Welcome to Kubernetes

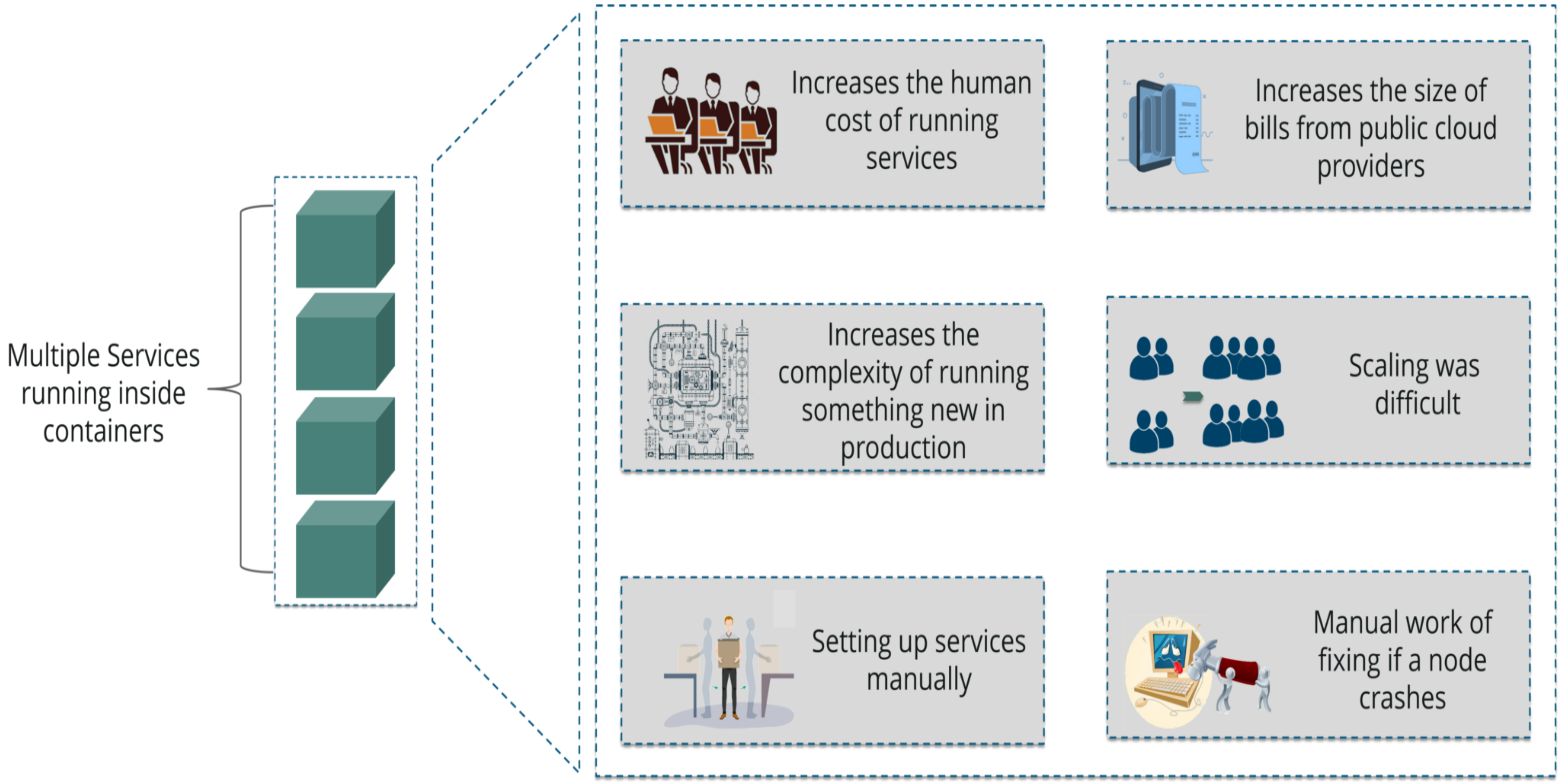
Introduction

- Handling a large number of containers all together was also a problem. Sometimes while running containers, on the product side, few issues were raised, which were not present at the development stage. This kind of scenarios introduced the Container Orchestration System.
- Kubernetes is a platform that eliminates the manual processes involved in deploying containerized applications.
- Kubernetes is an open-source system that handles the work of scheduling containers onto a compute cluster and manages the workloads to ensure they run as the user intends.
- Being the Google's brainchild, it offers excellent community and works brilliantly with all the cloud providers to become a *multi-container management solution.*

Container Orchestration

- Container Orchestration Tool
  - Monitoring
  - Deployment
  - Scaling
  - Optimally runs your workloads
  - Provides access to needed storage and networking resources
  - Ensures applications run with the business intent
    - Efficiency
    - High availability
    - Load balanced
    - Resource Management
    - Desired state configuration
    - Uses a “Declarative model”

Challenges without Container Orchestration


Multiple Services running inside containers
Increases the human cost of running services
Increases the size of bills from public cloud providers
Increases the complexity of running something new in production
Scaling was difficult
Setting up services manually
Manual work of fixing if a node crashes

