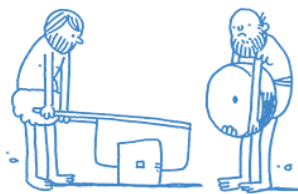


ONCE UPON A TIME,
SOFTWARE WAS WRITTEN
FOR SPECIFIC COMPUTERS.



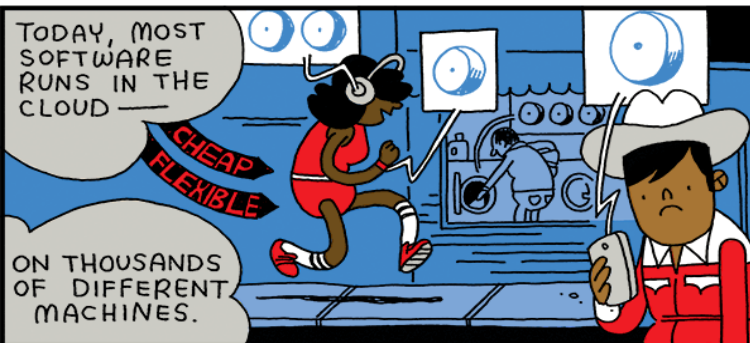
DIFFERENT TYPES OF
COMPUTERS REQUIRED
DIFFERENT VERSIONS OF
THE SAME APPLICATION.



TODAY, MOST
SOFTWARE
RUNS IN THE
CLOUD —

CHEAP
FLEXIBLE

ON THOUSANDS
OF DIFFERENT
MACHINES.

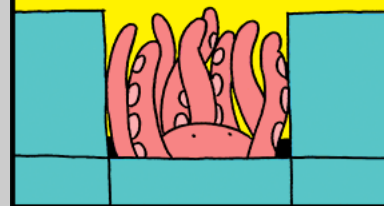


IT'S FROM A GREEK WORD
FOR HELMSMAN OR PILOT.
GEEKS JUST CALL
IT K8S.



Kubernetes Overview By: Utkarsh Rana

K8S WORKS BY
SORTING OUT HOW
TO STORE AND RUN CODE
MOST EFFICIENTLY.
IT'S A BIT LIKE TETRIS.



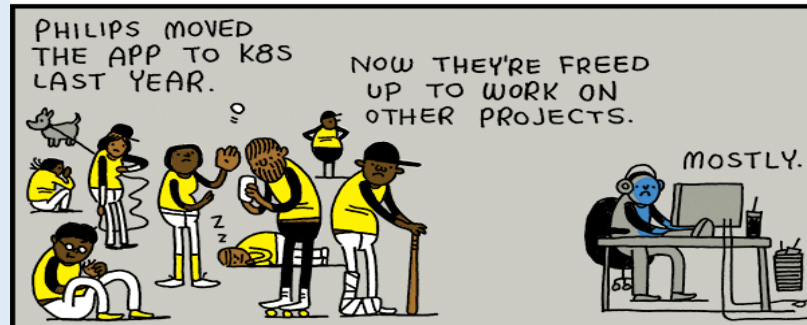
K8S ALSO LETS
COMPANIES SPLIT CLOUD
STORAGE OVER MULTIPLE
PROVIDERS OR EASILY
JUMP FROM ONE
TO ANOTHER.



K8S IS THE MAGIC
THAT MAKES LIGHTS
COME ON FASTER.



PHILIPS MOVED
THE APP TO K8S
LAST YEAR. NOW THEY'RE FREED
UP TO WORK ON
OTHER PROJECTS.
MOSTLY.





CLOUD NATIVE COMPUTING FOUNDATION



CNI allows different tools to provide overlay networks to multi-host container deployments. A user can use Weave, Contiv or Docker Network to provide networking services and swap them out as requirements dictate. Because all these services conform to CNI, the cost of switching is low, and users can try multiple solutions to find the best fit.

CSI functions the same way, but for the cloud-native application data layer, often called the persistence layer.

What is the Purpose of Cloud Native?

“Cloud Native” is the name of a particular approach to designing, building and running applications based on infrastructure-as-a-service. The overall objective is to improve speed, scalability and finally margin.

- Speed:** companies of all sizes now see strategic advantage in being able to move quickly and get ideas to market fast. By this we mean moving from months to get an idea into production to days or even hours.

- Scale:** as businesses grow, it becomes strategically necessary to support more users, in more locations, with a broader range of devices, while maintaining responsiveness, managing costs, and not falling over.

- Margin:** in the new world of infrastructure-as-a-service, a strategic goal may be to pay for additional resources only as they're needed – as new customers come online. Spending moves from up-front CAPEX (buying new machines in anticipation of success) to OPEX (paying for additional servers on-demand).



**Cloud Native disruption
can't be stopped**

The birth of Kubernetes

- **Kubernetes (K8s)** is an open source project that was released by Google in June, 2014.
- Google released the project as part of an effort to share their own infrastructure and technology advantage with the community at large.
- Google launches 2 billion containers a week in their infrastructure and has been using container technology for over a decade. Originally, they were building a system named **Borg**, now called **Omega**, to schedule their vast quantities of workloads



The result was the Kubernetes open-source project (you can at the end of the chapter).

- Since its initial release in 2014, K8s has undergone rapid development with contributions all across the open-source community, including Red Hat, VMware, and Canonical. The 1.0 release of Kubernetes went live in July, 2015.

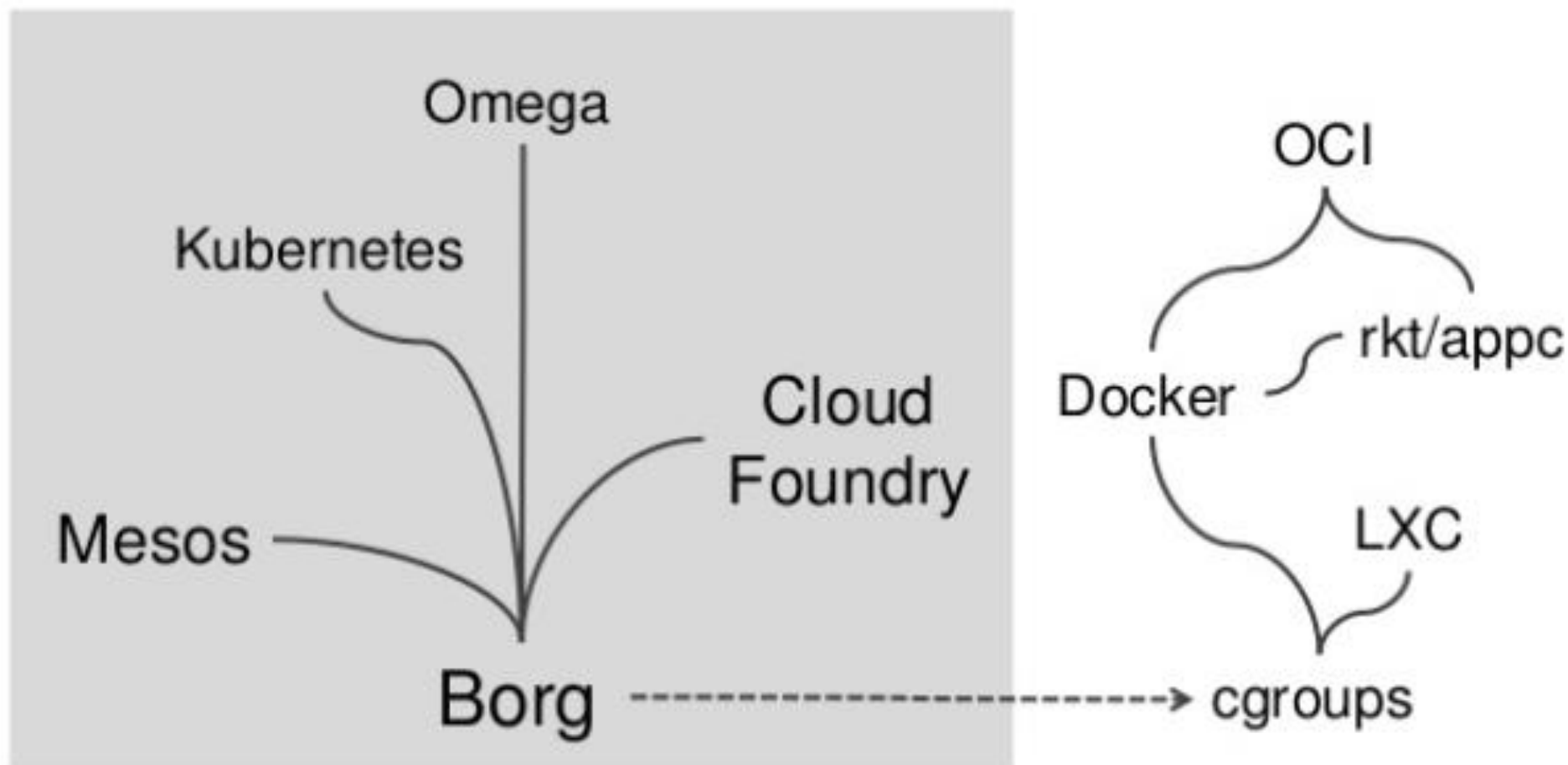
[Kubernetes](https://kubernetes.io) or k8s is an open-source system for automating deployment, scaling, and management of containerized applications. URL: <https://kubernetes.io>

