S3 - Part 4 - Storage classes (continue from Part 3) - 1
S3 - Part 5 - S3 Versioning
AWS CloudWatch - 17
Kubernetes - 29 (From 29-77 NodeKloud)
↳ Kubernetes Architecture (NodeKloud) - 29
Docker vs Container D - 35
etcd - 38
Kube-api - 40
Kube Controller Manager - 42
Kube Scheduler - 44
Kubelet - 45
Kube-Proxy - 46
Pod - 49
Replica Sets - 53
Deployment - 59
Services - 61
Namespace - 69
Imperative vs Declarative - 73
Kubectl - 77
Git - 81
~~kube~~
Class 1 - 81
Class 2 - 84
Class 3 - 93
Class 4 - 104
Scheduler - Part 1 - 105
Class 5 - 107
Scheduler - Part 2 - 119
Labels and Selectors - 119
Taints and Tolerations - 120
Node selectors - 124
Node Affinity - 125
Kubernetes Networking - 138
↳ CNI - 138
↳ Cluster Networking - 138 - 141

Pod Networking - 143
CNI in kubernetes - 146
CNI Weave - 148
IPAM CNI - 150
Service Networking - 151
ClusterDNS - 155
CoreDNS - 158
Ingress - 162

Application Lifecycle Management
↳ Rolling Updating and Rollbacks ~~176~~ Part - 1 — 176

01/06/24

Kubernetes Architecture (kodekloud)

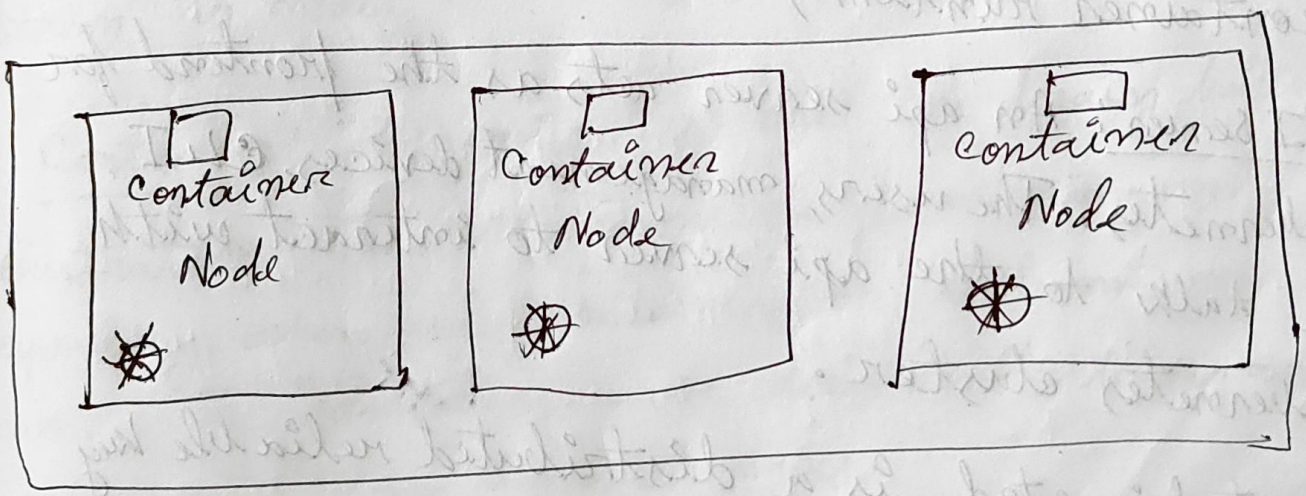
Kubernetes

container Orchestration technologies - Kubernetes, Docker swarm, MESOS.

Kubernetes Architecture

Node: A node is a machine, physical or virtual on which Kubernetes is installed. A node is a worker machine and that is where containers will be launched by Kubernetes. If the node goes down the applications will go down. So, it is needed to have more than one nodes.

Cluster: A cluster is a set of nodes grouped together.



If one node fails, the application is accessible

01/06/24

from other node(s). Moreover having multiple node helps in sharing load as well.

The Master is another node with Kubernetes installed in it and is configured as a Master. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

When Kubernetes is installed on a system, these components are installed -

02/06/24

An API server, etcd service, a Kubelet service, a container runtime, controllers and schedulers.

