Introduction to Kubernetes

What is container orchestration?

History of Kubernetes

source: https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/

Kubernetes (“K8s” or “Kube”, κυβερνήτης > Gr. "helmsman" or "pilot"), originally designed by Google, now maintained by the Cloud Native Computing Foundation (CNCF).

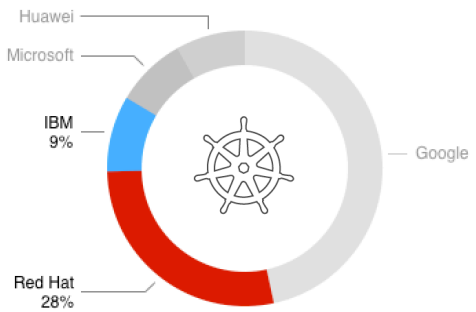
2003-2004, Borg System, Google’s proprietary specific cluster management system,
In 2013... container orchestration existed… but not in cloud and not in the enterprise. Docker changed all of that by popularizing a lightweight container runtime and providing a simple way to package, distribute and deploy applications.

The basic feature set for an orchestrator MVP was:

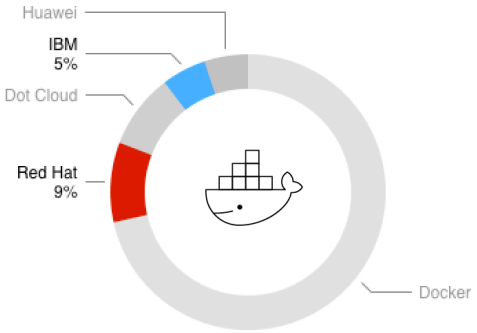
- Replication to deploy multiple instances of an application
- Load balancing and service discovery to route traffic to these replicated containers
- Basic health checking and repair to ensure a self-healing system
- Scheduling to group many machines into a single pool and distribute work to them

June, 2014 , Google open-sourced Kubernetes, as a generic cluster management system. The OpenShift team at Red Hat had joined even prior to launch.
July 21, 2015 , v1.0, Google partnered with the Linux Foundation to form the CNCF.
March 16, 2016, v1.2 introduced Ingress service type.
December 2016, v1.5 introduced Container Runtime Interface (CRI).
June 2017, v1.7 introduced Custom Resource Definitions (CRD)
December 2018, v1.13 introduced the GA release of the Container Storage Interface (CSI)
Current contributors: (1) Google, (2) VMWare, (3) IBM & RedHat, (4) Microsoft, (5) Independent.

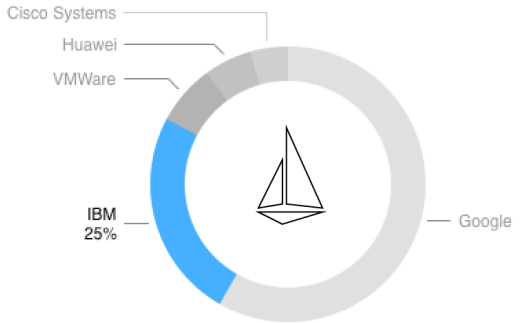
IBM and RedHat contributions to Kubernetes, Docker and Istio projects (top 5 orgs, apart from independent developers)



Top 5 organizations (apart from independent developers) who contribute to **Kubernetes** Open Source Projects



Top 5 organizations (apart from independent developers) who contribute to **Docker** Open Source Projects



Top 5 organizations (apart from independent developers) who contribute to **Istio** Open Source Projects

Container Orchestration

- Provision, manage, scale containers
- Manage resources
  - Volumes
  - Networks
  - Secrets
  - Environment Variables
  - Secrets and Configuration management
- Replication
- Service discovery
- Health management and self-healing
- Cluster management
- Scheduling and automated rollouts
- Declarative state management
- Extensions: Custom Resources and Operators

Benefits:

- Automated scheduling and scaling
- Zero downtime deployments
- High availability and fault tolerance
- A/B deployments

Kubernetes Architecture

- At its core, Kubernetes is a database (etcd) with "watchers" and "controllers" that react to changes in etcd,
- The kube-api-server is an API server that exposes the Kubernetes API,
- The kube-scheduler watches for new Pods not assigned to a node and selects a node to run the Pod on,
- The kube-controller-manager runs the controller loops that make Kubernetes,
- The cloud-controller-manager interacts with the underlying cloud provider,
- The etcd key-value store represents the user defined desired state of the objects,
- The kubelet makes sure that containers are running in a Pod,
- The kube-proxy is a network proxy that maintains network rules on nodes,