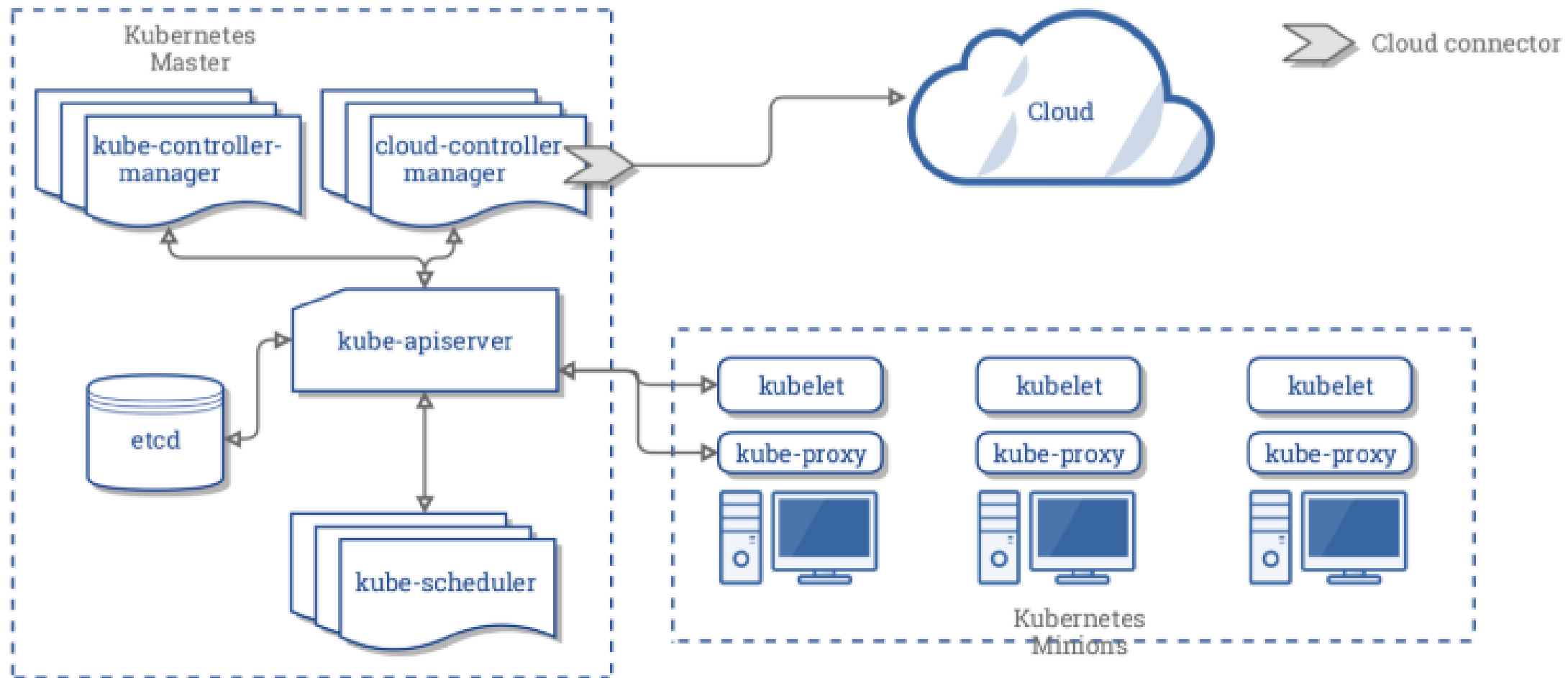
Kubernetes turned out to be the most popular standard for orchestration of containers in no time.

An IT infrastructure built on containers and orchestrated with Kubernetes is extremely powerful, scalable, redundant and efficient.

Kubernetes abstracts the underlying infrastructure by providing us with a simple API to which we can send requests.

The Borg Heritage





Kubernetes Architecture

Kubernetes Master Components

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.

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.

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.

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.

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.

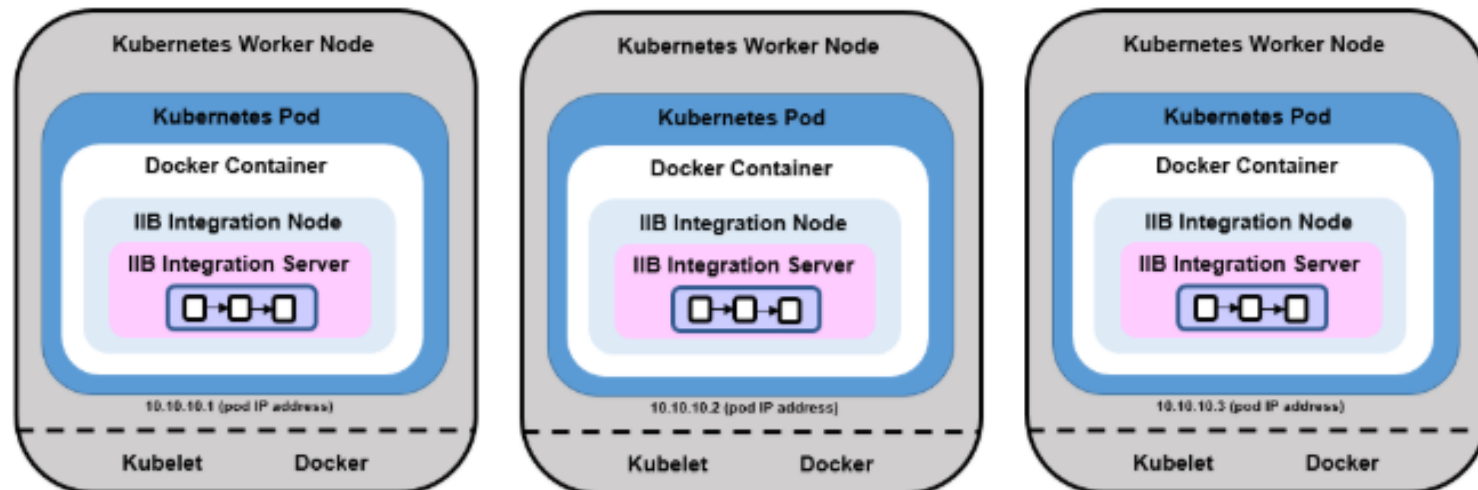
Kubernetes Worker Node/Minions Components

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**.

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.



Few Successful Adoptions

Kubernetes is being adopted at a very rapid pace. To learn more, you should check out the [case studies](#) presented on the **Kubernetes** website. Ebay, Box, Pearson and Wikimedia have all shared their stories.

[Pokemon Go](#), the fastest growing mobile game, also runs on **Google Container Engine (GKE)**, the **Kubernetes** service from **Google Cloud Platform (GCP)**.