API Server: An api server acts as the frontend for Kubernetes. The users, management devices, CLI all talk to the api server to interact with Kubernetes cluster.

etcd: etcd is a distributed reliable key value store used by Kubernetes to store all

02/06/24

data used to manage the cluster. For example, ~~In~~ in a cluster there is multiple nodes and multiple masters, ~~in~~ ~~ect~~ etcd stores all the information on all the nodes in the cluster in a distributed manner.

etcd is responsible for implementing locks within the cluster to ensure that there are no conflicts between the masters. ~~The~~

Scheduler: The scheduler is responsible for distributing work or containers across multiple nodes. It looks for newly created containers and assigns them to nodes.

Controller: The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers make decision to bring up new containers in such cases.

02/06/24

Container Runtime: The container runtime is the underlying software that is used to run containers. For example, docker can be a container runtime. Alternatives - Rocket, ~~cayo~~ cri-o.

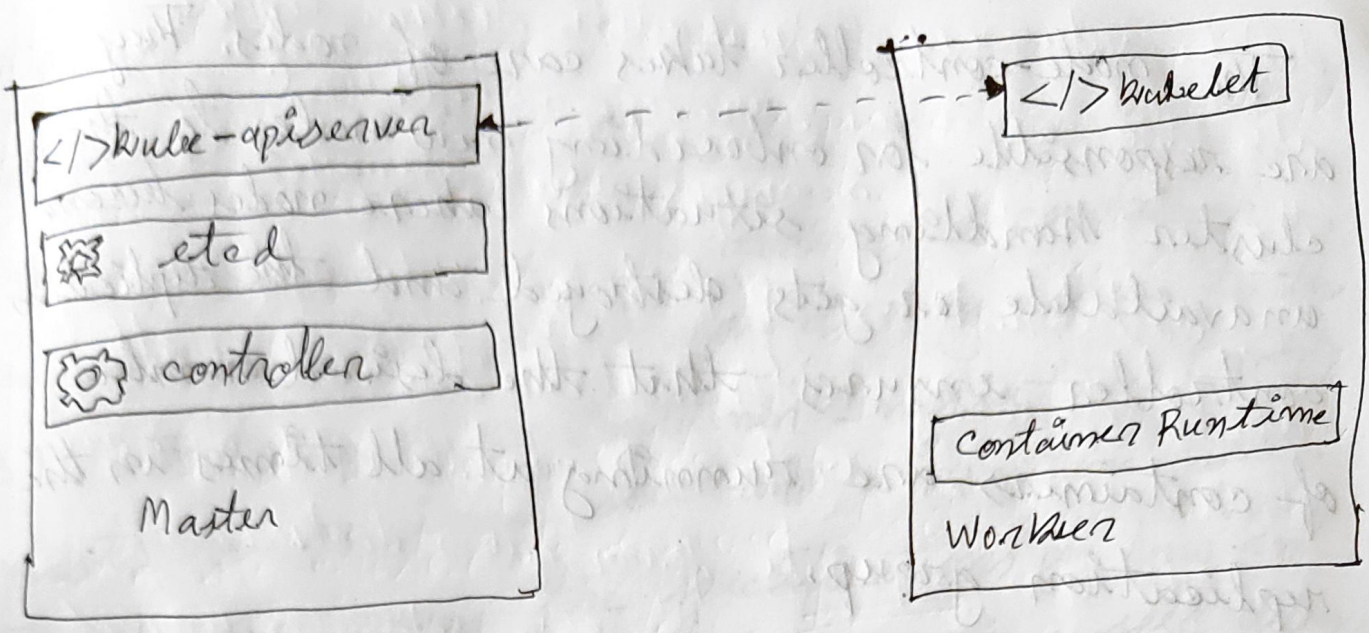
Kubelet: Kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

How these components distributed in master and worker?
~~To run docker containers on a system, cont~~

The master server has the kube API server and that is what makes it a master.

The worker nodes have the kubelet agent that is responsible for interacting with a master to provide health information of the worker node and carry out actions requested by the master on the worker nodes. All the information gathered are stored in a key value store on the master. The master also has the controller manager and the scheduler.

02/06/24



Kubectl: Kubectl is used to deploy and manage applications on a Kubernetes cluster. To get cluster information, to get the status of other nodes in the cluster and to manage many other things.

The purpose of Kubernetes is to host applications in the form of containers in an automated fashion.

It The deployment of as many instances of the particular application as required and easily enable communication.

Controllers - Node controller, Replication controller.

02/06/24

The node-controller takes care of nodes. They are responsible for onboarding new nodes to the cluster handling situations where nodes become unavailable or gets destroyed and the replication controller ensures that the desired number of containers are running at all times in the replication group.

Kube proxy

Communication between worker nodes are enabled by another component that runs on the worker node known as the kube-proxy service. The kube proxy service ensures that the necessary rules are in place on the worker nodes ~~to allow the containers~~ to allow the containers running on them to reach each other.

