K8s

Microservices-based architecture is aligned with Event-driven Architecture and Service-Oriented Architecture (SOA) principles,

where complex applications are composed of small independent processes which communicate with each other through Application Programming Interfaces (APIs) over a network.

APIs allow access by other internal services of the same application or external, third-party services and applications.

Seamless upgrades and patching processes are other benefits of microservices architecture.

There is virtually no downtime and no service disruption to clients because upgrades are rolled out seamlessly - one service at a time, rather than having to recompile, rebuild and restart an entire monolithic application.

As a result, businesses are able to develop and roll out new features and updates a lot faster, in an agile approach, having separate teams focusing on separate features, thus being more productive and cost-effective.

Containers

Container images allow us to confine the application code, its runtime, and all of its dependencies in a pre-defined format. The container runtimes like runC, containerd, or cri-o can use pre-packaged images as a source to create and run one or more containers.

or example, can be deployed on a workstation, with or without an isolation layer such as a local hypervisor or container runtime, inside a company's data center, in the cloud on AWS Elastic Compute Cloud (EC2) instances, Google Compute Engine (GCE) VMs, DigitalOcean Droplets, IBM Virtual Servers, OpenStack, etc.

Kubernetes

"Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications".

Kubernetes offers a very rich set of features for container orchestration. Some of its fully supported features are:

- ***Automatic bin packing***
  Kubernetes automatically schedules containers based on resource needs and constraints, to maximize utilization without sacrificing availability.

- ***Designed for extensibility***
  A Kubernetes cluster can be extended with new custom features without modifying the upstream source code.
- ***Self-healing***
  Kubernetes automatically replaces and reschedules containers from failed nodes. It terminates and then restarts containers that become unresponsive to health checks, based on existing rules/policy. It also prevents traffic from being routed to unresponsive containers.
- ***Horizontal scaling***
  Kubernetes scales applications manually or automatically based on CPU or custom metrics utilization.
- ***Service discovery and load balancing***
  Containers receive IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set.

Additional fully supported Kubernetes features are:

- ***Automated rollouts and rollbacks***
  Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, constantly monitoring the application's health to prevent any downtime.
- ***Secret and configuration management***
  Kubernetes manages sensitive data and configuration details for an application separately from the container image, in order to avoid a rebuild of the respective image. Secrets consist of sensitive/confidential information passed to the application without revealing the sensitive content to the stack configuration, like on GitHub.
- ***Storage orchestration***
  Kubernetes automatically mounts software-defined storage (SDS) solutions to containers from local storage, external cloud providers, distributed storage, or network storage systems.
- ***Batch execution***
  Kubernetes supports batch execution, long-running jobs, and replaces failed containers.
- ***IPv4/IPv6 dual-stack***
  Kubernetes supports both IPv4 and IPv6 addresses.

In order to communicate with the Kubernetes cluster, users send requests to the control plane via a Command Line Interface (CLI) tool, a Web User-Interface (Web UI) Dashboard, or an Application Programming Interface (API).

o persist the Kubernetes cluster's state, all cluster configuration data is saved to a distributed key-value store which only holds cluster state related data, no client workload generated data. The key-value store may be configured on the control plane node (stacked topology), or on its dedicated host (external topology) to help reduce the chances of data store loss by decoupling it from the other control plane agents.

A control plane node runs the following essential control plane components and agents: API server, scheduler, controller managers, and key-value data store.

n addition, the control plane node runs: container runtime, node agent (kubelet), proxy (kube-proxy), optional add-ons for observability, such as dashboard, cluster-level monitoring, and logging.

All the administrative tasks are coordinated by the kube-apiserver, a central control plane component running on the control plane node. The API Server intercepts RESTful calls from users, administrators, developers, operators and external agents, then validates and processes them.

API Server

The API Server is highly configurable and customizable. It can scale horizontally, but it also supports the addition of custom secondary API Servers, a configuration that transforms the primary API Server into a proxy to all secondary, custom API Servers, routing all incoming RESTful calls to them based on custom defined rules.

Scheduler

The role of the kube-scheduler is to assign new workload objects, such as pods encapsulating containers, to nodes - typically worker nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new workload object's requirements.

Controller managers

The controller managers are components of the control plane node running controllers or operator processes to regulate the state of the Kubernetes cluster. Controllers are watch-loop processes continuously running and comparing the cluster's desired state

Nodes

A worker node provides a running environment for client applications. These applications are microservices running as application containers. In Kubernetes the application containers are encapsulated in Pods, controlled by the cluster control plane agents running on the control plane node. Pods are scheduled on worker nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world.