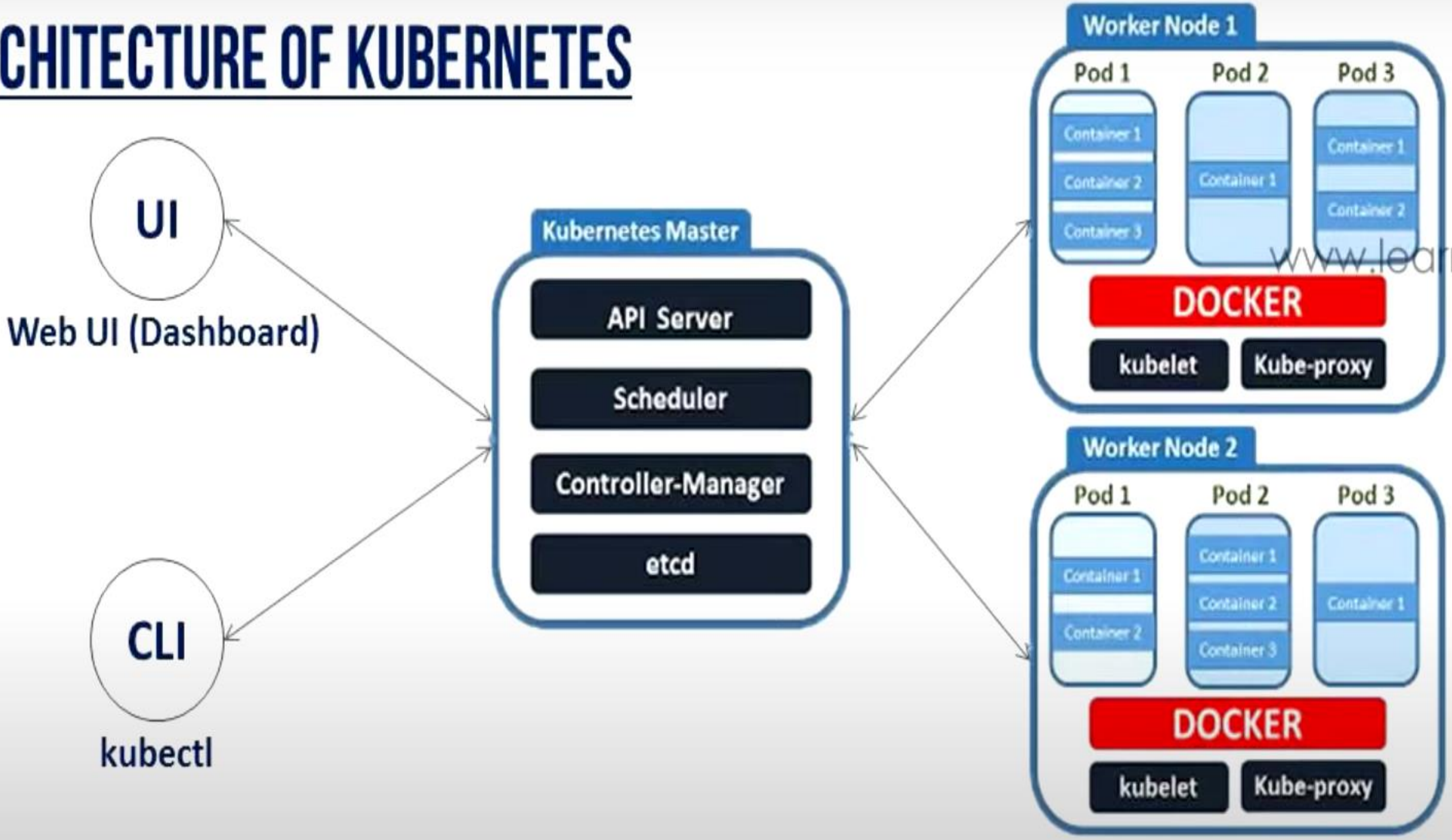
History

- 2003 – Project BORG (Google's large-scale internal cluster management system)
- 2013 – Project OMEGA
- 2014 – Google introduced Kubernetes (Kubernetes is introduced as an open source container orchestration system, based on Omega and Borg)
- 2015 – Version 1.0 and Google created Cloud Native Computer Foundation (CNCF)
- 2016 – Version 1.2-1.5 (Minikube was introduced)
- 2017 – Version 1.6-1.9 Enterprise Adoption (Amazon & Microsoft announced their own managed Kubernetes servers), Docker announced support for Kubernetes
- 2018 – Google Kubernetes Engine was Released (Version 1.10 – 1.12, EKS, AKS and Containerd is publicly available