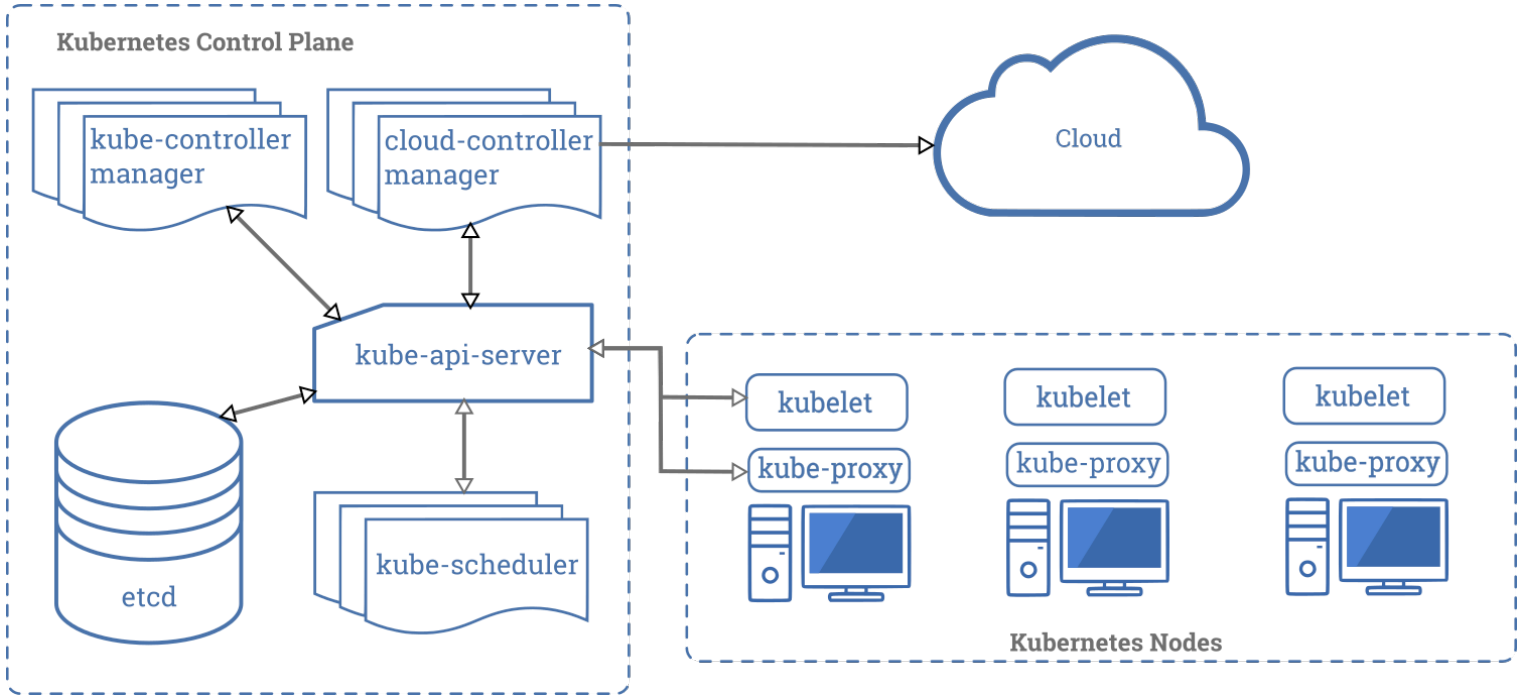


image source: https://kubernetes.io/docs/concepts/overview/components/

Kubernetes Objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.

Each Kubernetes object includes metadata and two nested object fields:

- the object ”spec” describes the desired state, provided in a .yaml file when you create an object,
- the object ”status” describes the current state.

