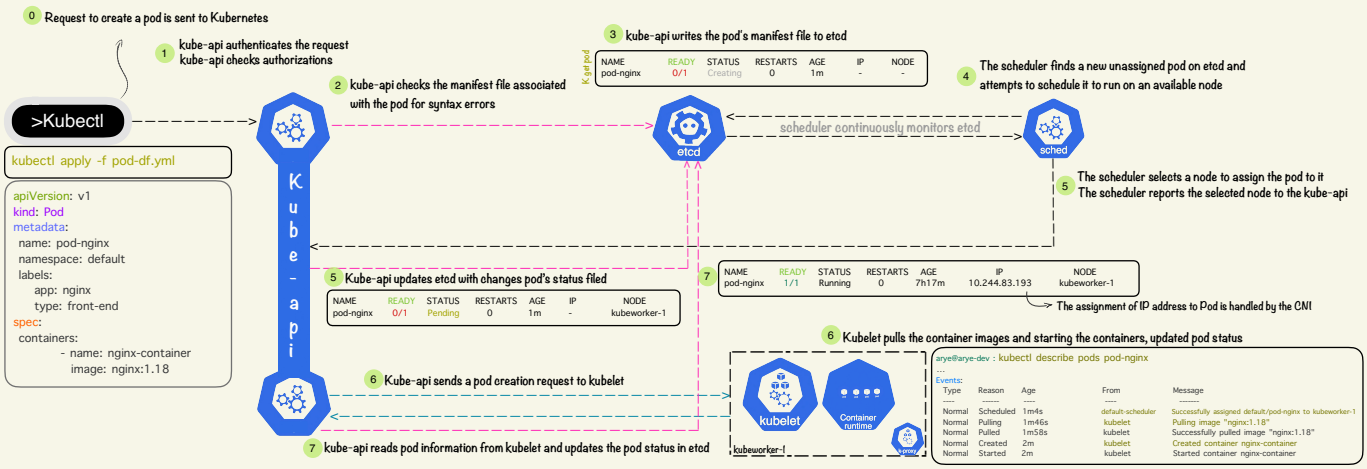
```
kubectl delete -f pod-df.yml
```

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

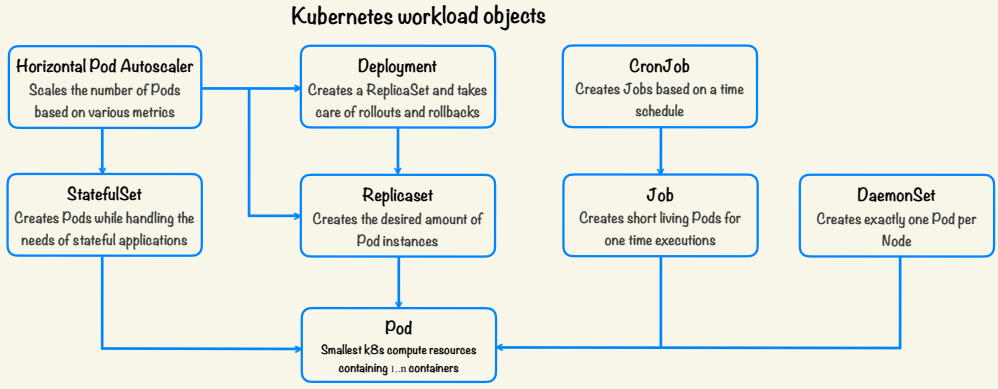
Process of creating a pod in Kubernetes:

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.



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

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

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

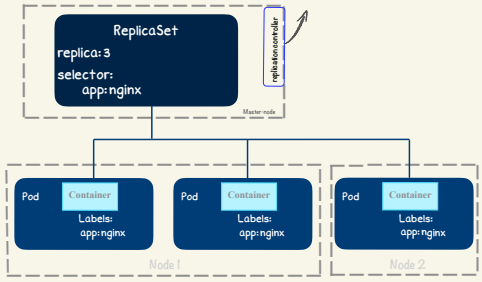
The pods that are subsets of this ReplicaSet must have a label with app: nginx

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



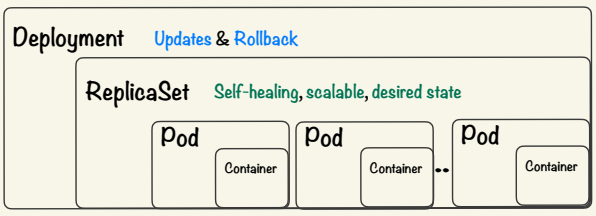

```
K get pod
NAME                      READY   STATUS    RESTARTS   AGE     IP              NODE
pod-nginx                 1/1     Running   0          7h17m   10.244.83.193   kubeworker-1
replicaset-nginx-bnrcs    1/1     Running   0          23s     10.244.83.195   kubeworker-1
replicaset-nginx-hm26d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

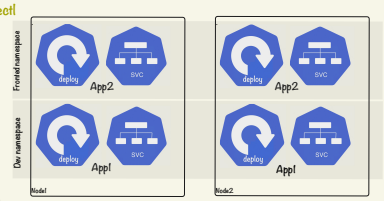
```
kubectl create namespace dev
```

1

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

2

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
```

=>

```
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
```

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

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

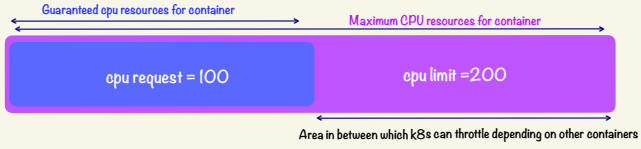
---

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.