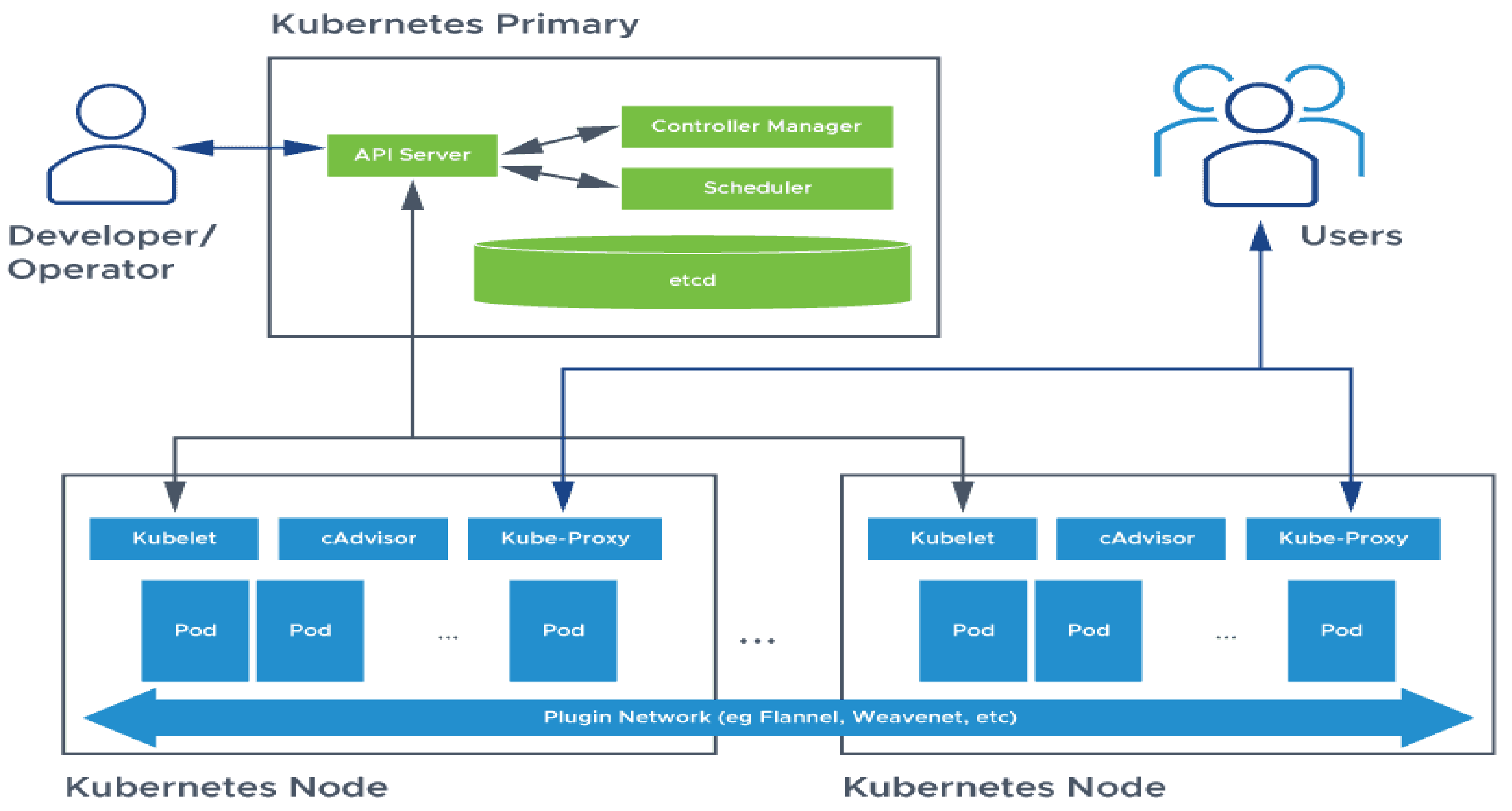
History

- 2019 – Version 1.13-1.15, kubeadm in GA,CSI, Windows nodes & more
- *Written in Go/GoLang*

ARCHITECTURE OF KUBERNETES



Architecture



Kubernetes Clusters

- **Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.**
- The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines.
- **Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.**
- A Kubernetes cluster consists of two types of resources:
  - The **Control Plane** coordinates the cluster
  - **Nodes** are the workers that run applications

Kubernetes Master

- Kubernetes Master is a main node responsible for managing the entire Kubernetes clusters , It handles the orchestration of the worker nodes. It has four main components that take care of Communication, Scheduling and controllers.
  - **API Server:** The API server is the entry point for all the REST commands used to control the cluster.
    - Consumes Manifest files via JSON
  - **Scheduler:** Scheduler watches the pods and assigns the pods to run on specific hosts.
  - **Kube-Controller-Manager:** Controller Manager runs the controller in background which runs different tasks in Kubernetes Cluster.

Controllers

- **Node Controller:** Its responsible for noticing and responding when nodes go down
- **Replication Controller:** It maintains the number of pods. It controls how many identical copies of a pod should be running somewhere on the cluster.
- **Endpoint Controller:** joins services and pods together.
- **Deployment Controller:** Provides declarative updates for pods and Replicasets
- **Jobs Controller:** is the supervisor process for pods carrying out batch jobs

- **Etcd:** is a simple distribute key value store. Kubernetes uses etcd as its database to store all cluster data. Some of the data stored in etcd is job scheduling information, pods, state information etc.

Kubernetes Node

- Worker nodes are the nodes where the applications actually running in Kubernetes cluster, it is also known as minions. These each worker nodes are controlled by the master node using kubelet process.
- Container platform must be running on each worker nodes and it works together with kubelet to run the containers, this is why we use Docker Engine and takes care of managing images and containers. We can also use other container platforms like CoreOS, Rocket etc.

- Requirements of Worker Nodes:
  - Kubelet must be running
  - Docker Container Platform
  - Kube-proxy must be running
  - supervisord