All Kubernetes objects are considered an API resource and have a corresponding endpoint in the Kubernetes API.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: guestbook-v1
  namespace: default
  labels:
    app: guestbook
    version: ‘1.0’
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
    spec:
      containers:
      - name: guestbook
          image: ibmcom/guestbook:v1
          ports:
            - name: http-port
              containerPort: 80
status:
  replicas: 3
  readyReplicas: 3
  availableReplicas: 3
  conditions:
    - type: Available
…
```

API Groups

All Kubernetes objects are considered an API resource and have a corresponding endpoint in the Kubernetes API.

The Kubernetes API is divided into API Groups to make it easier to extend the API, disabling APIs, supporting different versions, support API Plugin.

API groups:

- core,
- apps,
- extensions,
- batch,
- autoscaling,
- storage.k8s.io,
- admissionregistration.k8s.io,
- apiextensions.k8s.io,
- policy,
- scheduling.k8s.io,
- settings.k8s.io,
- apiregistration.k8s.io,
- certificates.k8s.io,
- rbac.authorization.k8s.io,
- authorization.k8s.io,
- networking.k8s.io,
- auditregistration.k8s.io

Kubernetes API - *core*

source: https://kubernetes.io/docs/reference/#api-reference

- Workloads APIs
  - Pod
  - Container
  - ReplicationController
- Service APIs
  - Endpoints
  - Service
- Config and Storage APIs
  - ConfigMap
  - Volume
  - PersistentVolumeClaim
- MetaData APIs
  - Event
  - LimitRange
  - PodTemplate
- Cluster APIs
  - Binding
  - ComponentStatus
  - Namespace
  - Node
  - PersistentVolume
  - ResourceQuota
  - ServiceAccount

apps API Group: deployment.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
   name: my-app-deployment
   namespace: my-ns
   labels:
      app: my-app
spec:
   replicas: 1
   selector:
      matchLabels:
         app: my-app
   template:
      metadata:
         labels:
            app: my-app
      spec:
         containers:
         -  name: my-app
            image: remkohdev/my-app:latest
            ports:
            -  name: main
               protocol: TCP
               containerPort: 6661
            envFrom:
            - configMapRef:
                 name: my-app-configmap
            resources:
               requests:
                  memory: "120M"
                  cpu: "500m"
```

Kubernetes API – *apps,rbac.authorization.k8s.io*

apps

- Workloads APIs
  - DaemonSet
  - Deployment
  - ReplicaSet
  - StatefulSet
- MetaData APIs
  - ControllerRevision

rbac.authorization.k8s.io

- Cluster
  - ClusterRole
  - ClusterRoleBinding
  - Role
  - RoleBinding

apiextensions

- MetaData APIs
  - CustomResourceDefinition

autoscaling

- MetaData APIs
  - HorizontalPodAutoscaler

ServiceTypes

Kubernetes ServiceTypes allow you to specify what kind of Service you want. The default is ClusterIP.

Type values and their behaviors are:

- **ClusterIP**: Exposes the Service on a cluster-internal IP. This is the default ServiceType.
- **NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- **ExternalName**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record.

You can also use Ingress in place of Service to expose HTTP/HTTPS Services.

OpenShift extended Networking with a Route resource.

Environment Configuration

Environment variables in the container environment can be set in 4 ways:

(1) literal key-value pair,

(2) Secret,

(3) ConfigMap, or

(4) Pod property using valueFrom.fieldRef.fieldPath.

Extending Kubernetes

Customization can be divided into:
- configuration,
- Extensions.

You can extend the Kubernetes by using the Controller pattern or the webhook model.
Controllers read a “spec”, do work, and then update the “status”.
When Kubernetes is the client that calls out to a remote service, it is called a Webhook and the remote service is the Webhook Backend.

Extension points:
1. Kubectl, kubectl plugins,
2. Extensions in the apiserver allow authenticating or blocking requests, editing content and handling deletion,
3. Custom Resources (CR) using CustomResourceDefinition API and Operators,
4. Scheduler Extensions,
5. Controllers are often used with CR,
6. Node-level Network Plugins, Container Network Interface (CNI) Plugins or Kubenet plugins,
7. Storage Plugins, Container Storage Interface (CSI) spec.