Kubernetes Users



Minikube-Single Node Cluster

You can also use **Minikube**, an open source project within the **GitHub** [Kubernetes organization](#). While you can download a release from **GitHub**, following listed directions, it may be easier to download a pre-compiled binary. Make sure to verify and get the latest version.

For example, to get the v.0.22.2 version, do:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v0.22.2/minikube-linux-amd64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin
```

With **Minikube** now installed, starting **Kubernetes** on your local machine is very easy:

```
$ minikube start
$ kubectl get nodes
```

This will start a **VirtualBox** virtual machine that will contain a single node **Kubernetes** deployment and the **Docker** engine. Internally, **minikube** runs a single Go binary called **localkube**. This binary runs all the components of Kubernetes together. This makes Minikube simpler than a full Kubernetes deployment. In addition, the Minikube VM also runs Docker, in order to be able to run containers.

CNI Projects – k8s Network Policies

Prior to initializing the Kubernetes cluster, the network must be considered and IP conflicts avoided. There are several Pod networking choices, in varying levels of development and feature set:

- [Calico](#)
A flat Layer 3 network which communicates without IP encapsulation, used in production with software such as **Kubernetes**, **OpenShift**, **Docker**, **Mesos** and **OpenStack**. Viewed as a simple and flexible networking model, it scales well for large environments. Another network option, **Canal**, also part of this project, allows for integration with **Flannel**. Allows for implementation of network policies.
- [Flannel](#)
A Layer 3 IPv4 network between the nodes of a cluster. Developed by **CoreOS**, it has a long history with **Kubernetes**. Focused on traffic between hosts, not how containers configure local networking, it can use one of several backend mechanisms, such as VXLAN. A **flanneld** agent on each node allocates subnet leases for the host. While it can be configured after deployment, it is much easier prior to any Pods being added.
- [Kube-router](#)
Feature-filled single binary which claims to "*do it all*". The project is in the alpha stage, but promises to offer a distributed load balancer, firewall, and router purposely built for **Kubernetes**.
- [Romana](#)
Another project aimed at network and security automation for cloud native applications. Aimed at large clusters, IPAM-aware topology and integration with **kops** clusters.
- [Weave Net](#)
Typically used as an add-on for a CNI-enabled **Kubernetes** cluster.

Many of the projects will mention the Container Network Interface (CNI), which is a CNCF project. Several container runtimes currently use CNI. As a standard to handle deployment management and cleanup of network resources, it will become more popular.

Protocol	ingress-nginx	traefik	haproxy	istio ingress	Citrix Ingress Controller
backend service discovery	static (has generator)	dynamic	dynamic	dynamic	dynamic
protocol	http,https,tcp (separate lb),udp,grpc	http,https,grpc	http,tcp	http,https,grpc	http,https,tcp,ssl-tcp,udp
based on	nginx	traefik	haproxy	envoy	Citrix ADC
ssl termination	yes	yes	yes	yes	yes
websocket	yes	yes	yes	yes	yes
routing	host,path	host,path	host,path	host,user	host,path
scope	cross-namespace	cross-namespace	optional cross-namespace	cross-namespace	cross-namespace
resiliency	-	circuit break, retries	-	circuit break, retries	health check
lb algorithms	rr,least_conn,ip_hash	rr, wrr	rr, srr, leastconn,first,source,uri,url_param,hdr,rdp-cookie	rr,leastconn,random,passthrough	rr,least_conn,wrr,least_response,hash
auth	basic	basic	basic	-	basic
Tracing	yes	yes	-	yes	-
canary/shadow	-	canary	-	canary,shadow	canary
istio integration	-	-	-	yes	-
state	kubernetes	kubernetes	kubernetes	kubernetes	kubernetes
paid support	-	-	yes	-	
link	https://kubernetes.github.io/ingress-nginx/	https://docs.traefik.io/configuration/backends/kubernetes/	https://www.haproxy.com/blog/haproxy-ingress-controller-for-kubernetes/	https://istio.io/docs/tasks/traffic-management/ingress/	https://github.com/citrix/citrix-k8s-ingress-controller

Tools/Utilities for easy k8s Cluster Setup

- **kubespray**

kubespray is now in the **Kubernetes** incubator. It is an advanced **Ansible** playbook which allows you to setup a **Kubernetes** cluster on various operating systems and use different network providers. It was once known as **kargo**.

- **kops**

kops lets you create a **Kubernetes** cluster on **AWS** via a single command line. Also in beta for **GKE** and alpha for **VMware**.

- **kube-aws**

kube-aws is a command line tool that makes use of the **AWS Cloud Formation** to provision a **Kubernetes** cluster on **AWS**.

- **kubicorn**

kubicorn is a tool which leverages the use of **kubeadm** to build a cluster. It claims to have no dependency on DNS, runs on several operating systems, and uses snapshots to capture a cluster and move it.

Deployment Configurations of k8s Cluster

At a high level, you have four main deployment configurations:

- Single-node
- Single head node, multiple workers
- Multiple head nodes with HA, multiple workers
- HA **etcd**, HA head nodes, multiple workers.

Which of the four you will use will depend on how advanced you are in your **Kubernetes** journey, but also on what your goals are.

With a single-node deployment, all the components run on the same server. This is great for testing, learning, and developing around Kubernetes.

Adding more workers, a single head node and multiple workers typically will consist of a single node **etcd** instance running on the head node with the API, the scheduler, and the controller-manager.

Multiple head nodes in an HA configuration and multiple workers add more durability to the cluster. The API server will be fronted by a load balancer, the scheduler and the controller-manager will elect a leader (which is configured via flags). The **etcd** setup can still be single node.

The most advanced and resilient setup would be an HA **etcd** cluster, with HA head nodes and multiple workers. Also, **etcd** would run as a true cluster, which would provide HA and would run on nodes separate from the Kubernetes head nodes.

The use of Kubernetes Federations also offers high availability. Multiple clusters are joined together with a common control plane allowing movement of resources from one cluster to another administratively or after failure.

The Amazing Kubernetes Pods

**Kubernetes
starting up.**

YOU WILL SEE RANDOM FLASHES OF GREEN COLORS WHILE THE SERVICE IS UP.
MANAGE TO KILL THE SERVICE AND THIS BOX GOES RED.





 **hello-server**

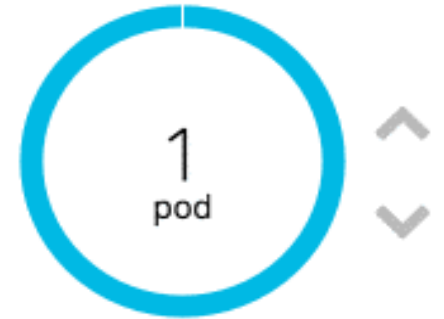
Deployment Config **hello-server** – 5 days ago

#1

CONTAINER: HELLO-SERVER

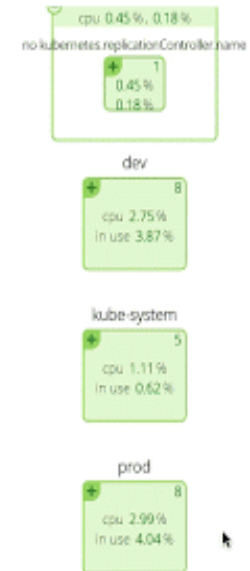
 **Image:** minishift-demo/hello-server

 **Ports:** 8080/TCP



Map

5 minutes       



Kubernetes Pods- The Basic Unit of Deployment

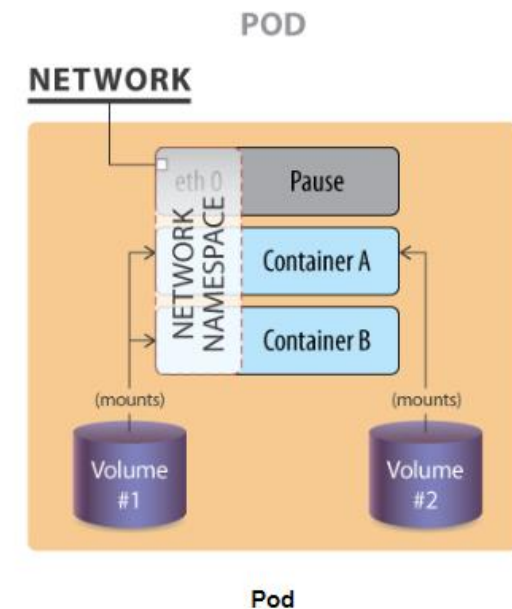
The whole point of Kubernetes is to orchestrate the lifecycle of a container. We do not interact with particular containers. Instead, the smallest unit we can work with is a *Pod*. Some would say a pod of whales or peas-in-a-pod. Due to shared resources, the design of a *Pod* typically follows a one-process-per-container 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.

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.



How Pod <-> Communication works

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:

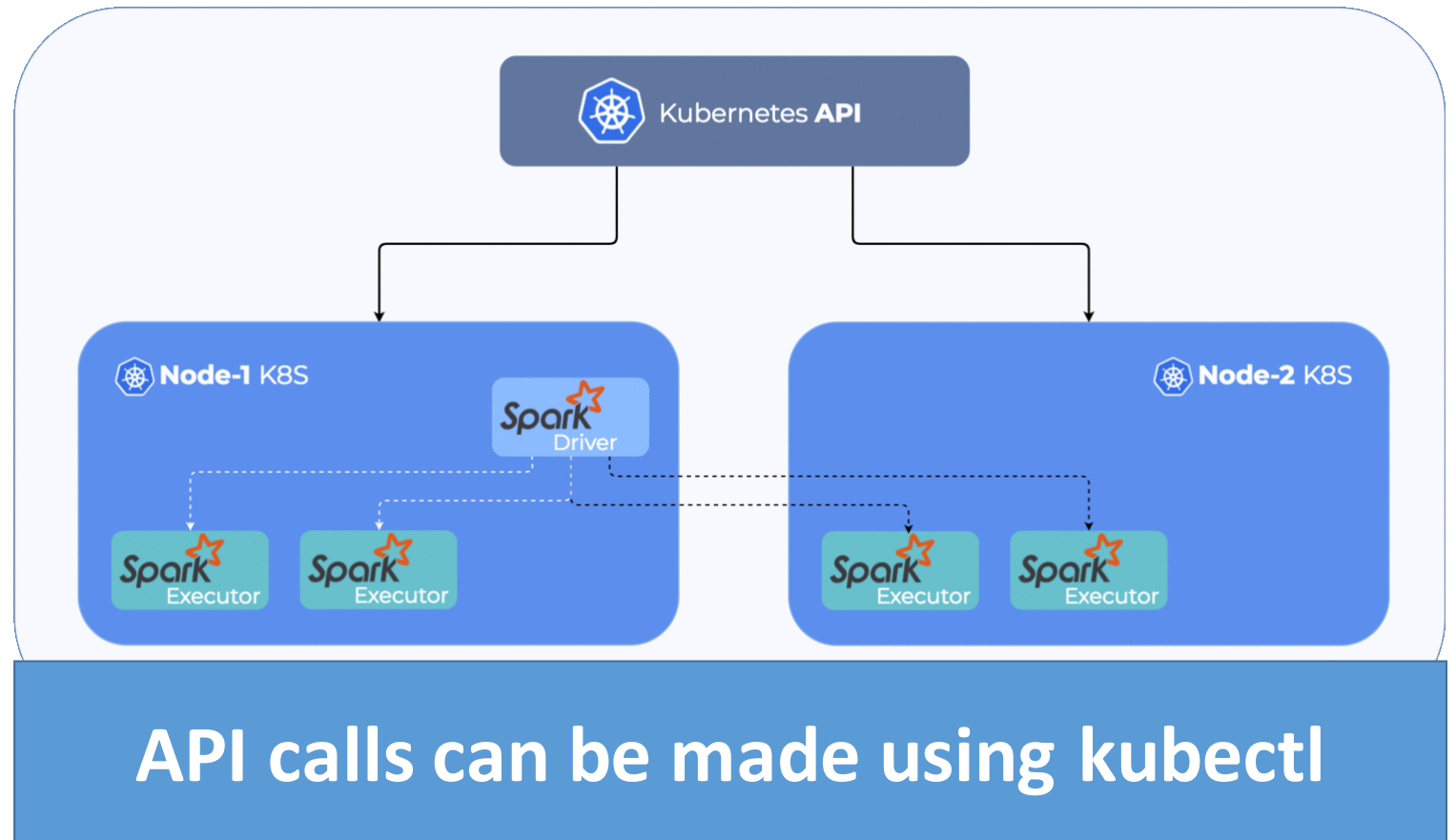
- [Weave](#)
- [Flannel](#)
- [Calico](#)
- [Romana](#).

Kubernetes API

Kubernetes has a powerful **REST**-based API. The entire architecture is API-driven. Knowing where to find resource endpoints and understanding how the API changes between versions can be important to ongoing administrative tasks, as there is much ongoing change and growth. Currently, there is no process for deprecation.

Such an API group is seen here:

```
$ curl https://127.0.0.1:6443/apis -k
....
{
  "name": "apps",
  "versions": [
    {
      "groupVersion": "apps/v1beta1",
      "version": "v1beta1"
    },
    {
      "groupVersion": "apps/v1beta2",
      "version": "v1beta2"
    }
  ],
  ....
}
```



Kubernetes API can be leveraged using kubectl

Kubernetes exposes resources via RESTful API calls, which allows all resources to be managed via HTTP, JSON or even XML, the typical protocol being HTTP. The state of the resources can be changed using standard HTTP verbs (e.g. GET, POST, PATCH, DELETE, etc.).

kubectl has a verbose mode argument which shows details from where the command gets and updates information. Other output includes **curl** commands you could use to obtain the same result. While the verbosity accepts levels from zero to any number, there is currently no verbosity value greater than nine. You can check this out for `kubectl get`. The output below has been formatted for clarity:

```
$ kubectl --v=10 get pods firstpod
....
I1215 17:46:47.860958 29909 round_tripper.go:417]
    curl -k -v -XGET -H "Accept: application/json"
    -H "User-Agent: kubectl/v1.8.5 (linux/amd64) kubernetes/ccellc6"
    https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod
....
```

If you delete this pod, you will see that the HTTP method changes from XGET to XDELETE:

```
$ kubectl --v=9 delete pods firstpod
....
I1215 17:49:32.166115 30452 round_tripper.go:417]
    curl -k -v -XDELETE -H "Accept: application/json, */*"
    -H "User-Agent: kubectl/v1.8.5 (linux/amd64) kubernetes/ccellc6"
    https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod
....
```

API Resources with kubectl

all	events (ev)	podsecuritypolicies (psp)
certificatesigningrequests (csr)	horizontalpodautoscalers (hpa)	podtemplates
clusterrolebindings	ingresses (ing)	replicasets (rs)
clusterroles	jobs	replicationcontrollers (rc)
clusters (valid only for federation apiservers)	limitranges (limits)	resourcequotas (quota)
componentstatuses (cs)	namespaces (ns)	rolebindings
configmaps (cm)	networkpolicies (netpol)	roles
controllerrevisions	nodes (no)	secrets
cronjobs	persistentvolumeclaims (pvc)	serviceaccounts (sa)
customresourcedefinition (crd)	persistentvolumes (pv)	services (svc)
daemonsets (ds)	poddisruptionbudgets (pdb)	statefulsets
deployments (deploy)	podpreset	storageclasses
endpoints (ep)	pods (po)	

Kubeconfig contents

- **apiVersion**
As with other objects, this instructs the `kube-apiserver` where to assign the data.
- **clusters**
This contains the name of the cluster, as well as where to send the API calls. The `certificate-authority-data` is passed to authenticate the `curl` request.
- **contexts**
A setting which allows easy access to multiple clusters, possibly as various users, from one configuration file. It can be used to set `namespace`, `user`, and `cluster`.
- **current-context**
Shows which cluster and user the `kubectl` command would use. These settings can also be passed on a per-command basis.
- **kind**
Every object within Kubernetes must have this setting, in this case a declaration of object type `config`.
- **preferences**
Currently not used, optional settings for the `kubectl` command, such as colorizing output.
- **users**
A nickname associated with client credentials, which can be client key and certificate, username and password, and a token. Token and username/password are mutually exclusive. These can be configured via the `kubectl config set-credentials` command.

To Access Cluster from outside we can use kubeconfig file which can be saved in your home directory. Curl can also be used.

NameSpace

Every API call includes a namespace, using `default` if not otherwise declared:

`https://10.128.0.3:6443/api/v1/namespaces/default/pods`.

Namespaces, a Linux kernel feature that segregates system resources, are intended to isolate multiple groups and the resources they have access to work with via quotas. Eventually, access control policies will work on namespace boundaries, as well. One could use *Labels* to group resources for administrative reasons.

There are three namespaces when a cluster is first created.

- `default`
This is where all resources are assumed, unless set otherwise.
- `kube-public`
A namespace readable by all, even those not authenticated. General information is often included in this namespaces.
- `kube-system`
Contains infrastructure pods.

Should you want to see all the resources on a system, you must pass the `--all-namespaces` option to the `kubectl` command.