Worker/Slave Node Components

Kubelet

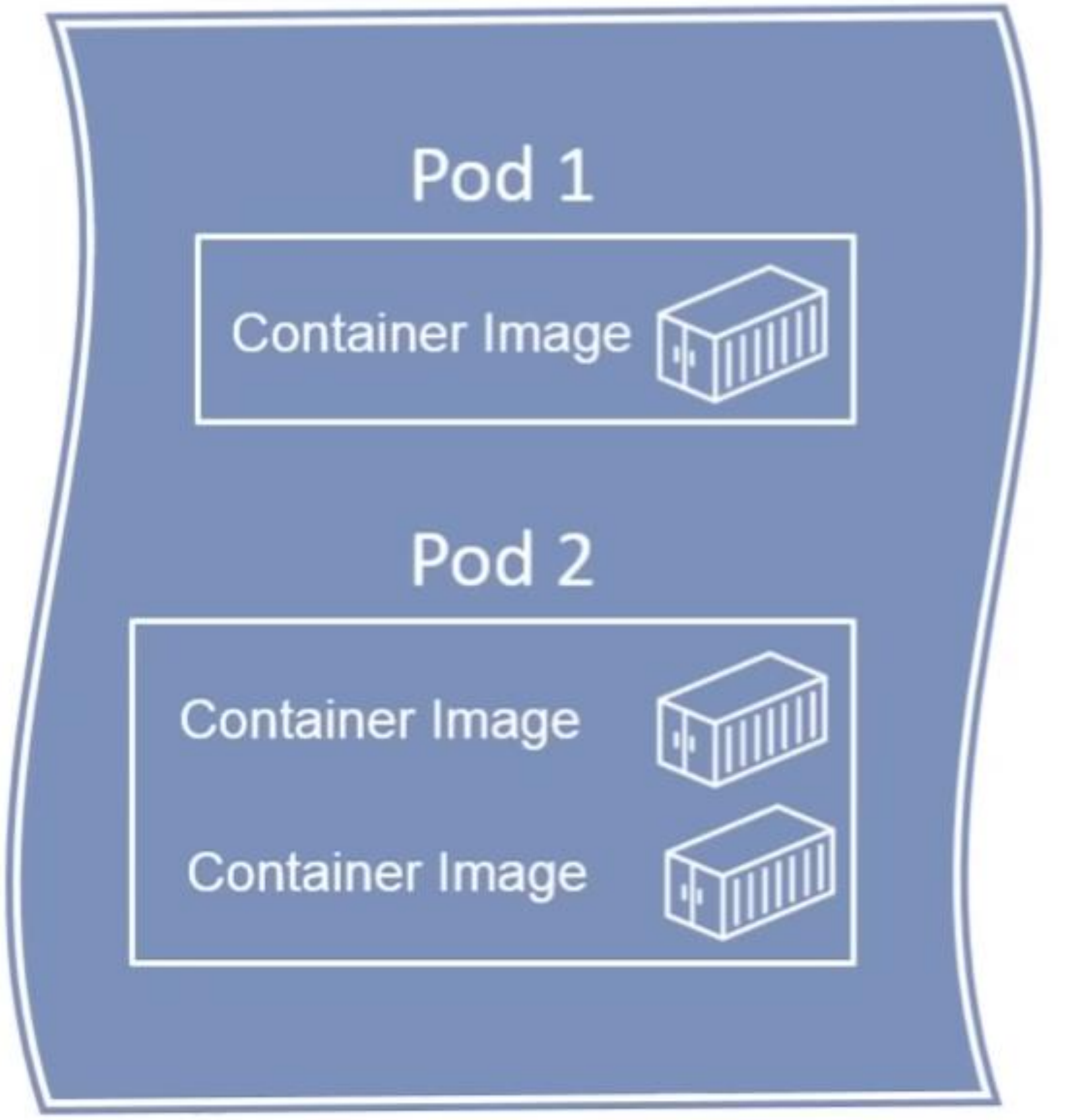
- Kubelet is the Primary node agent runs on each nodes and reads the container, manifests which ensures that containers are running and healthy.
- Kubelet gets the configuration of a Pod from the API server and ensures that the described containers are up and running.

Kube-Proxy

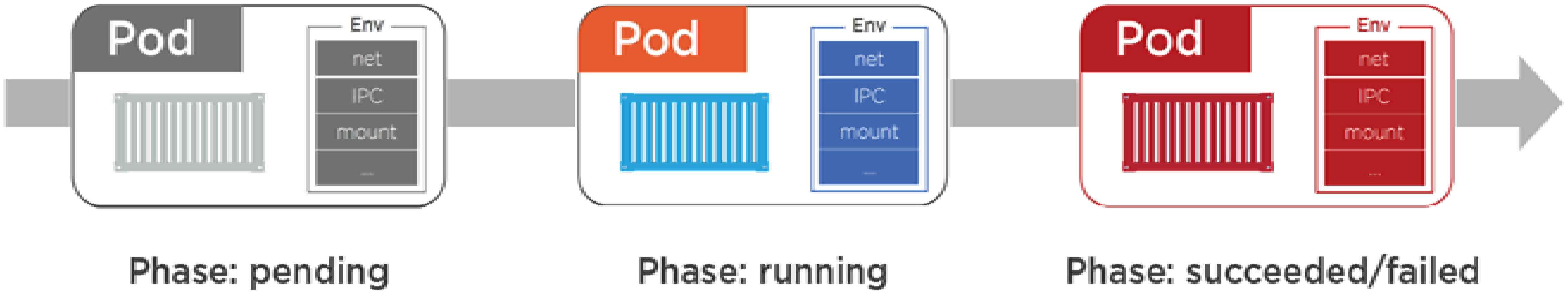
- Kube-proxy is a process helps us to have network proxy and load balancer for the services in a single worker node. It performs network routing for tcp and udp packets, and performs connection folding. Worker nodes can be exposed to internet via kube-proxy.

Pods

- Pod is the basic unit of Kubernetes. A pod can consist of one or more containes.
- Containers always run inside of pods
- Pods are instances of a container in a deployment
- Each pod gets its own IP address
- With Horizontal Pod Auto-Scaling, Pods of a deployment can be automatically started and halted based on CPU usage.
- Any data saved inside the pod will disappear without a persistent storage.



Pod Lifecycle



Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Overview of Minikube

- An all-in-one install of Kubernetes
- Takes all the distributed components of Kubernetes and packages from into a single virtual machine to run locally
- Minikube will be a Kubernetes cluster running on your machine

```
sudo minikube start --vm-driver=none

sudo minikube status
```

Kubernetes Distributions

- Kubernetes
- Openshift
- Rancher
- *Tanzo*
- *EKS*
- *AKS*
- *GKE*

Kubernetes Deployments

- Deployments are upgraded and higher version of replication controller.
- They manage the deployment of replica sets which is also an upgraded version of the replication controller.
- They have the capability to update the replica set and are also capable of rolling back to the previous version.
- To enable Pods to have Auto Healing and Auto Scaling Capabilities we need deployment
- This is an additional wrapper on top of Pod
- They provide many updated features of **matchLabels** and **selectors**. We have got a new controller in the Kubernetes master called the deployment controller which makes it happen. It has the capability to change the deployment midway.