A worker node has the following components: container runtime, node agent - kubelet, kubelet - CRI shims, proxy - kube-proxy, add-ons (for DNS, observability components such as dashboards, cluster-level monitoring and logging, and device plugins).

Container runtime

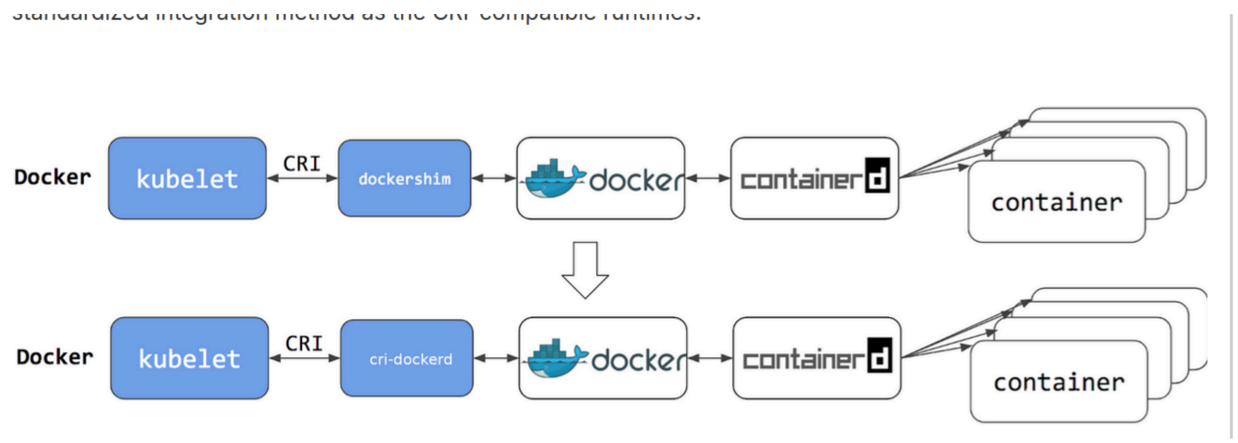
Although Kubernetes is described as a "container orchestration engine", it lacks the capability to directly handle and run containers. In order to manage a container's lifecycle, Kubernetes requires a container runtime on the node where a Pod and its containers are to be scheduled.

Node agent - kubelet

The kubelet is an agent running on each node, control plane and workers, and it communicates with the control plane. It receives Pod definitions, primarily from the API Server, and interacts with the container runtime on the node to run containers associated with the Pod.

kubelet - CRI shims

Originally the kubelet agent supported only a couple of container runtimes, first the Docker Engine followed by rkt, through a unique interface model integrated directly in the kubelet source code.



Proxy - kube-proxy

The kube-proxy is the network agent which runs on each node, control plane and workers, responsible for dynamic updates and maintenance of all networking rules on the node. It abstracts the details of Pods networking and forwards connection requests to the containers in the Pods.

Networking Challenges

Decoupled microservices based applications rely heavily on networking in order to mimic the tight-coupling once available in the monolithic era.

Container-to-Container communication inside Pods

Making use of the underlying host operating system's kernel virtualization features, a container runtime creates an isolated network space for each container it starts. On Linux, this isolated network space is referred to as a network namespace. A network namespace can be shared across containers, or with the host operating system.

*The Kubernetes network model aims to reduce complexity, and it treats Pods as VMs on a network, where each VM is equipped with a network interface - thus each Pod receiving a unique IP address. This model is called "**IP-per-Pod**" and ensures Pod-to-Pod communication, just as VMs are able to communicate with each other on the same network.*

External-to-Pod communication

A successfully deployed containerized application running in Pods inside a Kubernetes cluster may require accessibility from the outside world. Kubernetes enables external accessibility through Services, complex encapsulations of network routing rule definitions stored in iptables on cluster nodes and implemented by kube-proxy agents. By exposing services to the external world with the aid of kube-proxy, applications become accessible from outside the cluster over a virtual IP address and a dedicated port number.

```
$ minikube status
```

```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Stop Minikube. With the `minikube stop` command, we can stop Minikube. This command stops all applications running in Minikube, safely stops the cluster and the VirtualBox VM, preserving our work until we decide to start the Minikube cluster once again, while preserving the Minikube VM:

```
$ minikube stop
```

```
✋  Stopping node "minikube"  ...
🛑  1 node stopped.
```

When it is time to run the cluster again, simply run the **`minikube start`** *command (driver option is not required), and it will restart the earlier bootstrapped Minikube cluster.*

Remove Minikube. The **`minikube delete`** *command completely removes Minikube and the Minikube VM. This command should be attempted only when the Minikube cluster is to be decommissioned. All work will be lost after the completion of this command:*

```
$ minikube delete
```

```
🔥 Deleting "minikube" in virtualbox ...
 💀 Removed all traces of the "minikube" cluster.
```

The **`minikube profile`** *command allows us to view the status of all our clusters in a table formatted output. Assuming we have created only the default minikube cluster, we could list the properties that define the default profile with:*

```
$ minikube profile list
```

A command that allows users to list the nodes of a cluster, add new control plane or worker nodes, delete existing cluster nodes, start or stop individual nodes of a cluster:

```
$ minikube node list
```

```
$ minikube delete
```

```
🔥  Deleting "minikube" in virtualbox ...
💀  Removed all traces of the "minikube" cluster.
```

```
$ minikube delete -p minibox
```

```
🔥  Deleting "minibox" in virtualbox ...
🔥  Deleting "minibox-m02" in virtualbox ...
🔥  Deleting "minibox-m03" in virtualbox ...
💀  Removed all traces of the "minibox" cluster.
```

APIs

The main component of the Kubernetes control plane is the API Server, responsible for exposing the Kubernetes APIs. The APIs allow operators and users to directly interact with the cluster. Using both CLI tools and the Dashboard UI,

```
kubectl config view
```

```
kubectl cluster-info
```

```
minikube start --nodes 3 -p k8cluster

To switch the active cluster use

Minikube profile k8cluster
```

```
$ minikube addons list

$ minikube addons enable metrics-server

$ minikube addons enable dashboard

$ minikube addons list

$ minikube dashboard
```

Nodes

Worker nodes run the kubelet and kube-proxy node agents, the container runtime, and add-ons for container networking, monitoring, logging, DNS, etc.

Namespace

The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.

To list all the Namespaces, we can run the following command:

```
$ kubectl get namespaces
```

Good practice, however, is to create additional Namespaces, as desired, to virtualize the cluster and isolate users, developer teams, applications, or tiers:

```
$ kubectl create namespace new-namespace-name
```

It logical separates the resources like pods, services and deployment.

Pod

A Pod is the smallest Kubernetes workload object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, enclosing and isolating

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    run: nginx-pod
spec:
  containers:
  - name: nginx-pod
    image: nginx:1.22.1
    ports:
    - containerPort: 80
```

While **`create`** *is exemplified below, advanced Kubernetes practitioners may opt to use* **`apply`** *instead:*

```
$ kubectl create -f def-pod.yaml
```

Writing up definition manifests, especially complex ones, may prove to be quite time consuming and troublesome because YAML is extremely sensitive to indentation. When eventually editing such definition manifests keep in mind that each indent is two blank spaces wide, and **`TAB`** *should be omitted.*

Imperatively, we can simply run the Pod defined above without the definition manifest as such:

```
$ kubectl run nginx-pod --image=nginx:1.22.1 --port=80
```

The following is a multi-line command that should be selected in its entirety for copy/paste (including the backslash character "\"):

To create a yaml file direct use :-

```
$ kubectl run nginx-pod --image=nginx:1.22.1 --port=80 \
 --dry-run=client -o yaml > nginx-pod.yaml
```

After this use kubectl apply -f nginx-pod.yaml

The command above generates a definition manifest in YAML, but we can generate a JSON definition file just as easily with:

```
$ kubectl run nginx-pod --image=nginx:1.22.1 --port=80 \
 --dry-run=client -o json > nginx-pod.json
```

Both the YAML and JSON definition files can serve as templates or can be loaded into the cluster respectively as such:

```
$ kubectl create -f nginx-pod.yaml
 $ kubectl create -f nginx-pod.json
```

Before advancing to more complex application deployment and management methods, become familiar with Pod operations with additional commands such as:

```
$ kubectl apply -f nginx-pod.yaml
 $ kubectl get pods
 $ kubectl get pod nginx-pod -o yaml
 $ kubectl get pod nginx-pod -o json
 $ kubectl describe pod nginx-pod
 $ kubectl delete pod nginx-pod
```

If your the error as pullofimage error there might image pull error, make changes in the yaml file and then you use replace command it delete the existing pod and make the new one

Kubectl replace –force -f nginx-pod.yaml