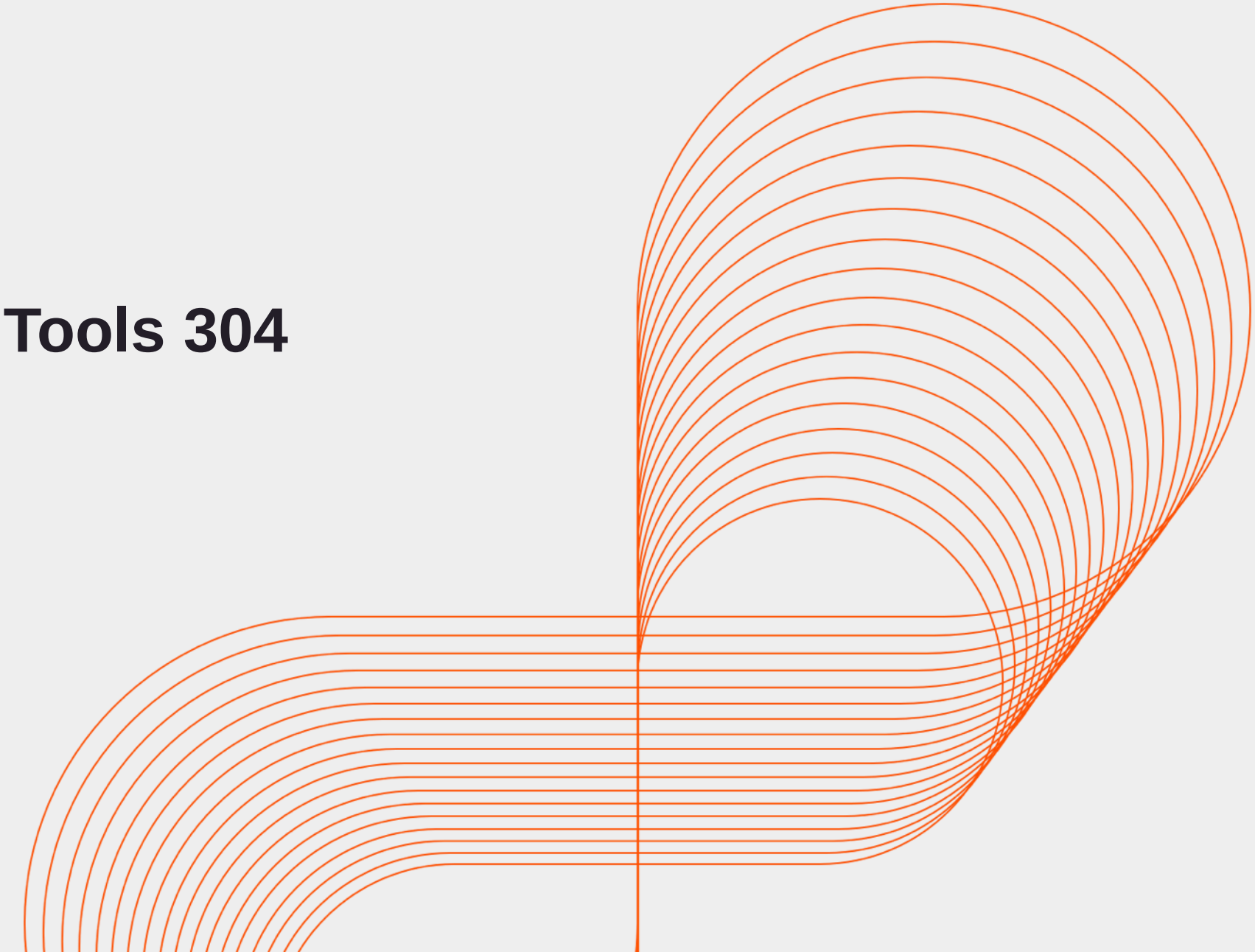
Persistent

Selfie Shots - Tools 304 Kubernetes

Persistent University



Kubernetes Architecture

Key learning points :

- Components
- Networking Setup
- Pod-to-Pod communication

Main Components

- **Kubernetes** has the following main components:
  - Master and worker nodes
  - Controllers
  - Services
  - Pods of containers
  - Namespaces and quotas
  - Network and policies
  - Storage.
- A **Kubernetes** cluster is made of a master node and a set of worker nodes. The cluster is all driven via API calls to controllers, both interior as well as exterior traffic. We will take a closer look at these components next.
- Most of the processes are executed inside a container. There are some differences, depending on the vendor and the tool used to build the cluster.