Kubernetes Deployments

- Kubernetes deployments are the high-level construct that define an application.
- Deployments are described as a collection of resources and references.
- Deployments take many forms based on the type of services being deployed
- A deployment is a blueprint for the pods to be created.
- Handles update of its respective pods.
- A deployment will create a pod by it’s spec from the template.
- Their target is to keep the pods running and update them (With rolling update) in a more controlled way.
- Pod(s) resource usage can be specified in the deployment
- Deployment can scale up replicas of pods.
- If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. **This provides a self-healing mechanism to address machine failure or maintenance.**
- Typically described in YAML format

Kubernetes Images

- A container image represents binary data that encapsulates an application and all its software dependencies.
- Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.
- You typically create a container image of your application and push it to a registry before referring to it in a Pod
- If you don't specify a registry hostname in Container Image, Kubernetes assumes that you mean the Docker public registry.

Image Pull Policy

The imagePullPolicy for a container and the tag of the image affect when the kubelet attempts to pull (download) the specified image.
Here's a list of the values you can set for imagePullPolicy and the effects these values have

Kubernetes Images – Image Pull Policy

IfNotPresent

- This is default Pull Policy
- The image is pulled only if it is not already present locally.

Always

- Kubernetes will always pull the image from the Repository.

Never

- The kubelet does not try fetching the image. If the image is somehow already present locally, the kubelet attempts to start the container; otherwise, startup fails.

Kubernetes Labels & Selectors

- A method to keep things Organized, and to help you and Kubernetes identify resources to act up on
- Labels are key-value pairs which are attached to pods, Deployments, Nodes replication controller and services. They are used as identifying attributes for objects such as pods and replication controller. They can be added to an object at creation time and can be added or modified at the run time.
- Labels do not provide uniqueness. In general, we can say many objects can carry the same labels. They are used by the users to select a set of objects.
- Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.
- Kubernetes API currently supports two type of selectors −
  - Equality-based selectors
  - Set-based selectors

Kubernetes Labels & Selectors

- matchExpressions is a more expressive label selector in Kubernetes and supports set-based matching
- matchLabels which can only be used for exact matching. This can be used with or without the matchLabels selector.

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

Namespace

- In Kubernetes, namespaces provides a mechanism for isolating groups of resources within a single cluster.
- Names of resources need to be unique within a namespace, but not across namespaces.
- Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).
- Default namespaces are
  - Default
  - Kube-system
  - Kube-public

When to Use Multiple Namespaces

- Namespaces are intended for use in environments with many users spread across multiple teams, or projects.
- For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.
- Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.
- Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.
- It is not necessary to use multiple namespaces to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

kubectl get namespace

Namespace

- Kubernetes starts with four initial namespaces:
    - **default** The default namespace for objects with no other namespace
    - **kube-system** The namespace for objects created by the Kubernetes system
    - **kube-public** This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
    - **kube-node-lease** This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

Namespace

- Create Namespace

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

- To set the namespace for a current request, use the --namespace flag.