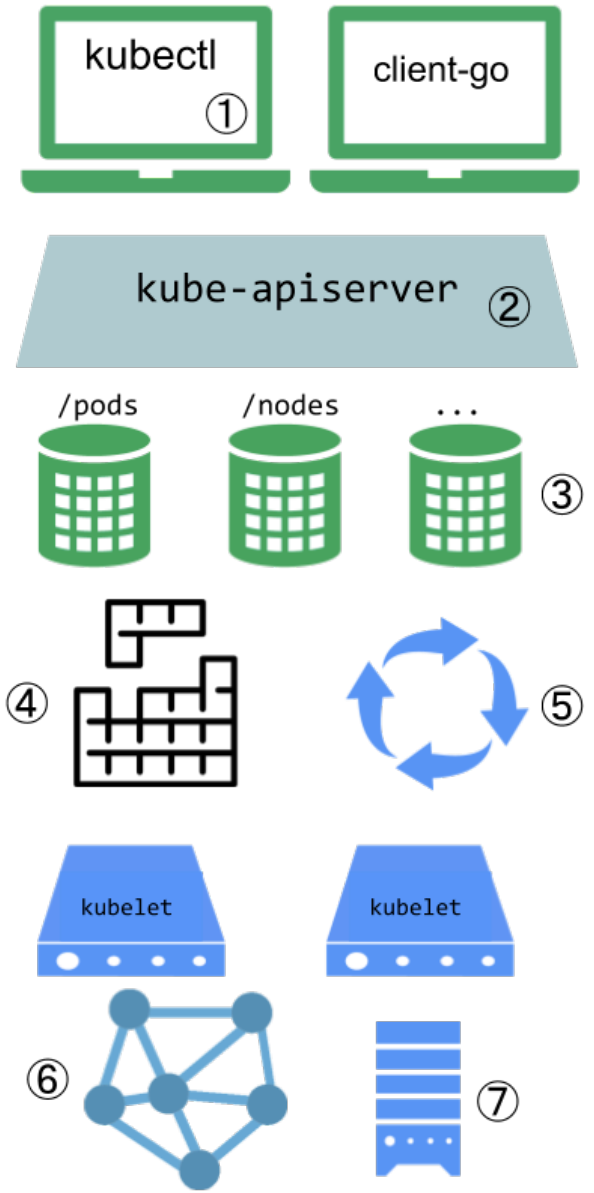


image source: https://kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster/

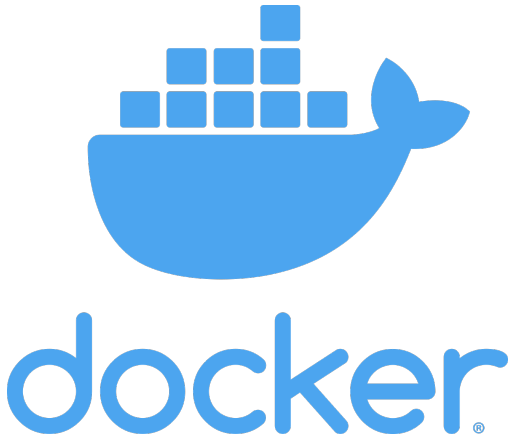
Kubernetes Container Runtime

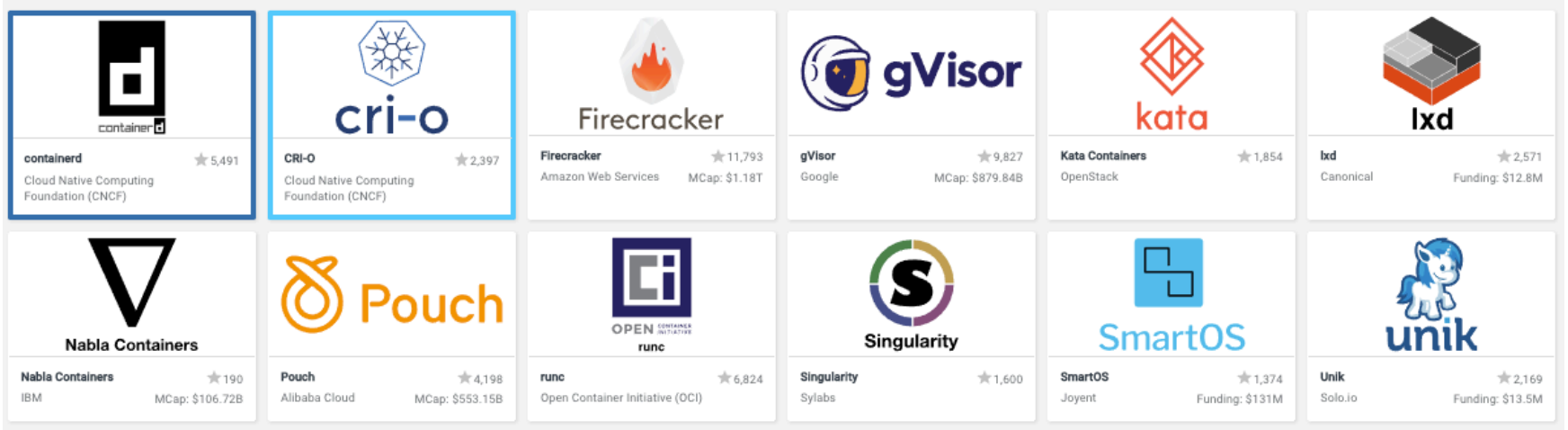
Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface), like dockerd, containerd, runc, Kata, Firecracker, singularity.