Master Node

- The Kubernetes master runs various server and manager processes for the cluster. Among the components of the master node are the **kube-apiserver**, the **kube-scheduler**, and the **etcd** database. As the software has matured, new components have been created to handle dedicated needs, such as the **cloud-controller-manager**; it handles tasks once handled by the **kube-controller-manager** to interact with other tools, such as **Rancher** or **DigitalOcean** for third-party cluster management and reporting.
- There are several add-ons which have become essential to a typical production cluster, such as DNS services. Others are third-party solutions where Kubernetes has not yet developed a local component, such as cluster-level logging and resource monitoring.

Worker Nodes

- All worker nodes run the **kubelet** and **kube-proxy**, as well as the container engine, such as **Docker** or **rkt**. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.
- The **kubelet** interacts with the underlying Docker Engine also installed on all the nodes, and makes sure that the containers that need to run are actually running. The **kube-proxy** is in charge of managing the network connectivity to the containers. It does so through the use of **iptables** entries. It also has the **userspace** mode, in which it monitors *Services* and *Endpoints* using a random port to proxy traffic and an alpha feature of **ipvs**.
- You can also run an alternative to the **Docker** engine: **rkt** by **CoreOS**. To learn how you can do that, you should check the documentation. In future releases, it is highly likely that Kubernetes will support additional container runtime engines.
- **Supervisord** is a lightweight process monitor used in traditional Linux environments to monitor and notify about other processes. In the cluster, this daemon monitors both the **kubelet** and **docker** processes. It will try to restart them if they fail, and log events.
- Kubernetes does not have cluster-wide logging yet. Instead, another **CNCF** project is used, called Fluentd. When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages

kube-apiserver

- The **kube-apiserver** is central to the operation of the Kubernetes cluster.
- All calls, both internal and external traffic, are handled via this agent. All actions are accepted and validated by this agent, and it is the only connection to the **etcd** database. As a result, it acts as a master process for the entire cluster, and acts as a frontend of the cluster's shared state.

kube-scheduler

- The **kube-scheduler** uses an algorithm to determine which node will host a Pod of containers. The scheduler will try to view available resources (such as volumes) to bind, and then try and retry to deploy the Pod based on availability and success.
- There are several ways you can affect the algorithm, or a custom scheduler could be used instead. You can also bind a Pod to a particular node, though the Pod may remain in a pending state due to other settings.
- One of the first settings referenced is if the Pod can be deployed within the current quota restrictions. If so, then the taints and tolerations, and labels of the Pods are used along with those of the nodes to determine the proper placement.

etcd Database

- The state of the cluster, networking, and other persistent information is kept in an **etcd** database, or, more accurately, a b+tree key-value store. Rather than finding and changing an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process. It works with **curl** and other HTTP libraries, and provides reliable watch queries.
- Simultaneous requests to update a value all travel via the **kube-apiserver**, which then passes along the request to **etcd** in a series. The first request would update the database. The second request would no longer have the same version number, in which case the **kube-apiserver** would reply with an error 409 to the requester. There is no logic past that response on the server side, meaning the client needs to expect this and act upon the denial to update.
- There is a **master** database along with possible **followers**. They communicate with each other on an ongoing basis to determine which will be **master**, and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with new tools, such as **kubeadm**, and features like whole cluster upgrades

kubelet

- The **kubelet** agent is the heavy lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications (a **PodSpec** is a JSON or YAML file that describes a pod). It will work to configure the local node until the specification has been met.
- Should a Pod require access to storage, Secrets or ConfigMaps, the **kubelet** will ensure access or creation. It also sends back status to the **kube-apiserver** for eventual persistence.

Other Agents

- The **kube-controller-manager** is a core control loop daemon which interacts with the **kube-apiserver** to determine the state of the cluster. If the state does not match, the manager will contact the necessary controller to match the desired state. There are several controllers in use, such as endpoints, namespace, and replication. The full list has expanded as Kubernetes has matured.
- In **alpha** since v1.8, the **cloud-controller-manager** interacts with agents outside of the cloud. It handles tasks once handled by **kube-controller-manager**. This allows faster changes without altering the core Kubernetes control process. Each **kubelet** must use the **--cloud-provider-external** settings passed to the binary.