```
$ kubectl get ns
$ kubectl create ns linuxcon
$ kubectl describe ns linuxcon
$ kubectl get ns/linuxcon -o yaml
$ kubectl delete ns/linuxcon
```

Namespace Commands

Kubernetes Objects

- **Node**
Represents a machine - physical or virtual - that is part of your **Kubernetes** cluster. You can get more information about nodes with the `kubectl get nodes` command. You can turn on and off the scheduling to a node with the `kubectl cordon/uncordon` commands.
- **Service Account**
Provides an identifier for processes running in a pod to access the API server and performs actions that it is authorized to do.
- **Resource Quota**
It is an extremely useful tool, allowing you to define quotas per namespace. For example, if you want to limit a specific namespace to only run a given number of pods, you can write a `resourcequota` manifest, create it with `kubectl` and the quota will be enforced.
- **Endpoint**
Generally, you do not manage endpoints. They represent the set of IPs for Pods that match a particular service. They are handy when you want to check that a service actually matches some running pods. If an endpoint is empty, then it means that there are no matching pods and something is most likely wrong with your service definition.
- **Deployment**
A controller which manages the state of `ReplicaSets` and the pods within. The higher level control allows for more flexibility with upgrades and administration. Unless you have a good reason, use a deployment.
- **ReplicaSet**
Orchestrates individual Pod lifecycle and updates. These are newer versions of Replication Controllers, which differ only in selector support.
- **Pod**
As we've mentioned, it is the lowest unit we can manage, runs the application container, and possibly support containers.

DaemonSets

Should you want to have a logging application on every node, a `DaemonSet` may be a good choice. The controller ensures that a single pod, of the same type, runs on every node in the cluster. When a new node is added to the cluster, a Pod, same as deployed on the other nodes, is started. When the node is removed, the `DaemonSet` makes sure the local Pod is deleted.

As usual, you get all the CRUD operations via `kubectl`:

```
$ kubectl get daemonsets
```

```
$ kubectl get ds
```

RBAC – Role Based Access Control

```
$ curl localhost:8080/apis/rbac.authorization.k8s.io/v1beta1
```

```
...
  "groupVersion": "rbac.authorization.k8s.io/v1beta1",
  "resources": [
...
    "kind": "ClusterRoleBinding"
...
    "kind": "ClusterRole"
...
    "kind": "RoleBinding"
...
    "kind": "Role"
...

```

These resources allow us to define Roles within a cluster and associate users to these Roles. For example, we can define a Role for someone who can only read pods in a specific namespace, or a Role that can create deployments, but no services. We will talk more about RBAC later in the course.

Replication Controllers

ReplicationControllers (RC) ensure that a specified number of pod replicas is running at any one time. *ReplicationControllers* also give you the ability to perform rolling updates. However, those updates are managed on the client side. This is problematic if the client loses connectivity, and can leave the cluster in an unplanned state. To avoid problems when scaling the RCs on the client side, a new resource has been introduced in the `extensions/v1beta1` API group: *Deployments*.

Deployments allow server-side updates to pods at a specified rate. They are used for canary and other deployment patterns. Deployments generate *ReplicaSets*, which offer more selection features than *ReplicationControllers*, such as `matchExpressions`.

```
$ kubectl run dev-web --image=nginx:1.13.7-alpine  
deployment "dev-web" created
```

MetaData Objects

- **annotations:**
These values do not configure the object, but provide further information that could be helpful to third-party applications or administrative tracking. Unlike labels, they cannot be used to select an object with **kubectl**.
- **creationTimestamp :**
Shows when the object was originally created. Does not update if the object is edited.
- **generation :**
How many times this object has been edited, such as changing the number of replicas, for example.
- **labels :**
Arbitrary strings used to select or exclude objects for use with **kubectl**, or other API calls. Helpful for administrators to select objects outside of typical object boundaries.
- **name :**
This is a **required** string, which we passed from the command line. The name must be unique to the namespace.
- **resourceVersion :**
A value tied to the **etcd** database to help with concurrency of objects. Any changes to the database will cause this number to change.
- **selfLink :**
References how the **kube-apiserver** will ingest this information into the API.
- **uid :**
Remains a unique ID for the life of the object.

Deployment Configuration Spec

There are two `spec` declarations for the deployment. The first will modify the *ReplicaSet* created, while the second will pass along the Pod configuration.

```
spec:
  replicas: 1
  selector:
    matchLabels:
      run: dev-web
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
```

- `spec` :
A declaration that the following items will configure the object being created.
- `replicas` :
As the object being created is a *ReplicaSet*, this parameter determines how many Pods should be created. If you were to use `kubectl edit` and change this value to two, a second Pod would be generated.
- `selector` :
A collection of values ANDed together. All must be satisfied for the replica to match. Do not create Pods which match these selectors, as the deployment controller may try to control the resource, leading to issues.
- `matchLabels` :
Set-based requirements of the Pod selector. Often found with the `matchExpressions` statement, to further designate where the resource should be scheduled.

Scaling & Rolling Update

The API server allows for the configurations settings to be updated for most values. There are some immutable values, which may be different depending on the version of Kubernetes you have deployed.

A common update is to change the number of replicas running. If this number is set to zero, there would be no containers, but there would still be a *ReplicaSet* and *Deployment*. This is the backend process when a *Deployment* is deleted.

```
$ kubectl scale deploy/dev-web --replicas=4
deployment "dev-web" scaled

$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
dev-web       4         4         4            1           12m
```

Non-immutable values can be edited via a text editor, as well. Use `edit` to trigger an update. For example, to change the deployed version of the `nginx` web server to an older version:

```
$ kubectl edit deployment nginx
....
  containers:
  - image: nginx:1.8 #<---Set to an older version
    imagePullPolicy: IfNotPresent
    name: dev-web
....
```

This would trigger a rolling update of the deployment. While the deployment would show an older age, a review of the Pods would show a recent update and older version of the web server application deployed.

Labels

Part of the metadata of an object is a *label*. Though labels are not API objects, they are an important tool for cluster administration. They can be used to select an object based on an arbitrary string, regardless of the object type. Labels are immutable as of API version `apps/v1`.

Every resource can contain labels in its metadata. By default, creating a *Deployment* with `kubectl run` adds a label, as we saw in:

```
....
  labels:
    pod-template-hash: "3378155678"
    run: ghost
....
```

You could then view labels in new columns:

```
$ kubectl get pods -l run=ghost
```

NAME	READY	STATUS	RESTARTS	AGE
ghost-3378155678-eq5i6	1/1	Running	0	10m