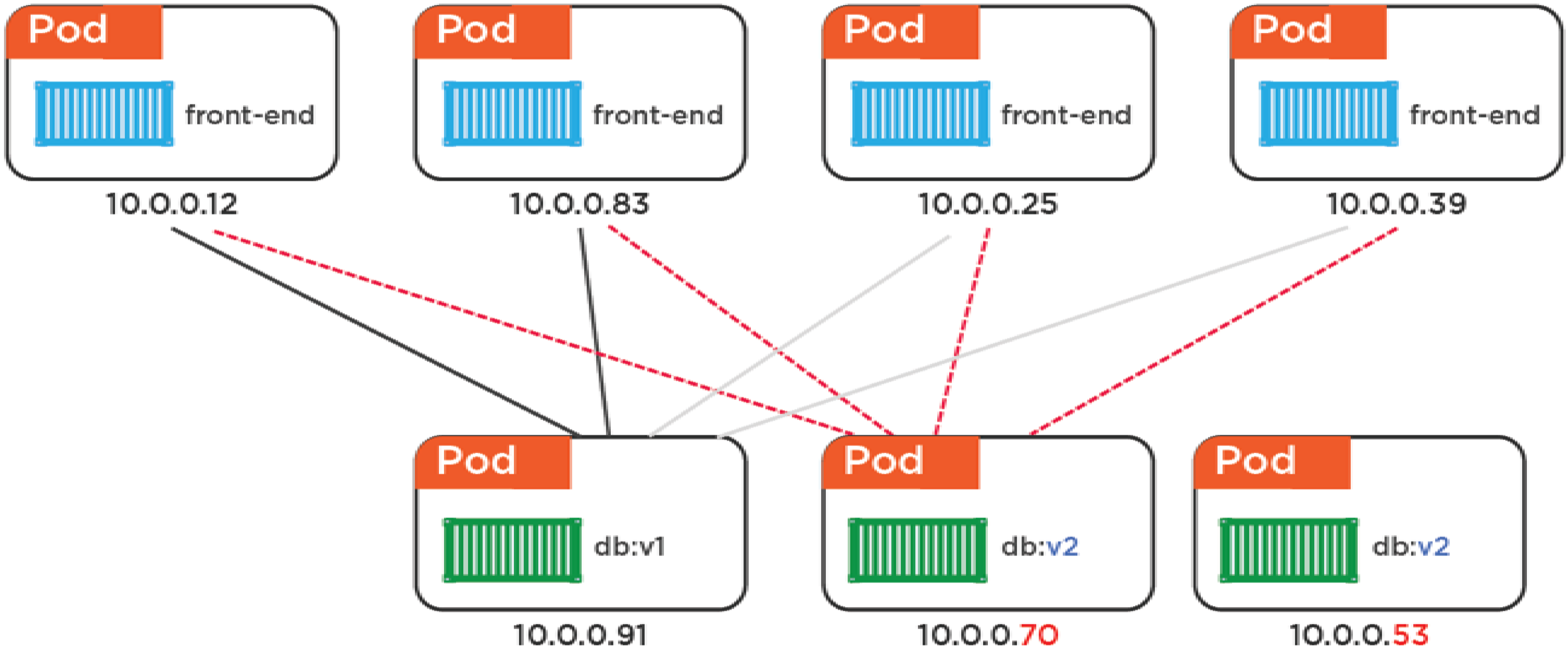
```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

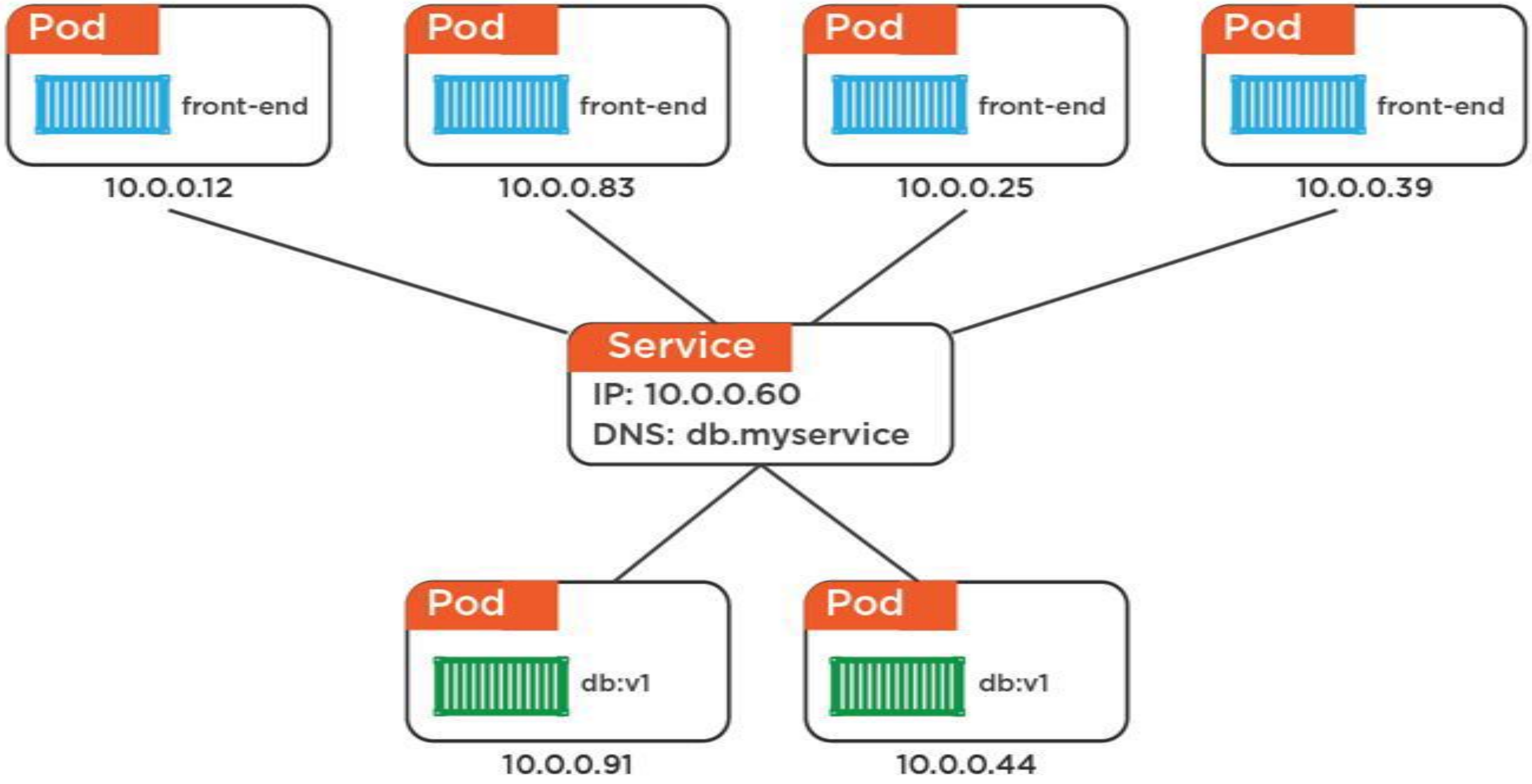
- You can permanently save the namespace for all subsequent kubectl commands in that context.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
# Validate it
kubectl config view --minify | grep namespace
```

Kubernetes Service

- A service is responsible for making our Pods discoverable inside the network or exposing them to the internet. A Service identifies pods by its LabelSelector.
- A service can be defined as a logical set of pods. It can be defined as an abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed.
- A service is a REST object in Kubernetes whose definition can be posted to Kubernetes apiServer on the Kubernetes master to create a new instance.
- Services are endpoints that export ports to the outside world
- Service Is having permanent IP address
- Types of Services available
  - ClusterIP
  - Node Port
  - Load Balancer


Pod
front-end
10.0.0.12
Pod
front-end
10.0.0.83
Pod
front-end
10.0.0.25
Pod
front-end
10.0.0.39
Pod
db:v1
10.0.0.91
Pod
db:v2
10.0.0.70
Pod
db:v2
10.0.0.53


Pod
front-end
10.0.0.12
Pod
front-end
10.0.0.83
Pod
front-end
10.0.0.25
Pod
front-end
10.0.0.39
Service
IP: 10.0.0.60
DNS: db.myservice
Pod
db:v1
10.0.0.91
Pod
db:v1
10.0.0.44

Publishing Services (Service Types)

- 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. Choosing this value makes the Service only reachable from within the cluster. 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.

Publishing Services (Service Types)

- **ExternalName**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

- Note: You need either kube-dns version 1.7 or CoreDNS version 0.0.8 or higher to use the ExternalName type.

Kubectl

- Kubectl is a Command Line Configuration (CLI) Tool for Kubernetes used to interact with master node of Kubernetes.
- Kubectl has a config file called kubeconfig, This file has the information about server and authentication information to access the API Server.