Pods

- The whole point of Kubernetes is to orchestrate the lifecycle of a container. We do not interact with particular containers. Instead, the smallest unit we can work with is a Pod. Some would say a pod of whales or peas-in-a-pod. Due to shared resources, the design of a Pod typically follows a one-process-per-container architecture.
- Containers in a Pod are started in parallel. As a result, there is no way to determine which container becomes available first inside a pod. To support a single process running in a container, you may need logging, a proxy, or special adapter. These tasks are often handled by other containers in the same pod.
- There is only one IP address per Pod. If there is more than one container, they must share the IP. To communicate with each other, they can either use IPC, or a shared filesystem.
- While Pods are often deployed with one application container in each, a common reason to have multiple containers in a Pod is for logging. You may find the term **sidecar** for a container dedicated to performing a helper task, like handling logs and responding to requests, as the primary application container may have this ability.

Containers

- While Kubernetes orchestration does not allow direct manipulation on a container level, we can manage the resources containers are allowed to consume.
- In the **resources** section of the **PodSpec** you can pass parameters which will be passed to the container runtime on the scheduled node:

```
resources:
  limits:
    cpu: "1"
    memory: "4Gi"
  requests:
    cpu: "0.5"
    memory: "500Mi"
```

- Another way to manage resource usage of the containers is by creating a **ResourceQuota** object, which allows hard and soft limits to be set in a namespace. The quotas allow management of more resources than just CPU and memory and allows limiting several objects.
- A beta feature in v1.11 uses the **scopeSelector** field in the quota spec to run a pod at a specific priority if it has the appropriate **priorityClassName** in its pod spec.

Services

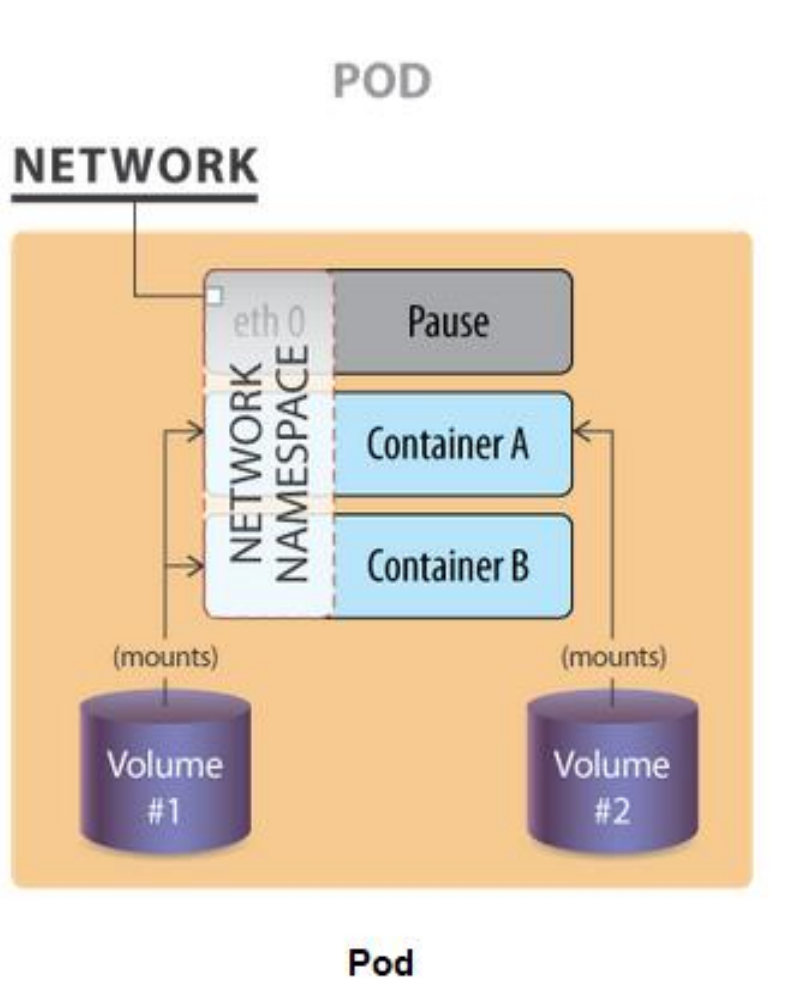
- With every object and agent decoupled we need a flexible and scalable agent which connects resources together and will reconnect, should something die and a replacement is spawned. Each *Service* is a microservice handling a particular bit of traffic, such as a single **NodePort** or a **LoadBalancer** to distribute inbound requests among many Pods.
- A *Service* also handles access policies for inbound requests, useful for resource control, as well as for security.