Kubernetes originally levered Docker for running containers. In December 2014, CoreOS (now RedHat) released “rkt” as an alternative to Docker and initiated *app container* (appc) and *application container image* (ACI) as independent committee-steered specifications, developed into OCI. Kubernetes 1.3 introduced rktnetes that enabled rkt.

On June 22, 2015 the Open Container Initiative (OCI) was announced, formed under the Linux Foundation and launched by Docker, CoreOS and others. The OCI currently contains a Runtime Specification (runtime-spec) and an Image Specification (image-spec).

In December 2016, Kubernetes v1.5 introduced Container Runtime Interface (CRI). Interaction between Kubernetes and any given runtime must use the CRI API. CRI-O was the first container runtime created for the Kubernetes CRI interface.





Kubernetes Client

CLI tool to interact with Kubernetes cluster

Platform specific binary available to download

- https://kubernetes.io/docs/tasks/tools/install-kubectl

The user directly manipulates resources via json/yaml

```
$ kubectl (create|get|apply|delete) -f myResource.yaml
```

Deploy

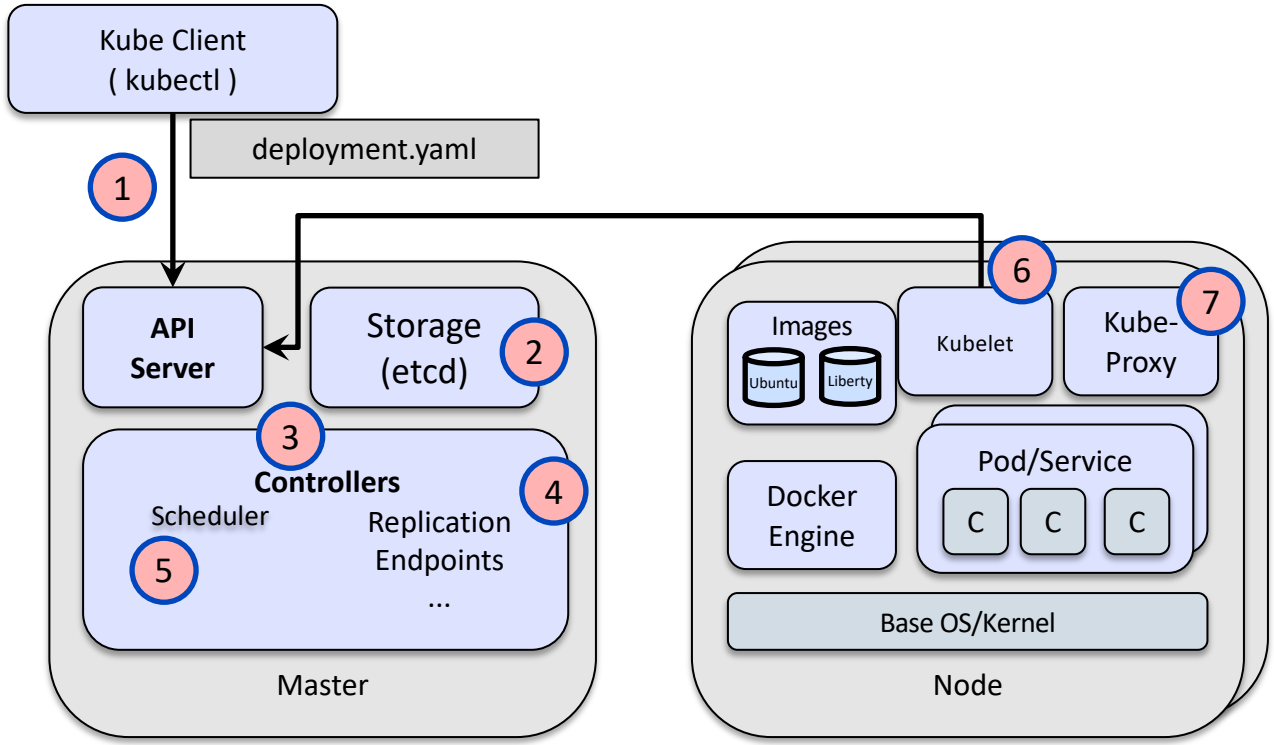
```
# configmap
kubectl delete configmap -n my-ns my-app-configmap
kubectl create -f ./helm/templates/configmap.yaml

# deployment
kubectl delete deployment -n my-ns my-app-deployment
kubectl create -f ./helm/templates/deployment.yaml

# service
kubectl delete svc -n my-ns my-app-svc
kubectl create -f ./helm/templates/svc.yaml
```

