Kubernetes brown bag

- history lesson
- what is k8s
- concepts
- components
- interactive

Script

Before Kubernetes was Kubernetes, Google engineers embarked on an internal project to build a large scale cluster manager that could run hundreds of thousands of jobs from several thousands of applications across a number of clusters each with up to tens of thousands of machines. The project was named “Borg”, ostensibly because the engineers working on the project were big Star Trek fans.

Following the theme from the prior project, the Kubernetes project was originally called “Project 7”, which was a reference to the “7 of 9” Borg character on one of the Star Trek series. They even took it a step further with the project logo by putting 7 spokes on the ship’s wheel.

Kubernetes launched at the now-defunct OSCON in the summer of 2015. Google worked with the Linux Foundation to form the Cloud Native Computing Foundation and offered up Kubernetes as an initial project, which is one of a few CNCF projects at the Graduated maturity level.

Google was offering managed Kubernetes services early on, along with RedHat as part of OpenShift, but principal competitors began rallying around Kubernetes and announced adding native support for it a couple of years after release, including: VMware, Mesosphere Inc, Docker Inc, Azure, and AWS.

Now, what is Kubernetes? The short answer is ‘a container orchestrator’ but that is a bit too reductive. Kubernetes provides a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. Kubernetes offers:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Automated rollouts and rollbacks** You can describe the desired state

for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Extensibility** There are various ways to customize and extend Kubernetes. The CLI to the Kubernetes API, called kubectl, can be extended with plugins. The API server can be extended with Admission Controllers that can allow or deny the scheduling of workloads based on any number of factors. The API server provides several resources out of the box, but custom resources can be added based on the users' needs. And lower level plugins that handle network requests, storage, and hardware devices can be created.

Kubernetes core architecture consists of a control plane (green box) and data plane (blue boxes) with a management plane that is abstracted out as a means to enforce policy and governance over the environment. Let's instead focus on the diagram. The Kubernetes control plane consists of various components, each its own process, that can run both on a single master node or on multiple masters supporting high-availability clusters.[36] The various components of the Kubernetes control plane are as follows:

- **etcd**[37] is a persistent, lightweight, distributed, key-value data store that CoreOS has developed. It reliably stores the configuration data of the cluster, representing the overall state of the cluster at any given point of time. etcd favors consistency over availability in the event of a network partition (see CAP theorem). The consistency is crucial for correctly scheduling and operating services.
- The **API server** serves the Kubernetes API using JSON over HTTP, which provides both the internal and external interface to Kubernetes.[35] [38] The API server processes and validates REST requests and updates the

state of the API objects in etcd, thereby allowing clients to configure workloads and containers across worker nodes.[39] The API server uses etcd's watch API to monitor the cluster, roll out critical configuration changes, or restore any divergences of the state of the cluster back to what the deployer declared. As an example, the deployer may specify that three instances of a particular "pod" (see below) need to be running. etcd stores this fact. If the Deployment Controller finds that only two instances are running (conflicting with the etcd declaration),[40] it schedules the creation of an additional instance of that pod.[36]

- The **scheduler** is the extensible component that selects on which node an unscheduled pod (the basic entity managed by the scheduler) runs, based on resource availability. The scheduler tracks resource use on each node to ensure that workload is not scheduled in excess of available resources. For this purpose, the scheduler must know the resource requirements, resource availability, and other user-provided constraints or policy directives such as quality-of-service, affinity vs. anti-affinity requirements, and data locality. The scheduler's role is to match resource "supply" to workload "demand".[41]
- A **controller** is a reconciliation loop that drives the actual cluster state toward the desired state, communicating with the API server to create, update, and delete the resources it manages (e.g., pods or service endpoints).[42] [38] One kind of controller is a Replication Controller, which handles replication and scaling by running a specified number of copies of a pod across the cluster. It also handles creating replacement pods if the underlying node fails.[42] Other controllers that are part of the core Kubernetes system include a DaemonSet Controller for running exactly one pod on every machine (or some subset of machines), and a Job Controller for running pods that run to completion (e.g., as part of a batch job).[43] Labels selectors that are part of the controller's definition specify the set of pods that a controller manages.[44]
- The **controller manager** is a process that manages a set of core Kubernetes controllers.

Kubernetes Nodes host the data plane. Every node in the cluster must run a container runtime such as containerd, as well as a few other components, for communication with the primary for network configuration of these containers.

- **Kubelet** is responsible for the running state of each node, ensuring that all containers on the node are healthy. It takes care of starting, stopping, and maintaining application containers organized into pods as directed by the control plane.[35][45] Kubelet monitors the state of a pod, and if not in the

desired state, the pod re-deploys to the same node. Node status is relayed every few seconds via heartbeat messages to the primary. Once the primary detects a node failure, the Replication Controller observes this state change and launches pods on other healthy nodes.[46]

- **Kube-proxy** is an implementation of a network proxy and a load balancer, and it supports the service abstraction along with other networking operation.[35] It is responsible for routing traffic to the appropriate container based on IP and port number of the incoming request.
- **cAdvisor (Container Advisor)** provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide.

Let’s briefly go over a few Kubernetes components. A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled.

A Deployment provides declarative updates for Pods and ReplicaSets.
You describe a *desired state* in a Deployment, and the
Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

A Service is a method for exposing a network application that is running as one or more Pods in your cluster. A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You run code in Pods and you use a Service to make that set of Pods available on the network so that clients can interact with it. For example, consider a ETL backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves. The Service abstraction enables this decoupling.

We touched on Nodes earlier, but they're simply hosts of compute workloads that have a process called a kubelet running on them which interacts with the API server in the control plane. The kubelet takes Pod specs that are provided through various mechanisms—primarily the API server—and ensures the containers described in those specs are running and healthy. The scheduler in the control plane is responsible for deciding which Nodes are most appropriate in terms of available resources and custom restrictions put in place for assigning Pods. Kubelet is responsible for communicating with the control plane which nodes are Ready for Pods and whether their Node will Tolerate the Pod.

<tutorial time>