—o—

02/06/24

Docker vs Containerd

Container Runtime Interface (CRI)

CRI allows any vendor to work as a container runtime for kubernetes as long as they adhere to the OCI standards. Open Container Initiative (OCI) consists of an image spec and a runtime spec. Image spec means the specifications on how an image should be built. The runtime spec defines the standards on how any container should be developed. Rocket and other container runtimes that adhered to the OCI standards, are now supported as container runtimes for kubernetes via the CRI. Docker wasn't built to support the CRI standards.

Kubernetes introduced what is known as dockershim, a temporary way to continue to support docker outside of CRI.

Docker is not a container runtime alone. Docker consists of docker cli, docker api, docker build,

02/06/24

docker volumes, docker auth, docker security, also the container runtime called Runc and the daemon manages which is called containerd.

So, containerD is CRI compatible and can work directly with Kubernetes as all other runtimes. containerD can be used as a runtime on its own seperate from docker.

This is the reason why docker-shim is removed and docker support stopped. All the docker images are built on containerD, so docker-shim is unnecessary.

containerD

containerD can be installed without installing docker.

ctr comes with containerD, made for debugging containerD, only supports limited set of features. Any other way to interact with containerD, is making api calls directly.

02/06/24

nerdctl

→ nerdctl provides a docker-like CLI for containerD.
→ nerdctl supports docker compose
→ nerdctl supports newest features in containerD.
  └→ Encrypted container images
  → Lazy pulling
  → P2P image distribution
  → Image signing and verifying
  → Namespaces in Kubernetes.

crictl

→ crictl provides a CLI for CRI compatible container runtimes.
  → Installed seperately
  → Used to inspect and debug container runtimes
    └→ Not to create containers ideally.
  → Works across different runtimes

—o—

02/06/24

etcd

ETCD is a distributed reliable key-value store that is simple, secure and fast.

'etcd' service runs on port 2379. The default client of 'etcd' is etcd control client. It is a command line client to store and retrieve key value pairs.

etcd stores information about nodes, pods, configs, secrets, accounts, roles, bindings and others.

If kubernetes cluster is deployed from scratch, deploy etcd by downloading the ~~ect~~ etcd binaries and configuring etcd as a service in the master node. There are many options passed into the service, a number of them relate to certificates. The others are about configuring etcd as a cluster.

The '--advertised-client-urls https://${INTERNAL_IP}:2379', is the address on which etcd

02/06/24

listens. It happens to be on the ip address of the server and on port 2379. This is the URL that should be configured on the kube-api server when it tries to reach the etcd server.

If cluster is deployed using ~~kubeadm~~ kubeadm, then kubeadm deploys the etcd server as a pod in the kube-system namespace.

The 'etcd' database can be explored using the 'etcdtl' utility within this pod.

In the high availability environment, there will be multiple master nodes in the cluster, so multiple etcd instances across master nodes. In that case, make sure to specify the etcd instances know about each other by setting the right parameter in the etcd service configuration.

The initial-cluster option is where it must be specified the different instances of the etcd service.

—o—

02/06/24

Kube API Server

When kubectl command is executed, the kubectl utility is reaching to the kube-apiserver. The kube-api server first authenticates the request and validates it. It then retrieves the data from the etcd cluster and responds back with the requested information.

Instead of ~~kube-api~~ kubectl, the api can also be invoked by sending a post request.

Looks at an example by creating a pod, like before the ~~requested~~ is authenticated first and then validated. In this case, the API server creates a pod object without assigning it to a node, updates the information in the etcd server, updates the user that the POD has been created.

The scheduler continuously monitors the API server and realizes that there is a new pod

02/06/24

with no node assigned. The scheduler identifies the right node to place the new pod on and communicates that back to the kube-api server. The api server then updates the information in the eted cluster. The api server then passes that information to the kubelet in appropriate worker node.

The kubelet then creates the pod on the node and instructs the container runtime engine to deploy the application image. Once done, the kubelet updates the status back to the api server and the api server then updates the data back in the eted cluster. A similar pattern is followed everytime a change is requested.

The kube-api server is at the center of all the different tasks that needs to be performed to make a change in the cluster. ~~Kube~~ kube-api server is the only component that interacts directly with the eted datastore.

If cluster is deployed using ~~kubeadm~~ kubeadm,

02/06/24

process
the previously mentioned is not visually noticeable. If Kubernetes is deployed from the scratch, the kube-api server is available as a binary in the Kubernetes release page.