Manifest File

- A metadata file for an accompanying file or group of accompanying files
- Manifests are created in either YAML or JSON
- Manifests are converted to JSON and sent to the Kubernetes API to be processed
- Manifests are human-readable and serve as great documentation

Deployment file (Manifest file)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Replica Set (Kubernetes Controller)

- Replica Set is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle.
- It is responsible for making sure that the specified number of pod replicas are running at any point of time.
- It is used in time when one wants to make sure that the specified number of pod or at least one pod is running. It has the capability to bring up or down the specified no of pod.
- It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.
- This is the Kubernetes default Controller.

Kubernetes Deployments

Create Deployments:

- sudo kubectl create -f Deploy.yaml

Get Deployments:

- sudo kubectl get deployments
- sudo kubectl get rs

Describe Deployments:

- sudo kubectl describe deployment <deployment-name>

Update Deployments:

- kubectl set image deployment/tomcat-deployment tomcat=tomcat:9.0.1

Kubernetes Jobs

- The main function of a job is to create one or more pod and tracks about the success of pods. They ensure that the specified number of pods are completed successfully.
- When a specified number of successful run of pods is completed, then the job is considered complete.
- Scheduled job in Kubernetes uses **Cronetes**, which takes Kubernetes job and launches them in Kubernetes cluster.
    - Scheduling a job will run a pod at a specified point of time.
    - A periodic job is created for it which invokes itself automatically.
- This scheduled job concept is useful when we are trying to build and run a set of tasks at a specified point of time and then complete the process.

Kubernetes Jobs – Use Cases

- A typical use case for cron jobs is to automatically perform important tasks on a recurring basis. For example:
  - Cleaning disk space
  - Backing up files or directories
  - Generating metrics or reports
- Cron jobs run on a schedule. Using a standard notation, we can define a wide range of schedules to execute a job:
  - Every night at 10:00 PM
  - The 1st day of every month at 6:00 AM
  - Every day at 8:00 AM and 6:00 PM
  - Every Tuesday at 7:00 AM

Kubernetes Jobs

- You can use a CronJob to run Jobs on a time-based schedule. These automated jobs run like Cron tasks on a Linux or UNIX system.

- Cron jobs require a config file. Here is a manifest for a CronJob that runs a simple demonstration task every minute:

```
kubectl create -f cronjob.yaml
```

- After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

- Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

- You can stop watching the job and view the cron job again to see that it scheduled the job:

```
kubectl get cronjob hello
```

Kubernetes Jobs

- Now, find the pods that the last scheduled job created and view the standard output of one of the pods.

```
kubectl get pods
```

- Show the pod log:

```
kubectl logs <Pod-Name>
```

- When you don't need a cron job any more, delete it with kubectl delete cronjob <cronjob name>:

```
kubectl delete cronjob hello
```

- Jobs execute in UTC Time

Kubernetes Secrets

- A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.
- Such information might otherwise be put in a Pod specification or in a container image.
- Using a Secret means that you don't need to include confidential data in your application code.
- Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods.
- Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing secret data to nonvolatile storage.
- Kubernetes Secrets are, by default, stored unencrypted in the API server's underlying data store (etcd).

Working with Secrets

Creating a Secret

- There are several options to create a Secret:
    - Create Secret using kubectl command
    - Create Secret from config file
    - Create Secret using kustomize

Create Secret using Kubectl Command

- A Secret can contain user credentials required by pods to access a database.
- For example, a database connection string consists of a username and password. You can store the username in a file ./username.txt and the password in a file ./password.txt on your local machine.

```
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt
```

- The -n flag ensures that the generated files do not have an extra newline character at the end of the text. This is important because when kubectl reads a file and encodes the content into a base64 string, the extra newline character gets encoded too.

```
kubectl create secret generic db-user-pass --from-file=./username.txt --from-
file=./password.txt
```

Create Secret using Kubectl Command

- The default key name is the filename. You can optionally set the key name using --from-file=[key=]source. For example:

```
kubectl create secret generic db-user-pass \
  --from-file=username=./username.txt \
  --from-file=password=./password.txt
```

- Verify the Scret

```
kubectl get secrets
```

```
kubectl describe secrets/db-user-pass
```

Create Secret using Kubectl Command

- To view the contents of the Secret you created, run the following command:

kubectl get secret db-user-pass -o jsonpath='{.data}'

- The output is similar to:

{"password":"MWYyZDFlMmU2N2Rm","username":"YWRtaW4="}

- Now you can decode the password data:

echo 'MWYyZDFlMmU2N2Rm' | base64 --decode

- The output is similar to:

1f2d1e2e67df

- In order to avoid storing a secret encoded value in your shell history, you can run the following command:

kubectl get secret db-user-pass -o jsonpath='{.data.password}' | base64 –decode

- Delete the Secret you created:

kubectl delete secret db-user-pass

Kubernetes Volumes

- Since containers in Kubernetes clusters are ephemeral by design, there needs a way to persist data and share storage. Otherwise, files and data stored on-disk inside the containers will be lost whenever the container crashes or is restarted.
- This is where Kubernetes volumes come into play.
- In Kubernetes, a pod is a group of containers with shared storage and network resources.
- This means that containers with a shared storage will be able to communicate with each other.
- Kubernetes uses volumes as an abstraction layer to provide shared storage for containers.
- Fundamentally, volumes are directories that can be mounted to pods.
- One or more volumes can be mounted to a single pod as specified under volumes and containers.volumeMounts sections in the pod specs.

Persistent Volumes

- A **PersistentVolume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
- It is a resource in the cluster just like a node is a cluster resource.
- PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV.

- A **PersistentVolumeClaim (PVC)** is a request for storage by a user. It is similar to a Pod.
- Pods consume node resources and PVCs consume PV resources.
- Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany, see AccessModes).