```
$ kubectl get pods -Lrun
```

NAME	READY	STATUS	RESTARTS	AGE	RUN
ghost-3378155678-eq5i6	1/1	Running	0	10m	ghost
nginx-3771699605-4v27e	1/1	Running	1	1h	nginx

Services

Services can be of the following types:

- **ClusterIP**
- **NodePort**
- **LoadBalancer**
- **ExternalName**.

The **ClusterIP** service type is the default, and only provides access internally (except if manually creating an external endpoint). The range of ClusterIP used is defined via an API server startup option.

The **NodePort** type is great for debugging, or when a static IP address is necessary, such as opening a particular address through a firewall. The NodePort range is defined in the cluster configuration.

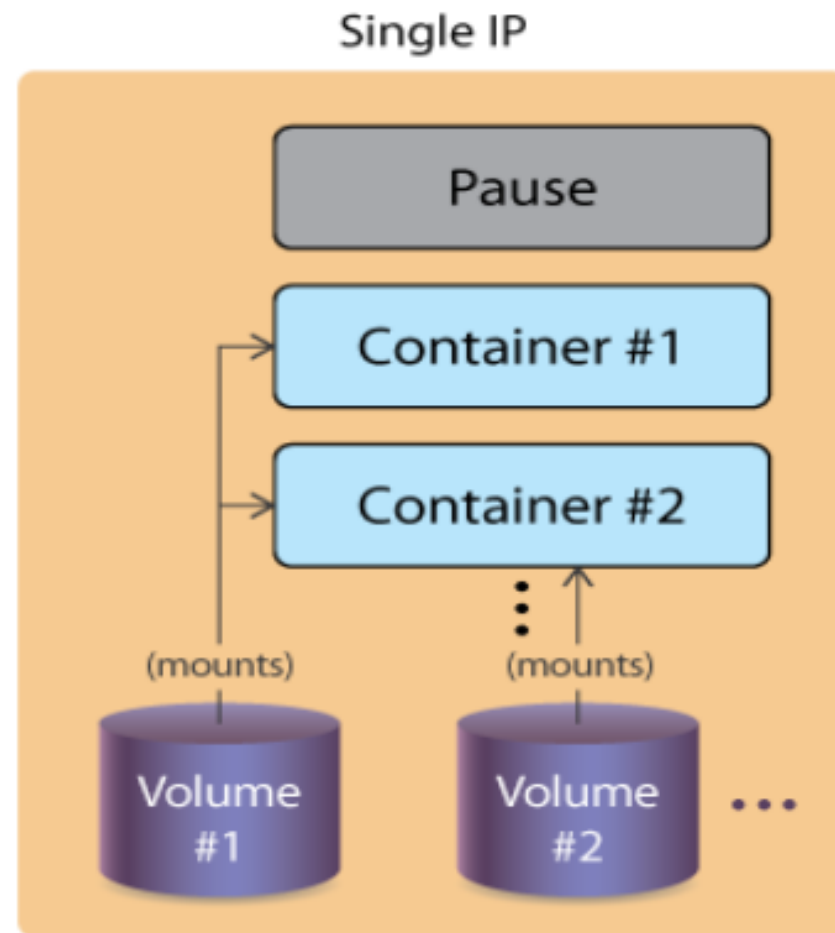
The **LoadBalancer** service was created to pass requests to a cloud provider like GKE or AWS. Private cloud solutions also may implement this service type if there is a cloud provider plugin, such as with **CloudStack** and **OpenStack**. Even without a cloud provider, the address is made available to public traffic, and packets are spread among the Pods in the deployment automatically.

A newer service is **ExternalName**, which is a bit different. It has no selectors, nor does it define ports or endpoints. It allows the return of an alias to an external service. The redirection happens at the DNS level, not via a proxy or forward. This object can be useful for services not yet brought into the Kubernetes cluster. A simple change of the type in the future would redirect traffic to the internal objects.

The `kubectl proxy` command creates a local service to access a ClusterIP. This can be useful for troubleshooting or development work.

Persistent Volumes

A Pod specification can declare one or more volumes and where they are made available. Each requires a name, a type, and a mount point. The same volume can be made available to multiple containers within a Pod, which can be a method of container-to-container communication. A volume can be made available to multiple Pods, with each given an `access mode` to write. There is no concurrency checking, which means data corruption is probable, unless outside locking takes place.



K8s Pod Volumes

There are several types that you can use to define volumes, each with their pros and cons. Some are local, and many make use of network-based resources.

In **GCE** or **AWS**, you can use volumes of type `GCEPersistentDisk` or `awsElasticBlockStore`, which allows you to mount **GCE** and **EBS** disks in your Pods, assuming you have already set up accounts and privileges.

NFS (Network File System) and **iSCSI** (Internet Small Computer System Interface) are straightforward choices for multiple readers scenarios.

rbd for block storage or **CephFS** and **GlusterFS**, if available in your **Kubernetes** cluster, can be a good choice for multiple writer needs.

Besides the volume types we just mentioned, there are many other possible, with more being added: `azureDisk`, `azureFile`, `csi`, `downwardAPI`, `fc` (fibre channel), `flocker`, `gitRepo`, `local`, `projected`, `portworxVolume`, `quobyte`, `scaleIO`, `secret`, `storageos`, `vsphereVolume`, `persistentVolumeClaim`, etc.

The following YAML file creates a pod with two containers, both with access to a shared volume:

```
containers:
  - image: busybox
  volumeMounts:
    - mountPath: /busy
name: test
name: busy
  - image: busybox
  volumeMounts:
    - mountPath: /box
name: test
name: box
volumes:
  - name: test
    emptyDir: {}
```


Secrets

Pods can access local data using volumes, but there is some data you don't want readable to the naked eye. Passwords may be an example. Someone reading through a YAML file may read a password and remember it. Using the *Secret* API resource, the same password could be encoded. A casual reading would not give away the password. You can create, get, or delete secrets:

```
$ kubectl get secrets
```

Secrets can be manually encoded with `kubectl create secret`:

```
$ kubectl create secret generic --help
```

```
$ kubectl create secret generic mysql --from-literal=password=root
```

A secret is not encrypted, only **base64**-encoded. You can see the encoded string inside the secret with **kubectl**. The secret will be decoded and be presented as a string saved to a file. The file can be used as an environmental variable or in a new directory, similar to the presentation of a volume.

A secret can be made manually as well, then inserted into a YAML file:

```
$ echo LFTr@1n | base64
```

```
TEZUckAxbgo=
```

```
$ vim secret.yaml
```

```
apiVersion: v1
```

```
kind: Secret
```

```
metadata:
```

```
  name: LF-secret
```

```
data:
```

```
  password: TEZUckAxbgo=
```

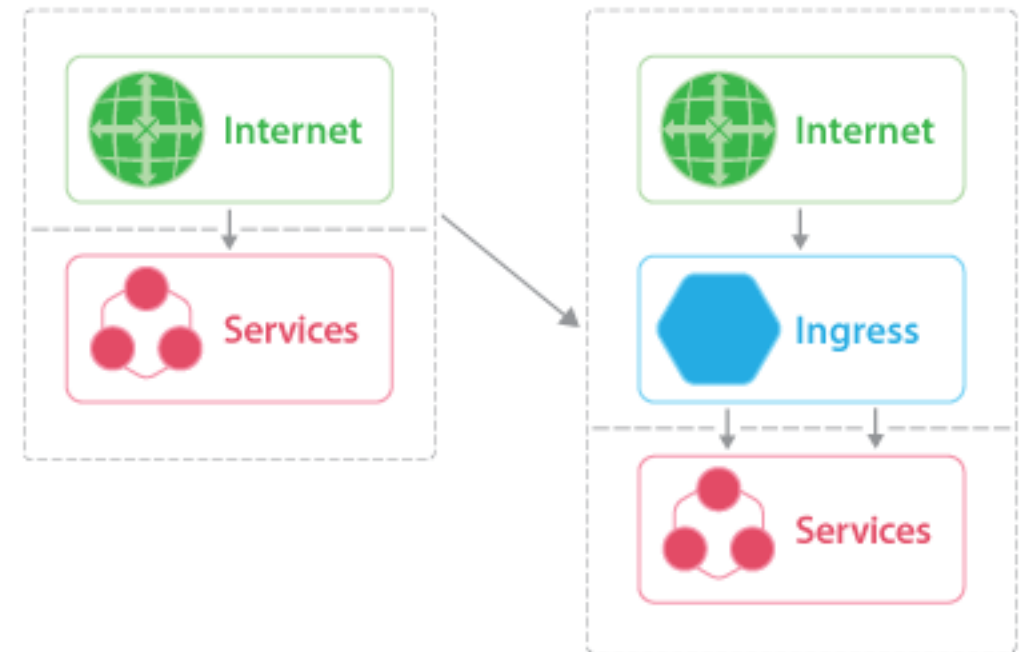
Ingress Controllers

An *Ingress Controller* is a daemon running in a Pod which watches the `/ingresses` endpoint on the API server, which is found under the `extensions/v1beta1` group for new objects. When a new endpoint is created, the daemon uses the configured set of rules to allow inbound connection to a service, most often HTTP traffic. This allows easy access to a service through an edge router to Pods, regardless of where the Pod is deployed.

Multiple *Ingress Controllers* can be deployed. Traffic should use annotations to select the proper controller. The lack of a matching annotation will cause every controller to attempt to satisfy the ingress traffic.

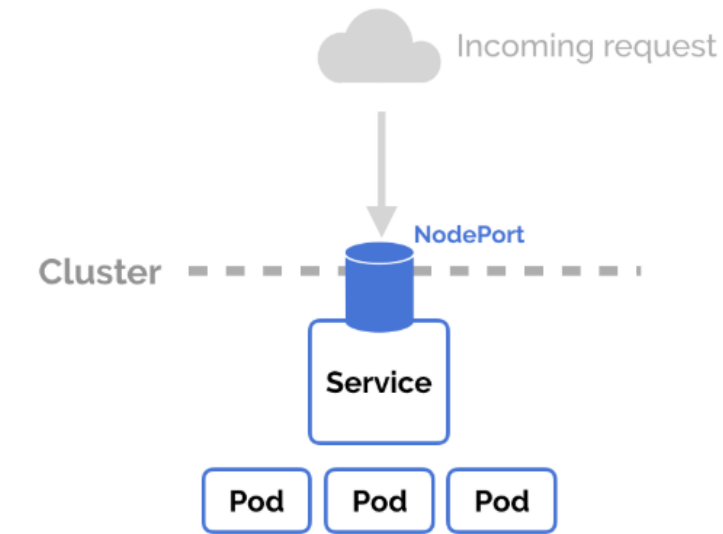
The Ingress

is a collection of rules that allow inbound connections to reach the cluster services.

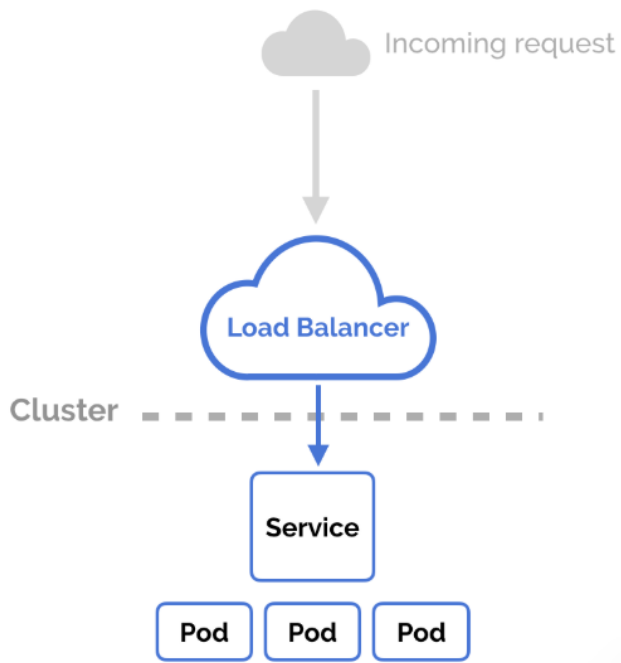


Ingress Controller for inbound connections

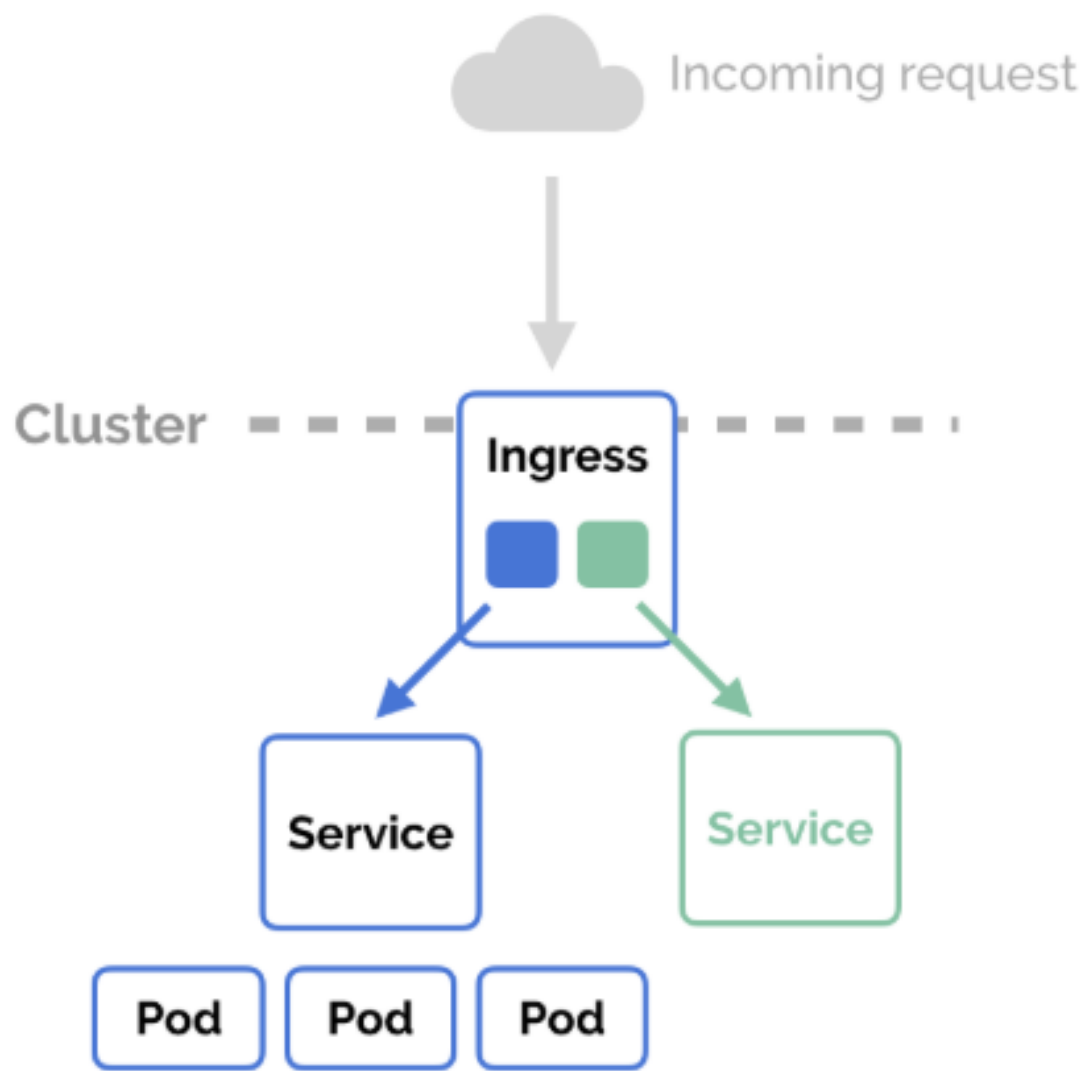
NodePort



LoadBalancer



Ingress



Helm- The Kubernetes Package Manager

A typical containerized application will have several manifests. Manifests for deployments, services, and ConfigMaps. You will probably also create some secrets, Ingress, and other objects. Each of these will need a manifest.

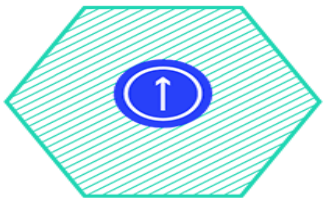
With **Helm**, you can package all those manifests and make them available as a single tarball. You can put the tarball in a repository, search that repository, discover an application, and then, with a single command, deploy and start the entire application.

The server runs in your Kubernetes cluster, and your client is local, even a local laptop. With your client, you can connect to multiple repositories of applications.

You will also be able to upgrade or roll back an application easily from the command line.



\$ helm upgrade



Upgrade a
release to a new
version of a chart

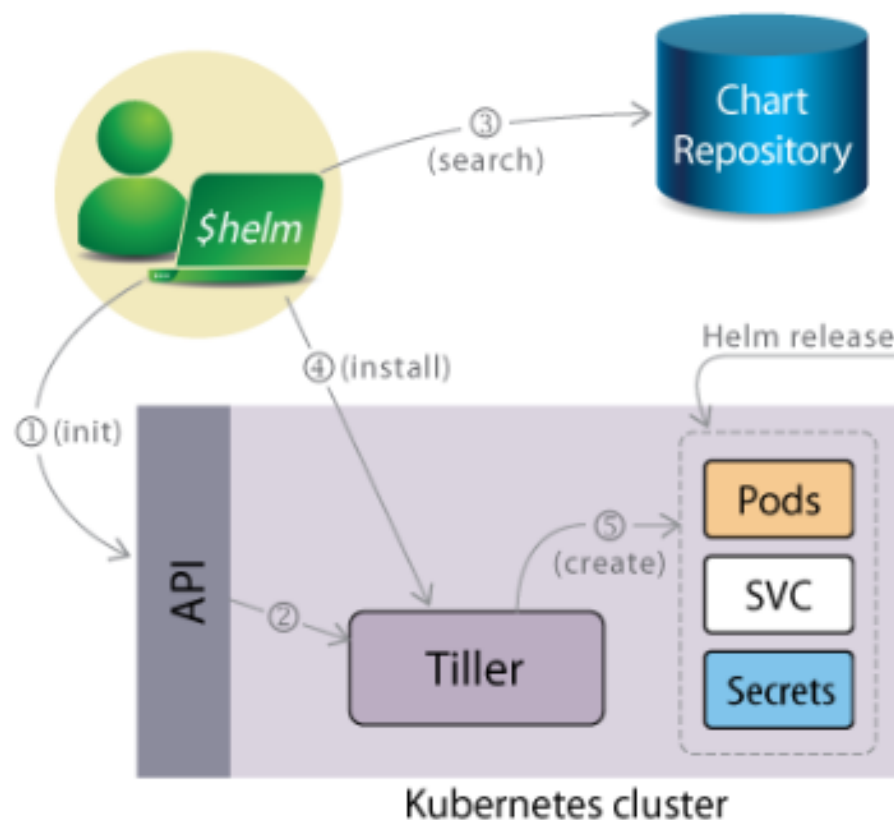
HELM

The **helm** tool packages a Kubernetes application using a series of YAML files into a chart, or package. This allows for simple sharing between users, tuning using a templating scheme, as well as provenance tracking, among other things.