Controller

- An important concept for orchestration is the use of controllers. Various controllers ship with Kubernetes, and you can create your own, as well. A simplified view of a controller is an agent, or **Informer**, and a downstream store. Using a **DeltaFIFO** queue, the source and downstream are compared. A loop process receives an **obj** or object, which is an array of deltas from the FIFO queue. As long as the delta is not of the type **Deleted**, the logic of the controller is used to create or modify some object until it matches the specification.
- The **Informer** which uses the API server as a source requests the state of an object via an API call. The data is cached to minimize API server transactions. A similar agent is the **SharedInformer**; objects are often used by multiple other objects. It creates a shared cache of the state for multiple requests.
- A **Workqueue** uses a key to hand out tasks to various workers. The standard **Go** work queues of rate limiting, delayed, and time queue are typically used.
- The **endpoints**, **namespace**, and **serviceaccounts** controllers each manage the eponymous resources for Pods.

Single IP per Pod

- A pod represents a group of co-located containers with some associated data volumes. All containers in a pod share the same network **namespace**.
- The graphic shows a pod with two containers, A and B, and two data volumes, 1 and 2. Containers A and B share the network namespace of a third container, known as the ***pause* container**. The pause container is used to get an IP address, then all the containers in the pod will use its network namespace. Volumes 1 and 2 are shown for completeness.
- To communicate with each other, Pods can use the loopback interface, write to files on a common filesystem, or via inter-process communication (IPC).



Networking Setup

- Getting all the previous components running is a common task for system administrators who are accustomed to configuration management. But, to get a fully functional **Kubernetes** cluster, the network will need to be set up properly, as well.
- A detailed explanation about the **Kubernetes** networking model can be seen on the Cluster Networking page in the Kubernetes documentation.
- If you have experience deploying virtual machines (VMs) based on IaaS solutions, this will sound familiar. The only caveat is that, in **Kubernetes**, the lowest compute unit is not a **container**, but what we call a **pod**.
- A pod is a group of co-located containers that share the same IP address. From a networking perspective, a pod can be seen as a virtual machine of physical hosts. The network needs to assign IP addresses to pods, and needs to provide traffic routes between all pods on any nodes.
- The three main networking challenges to solve in a container orchestration system are:
- Coupled container-to-container communications (solved by the pod concept).
- Pod-to-pod communications.
- External-to-pod communications (solved by the services concept, which we will discuss later).
- **Kubernetes** expects the network configuration to enable pod-to-pod communications to be available; it will not do it for you.

CNI Network Configuration File

- To provide container networking, **Kubernetes** is standardizing on the Container Network Interface (CNI) specification. As of v1.6.0, **kubeadm** (the **Kubernetes** cluster bootstrapping tool) uses CNI as the default network interface mechanism.
- CNI is an emerging specification with associated libraries to write plugins that configure container networking and remove allocated resources when the container is deleted. Its aim is to provide a common interface between the various networking solutions and container runtimes. As the CNI specification is language-agnostic, there are many plugins from Amazon ECS, to SR-IOV, to Cloud Foundry, and more.
- The adjacent configuration defines a standard Linux bridge named cni0, which will give out IP addresses in the subnet 10.22.0.0./16. The bridge plugin will configure the network interfaces in the correct namespaces to define the container network properly.

```
{
   "cniVersion": "0.2.0",
   "name": "mynet",
   "type": "bridge",
   "bridge": "cni0",
   "isGateway": true,
   "ipMasq": true,
   "ipam": {
      "type": "host-local",
      "subnet": "10.22.0.0/16",
      "routes": [
         { "dst": "0.0.0.0/0" }
      ]
   }
}
```

Pod-to-Pod Communication

- While a CNI plugin can be used to configure the network of a pod and provide a single IP per pod, CNI does not help you with pod-to-pod communication across nodes.
- The requirement from **Kubernetes** is the following:
- All pods can communicate with each other across nodes.
- All nodes can communicate with all pods.
- No Network Address Translation (NAT).
- Basically, all IPs involved (nodes and pods) are routable without NAT. This can be achieved at the physical network infrastructure if you have access to it (e.g. GKE). Or, this can be achieved with a software defined overlay with solutions like:
  - Weave
  - Flannel
  - Calico
  - Romana.
- See this documentation page or the list of networking add-ons for a more complete list.

Summary

At the end of this session, we see that you are now able to

- Discuss the main components of a Kubernetes cluster.
- Learn details of the master agent kube-apiserver.
- Explain how the etcd database keeps the cluster state and configuration.
- Study the kubelet local agent.
- Examine how controllers are used to manage the cluster state.
- Discover what a Pod is to the cluster.
- Examine network configurations of a cluster.

Lab Exercise

- Read about Pod Networking
- Read about CNI in Kubernetes Documentation