Kubectl Commands

- Kubectl get nodes
- Kubectl label node <nodename> storageType=ssd
- Kubectl describe node <nodename>
- Kubectl describe deployment <DeploymentName>
- kubectl get pod
- kubectl describe pod <pod name>
- kubectl attach <pod name> -c <container name>
- Kubectl exec –it <pod name> bash
- kubectl get deployments
- kubectl describe deployments
- kubectl describe jobs/py
- Kubectl get all –a (Get all Kubernetes objects from all namespaces)
- kubectl get pods –v=7 (Verbosity level max 9)

Autoscaling

- In Kubernetes, a HorizontalPodAutoscaler automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

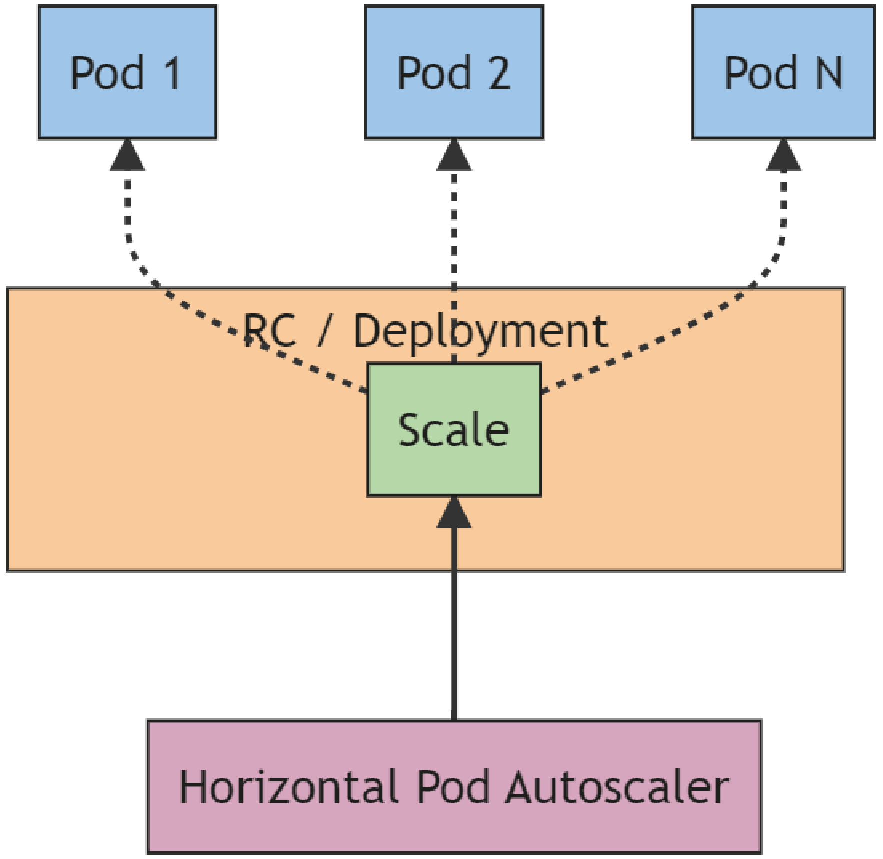
- Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from vertical scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

- If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

- The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller.

Autoscaling

- The resource determines the behavior of the horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.



Autoscaling

- Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently (it is not a continuous process). The interval is set by the --horizontal-pod-autoscaler-sync-period parameter to the kube-controller-manager (and the default interval is 15 seconds).

- Once during each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition.
- The controller manager finds the target resource defined by the scaleTargetRef, then selects the pods based on the target resource's .spec.selector labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

Autoscaling - Lab

- Deploy a simple Apache web server application with the following command.

  kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
- This Apache web server pod is given a 500 millicpu CPU limit and it is serving on port 80.

- Create a Horizontal Pod Autoscaler resource for the php-apache deployment.

kubectl autoscale deployment php-apache --cpu-percent=10 --min=1 --max=10

- This command creates an autoscaler that targets 50 percent CPU utilization for the deployment, with a minimum of one pod and a maximum of ten pods.
- When the average CPU load is lower than 50 percent, the autoscaler tries to reduce the number of pods in the deployment, to a minimum of one.
- When the load is greater than 50 percent, the autoscaler tries to increase the number of pods in the deployment, up to a maximum of ten.

Autoscaling - Lab

- Describe the autoscaler with the following command to view its details.

```
kubectl get hpa
```

- As you can see, the current CPU load is 0%, because there's no load on the server yet. The pod count is already at its lowest boundary (one), so it cannot scale in.
- Install Metrics Server

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

- Verify if Metrics Server is running

```
kubectl get po -n kube-system | grep metric
```

- Create a load for the web server by running a container.

```
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
```

Autoscaling - Lab

- To watch the deployment scale out, periodically run the following command in a separate terminal from the terminal that you ran the previous step in.

```
kubectl get hpa php-apache
```

- It may take over a minute for the replica count to increase. As long as actual CPU percentage is higher than the target percentage, then the replica count increases, up to 10. In this case, it's 250%, so the number of REPLICAS continues to increase.
- Stop the load. In the terminal window you're generating the load in, stop the load by holding down the Ctrl+C keys. You can watch the replicas scale back to 1 by running the following command again in the terminal that you're watching the scaling in.

```
kubectl get hpa
```

- The default timeframe for scaling back down is five minutes, so it will take some time before you see the replica count reach 1 again, even when the current CPU percentage is 0 percent. The timeframe is modifiable.

Autoscaling - Lab

- When you are done experimenting with your sample application, delete the php-apache resources.

```
kubectl delete deployment.apps/php-apache service/php-apache horizontalpodautoscaler.autoscaling/php-apache
```

Q & A