Helm is made of two components:

- A server called *Tiller*, which runs inside your **Kubernetes** cluster.
- A client called *Helm*, which runs on your local machine.

With the *Helm* client you can browse package repositories (containing published *Charts*), and deploy those *Charts* on your **Kubernetes** cluster. The *Helm* will download the chart and pass a request to Tiller to create a release, otherwise known as an instance of a **chart**. The release will be made of various resources running in the Kubernetes cluster.

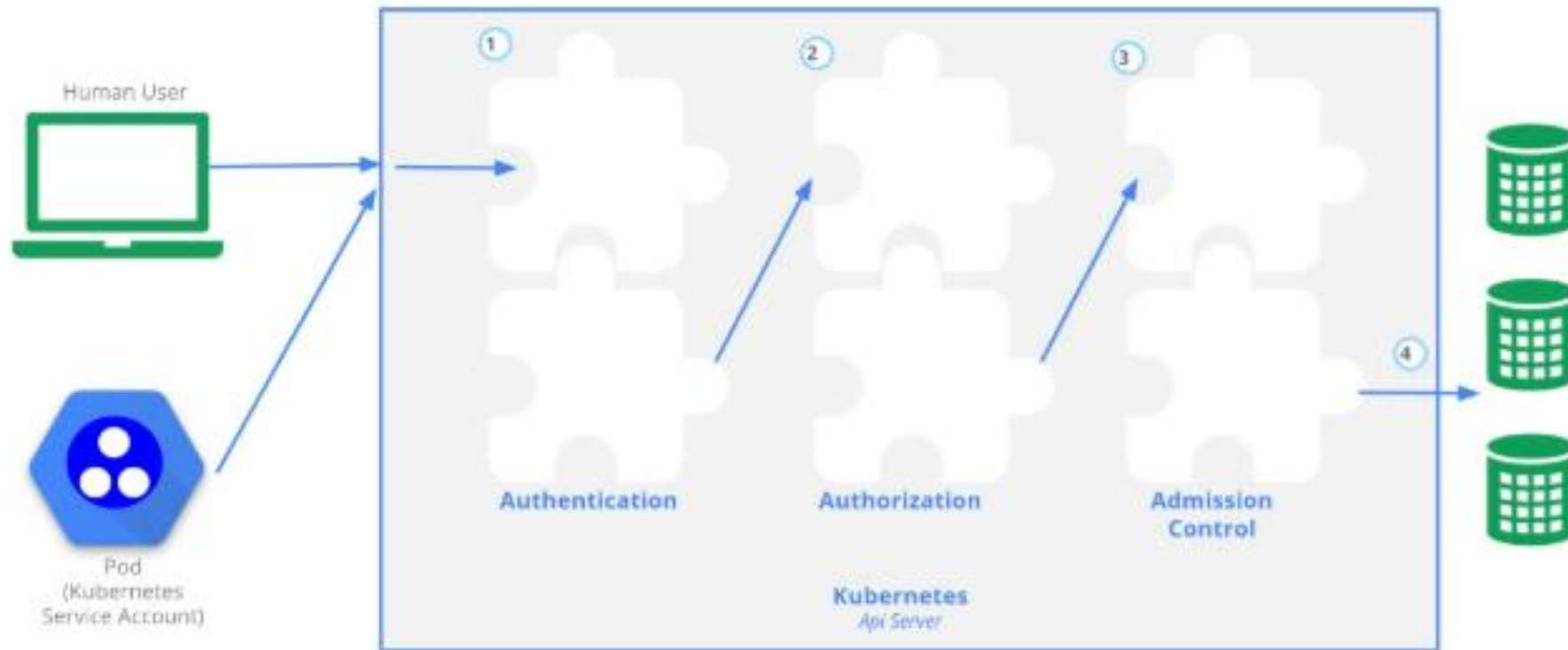


Basic Helm and Tiller Flow

Security within Kubernetes Cluster

To perform any action in a **Kubernetes** cluster, you need to access the API and go through three main steps:

- Authentication
- Authorization (ABAC or RBAC)
- Admission Control.



Authorization

Once a request is authenticated, it needs to be authorized to be able to proceed through the **Kubernetes** system and perform its intended action.

There are three main authorization modes and two global Deny/Allow settings. The three main modes are:

- ABAC
- RBAC
- **WebHook**.

They can be configured as **kube-apiserver** startup options:

```
--authorization-mode=ABAC
--authorization-mode=RBAC
--authorization-mode=Webhook
--authorization-mode=AlwaysDeny
--authorization-mode=AlwaysAllow
```

The authorization modes implement policies to allow requests. Attributes of the requests are checked against the policies (e.g. user, group, namespace, verb).


ABAC – Attribute Based Access Control. This is default authorization mode.

Policies are defined in a JSON file and referenced to by a **kube-apiserver** startup option:

```
--authorization-policy-file=my_policy.json
```

RBAC – Role Based Access Control.

All resources are modeled API objects in Kubernetes, from Pods to Namespaces. They also belong to **API Groups**, such as **core** and **apps**. These resources allow operations such as Create, Read, Update, and Delete (CRUD), which we have been working with so far. Operations are called **verbs** inside YAML files. Adding to these basic components, we will add more elements of the API, which can then be managed via RBAC.



Kubernetes is a **disruptive pirate ship** —
built by Google, with sails set straight for Amazon

THANK
YOU