Kubernetes in Action

1. User via "kubectl" deploys a new application
2. API server receives the request and stores it in the DB (etcd)
3. Watchers/controllers detect the resource changes and act upon it
4. ReplicaSet watcher/controller detects the new app and creates new pods to match the desired # of instances
5. Scheduler assigns new pods to a kubelet
6. Kubelet detects pods and deploys them via the container runing (e.g. Docker)
7. Kubeproxy manages network traffic for the pods – including service discovery and load-balancing

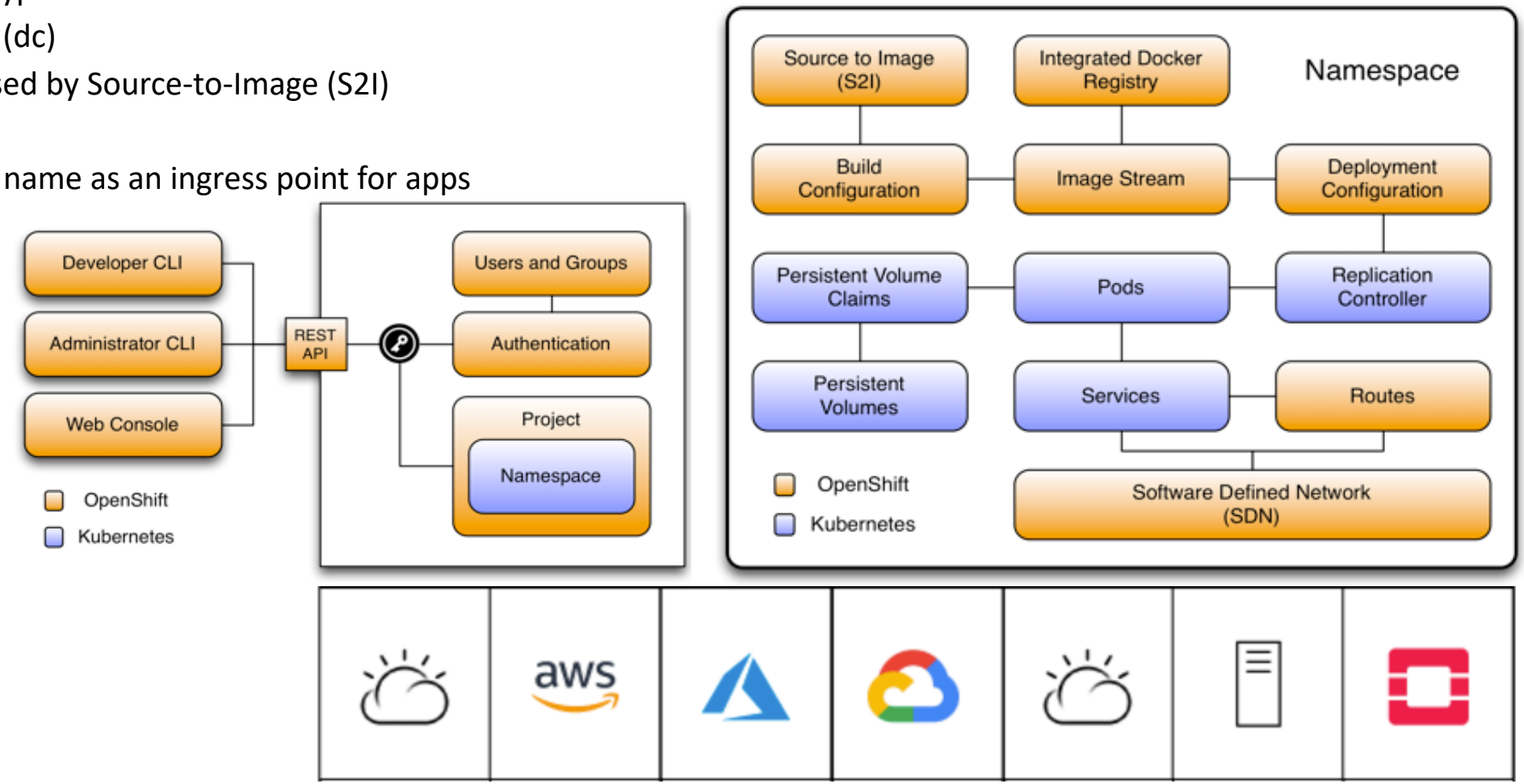


Enterprise Kubernetes - OpenShift

OpenShift is a set of modular components and services built on top of Kubernetes. From OpenShift v4, all hosts use RHEL CoreOS. OpenShift adds multitenancy, security (e.g. SELinux, CRIO), UI, and CI/CD. OpenShift 4 has CoreOS as the mandatory operating system for all nodes.

OpenShift Resource Types:

- Deployment config (dc)
- Build config (bc), used by Source-to-Image (S2I)
- ImageStream,
- Routes, a DNS host name as an ingress point for apps



Enterprise Kubernetes - OpenShift

DevOps Tools and User Experience
Web Console, CLI, REST API, SCM integration

Containerized Services
Auth, Networking, Image Registry

Runtimes and xPaaS
Java, Ruby, Node.js and more

Kubernetes
Container orchestration and management

Etcd
Cluster state and configs

CRDs
Kubernetes Operators

CRI-O
Container runtime

Red Hat CoreOS
Container optimized OS

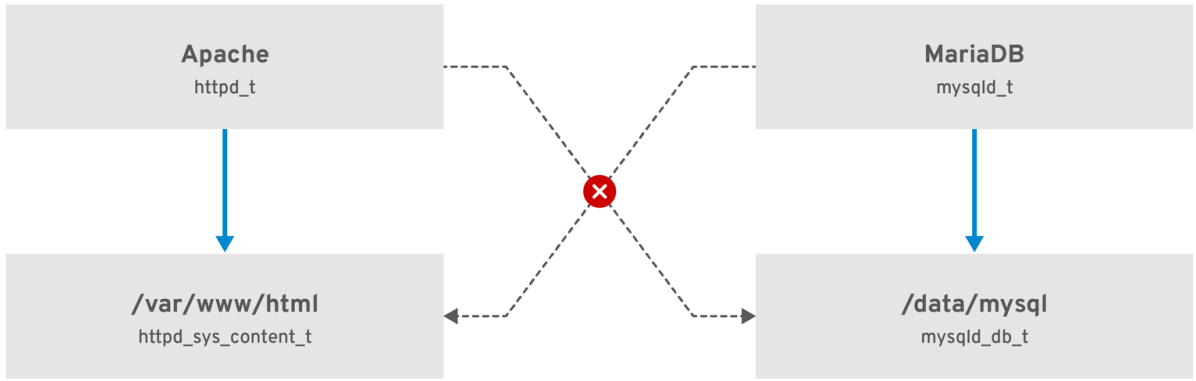
SELinux

SELinux provides enhanced security control over applications on OpenShift. SELinux implements Mandatory Access Control (MAC) in addition to Discretionary Access Control (DAC). DAC is the standard access policy based on the user, group and other permissions. MAC adds an SELinux context label to every process and system resource. By default, the SELinux policy does not allow any interaction between processes and resources unless a rule explicitly grants access. SELinux policy rules are checked after DAC rules.

SELinux contexts have several fields: user, role, type, and security level. The most common policy rule uses SELinux types and not the full SELinux context. **SELinux types usually end with *_t***. For example, the type name for the web server is httpd_t. The type context for files and directories normally found in /var/www/html/ is httpd_sys_content_t. The type contexts for files and directories normally found in /tmp and /var/tmp/ is tmp_t. The type context for web server ports is http_port_t.

Local folders used for container volume mounts must satisfy the following:

- The user executing the container processes must be the owner of the folder, or have the necessary rights. Use the *chown* command to update folder ownership.
- The local folder must satisfy the SELinux requirements to be used as a container volume. Assign the container_file_t group to the folder by using the *semanage fcontext -a -t container_file_t <folder>* command, then refresh the permissions with the *restorecon -R <folder>* command



The End