—o—

Kube Controller Manager

Kube controller manager manages various controllers in Kubernetes. The controller is a process that continuously monitors the state of various components within the system and works towards bringing the whole system to the desired functioning state. ~~It works towards bringing the whole system to the desired functioning state.~~

The ~~co~~ node controller is responsible for monitoring the status of the nodes and taking necessary actions to keep the application running. It does that through the kube-api server.

03/06/24

The node controller checks the status of the nodes every 5 seconds. That way the node controller can monitor the health of the nodes. If it stops receiving heartbeat from a node, the node is marked as unreachable but it waits for 40 seconds before marking it unreachable. After a node is marked unreachable, it gives it five minutes to come back up. If it doesn't, it removes the pods assigned to that node and provisions them on the healthy ones if the pods are part of a replica set.

Replication Controller: It is responsible for monitoring the status of replica sets and ensuring that the desired number of pods are available at all times within the set. If a pod dies, it creates another one.

These are the two examples of the controller. There are many more such controllers available within Kubernetes such as deployment controller, namespace controller

job controller etc.

All the controllers are packaged into a single process known as Kubernetes controller manager.

If kube-controller-manager is set up with the kubeadm tool, ~~kubed~~ kubeadm deploys the kube-controller-manager as a pod in the kube-system namespace on the master node.

The kube-controller-manager configuration file is ~~located~~ in /etc/kubernetes/manifests/kube-controller-manager.yaml.

For non kubeadm, it is /etc/systemd/system/kube-controller-manager.service.

—o—

Kube Scheduler

Kubernetes scheduler is responsible for scheduling pods on nodes. Scheduler is only responsible for deciding which pod goes on which node. It doesn't actually place the pod on the nodes. That's the

03/06/24

pod of the kubelet. The scheduler only decides which pod goes where.

The scheduler looks at each pod and tries to find the best node for it. Scheduler has two phases to find the best node for the pod.

Filter Nodes: The scheduler tries to filter out the nodes that do not fit the profile for ~~this~~ the pod. For example, the nodes that do not have sufficient CPU and memory resources requested by the pod. So the unsupported nodes are filtered out.

Rank Nodes: The scheduler uses a priority function to assign a score to the nodes on a scale of 0 to 10. For example, the scheduler calculates the amount of resources that would be free on the nodes after placing the pod on them.

These processes can be customized and one can write one's own scheduler.

Kubelet: Kubelet is the sole point of contact from

the master node. They load or unload containers as instructed by the scheduler on the master. The kubelet in the Kubernetes worker node, registers the node with the Kubernetes cluster. When it receives instructions to load a container or a pod on the node, it requests the container run time engine, which may be docker, to pull the required image and run an instance.

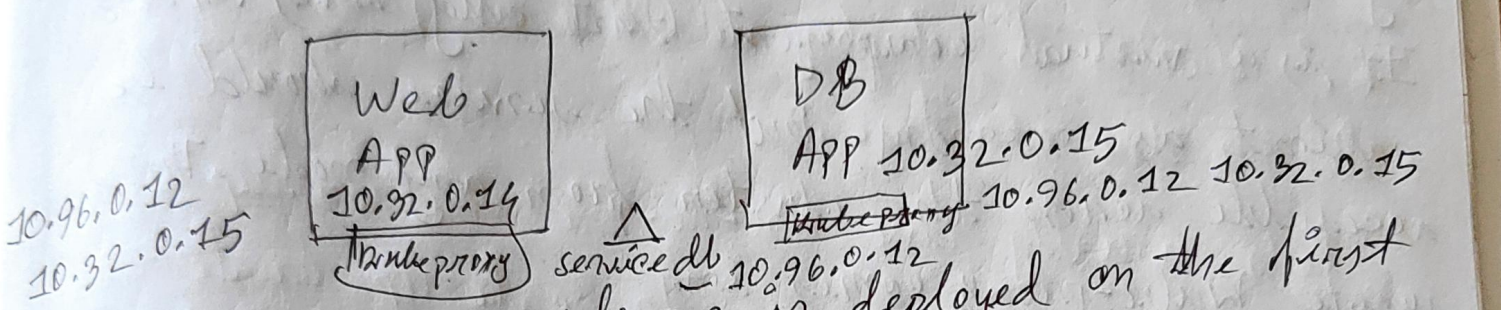
The kubelet then continues to monitor the state of the pod and the containers in it and reports to the kube-api server on a timely basis.

Kube-proxy

Within the Kubernetes cluster every ~~pod~~ pod can reach every other pod. This is accomplished by deploying a pod networking solution to the cluster. A pod network is an internal virtual network that spans across all the nodes in the cluster to which all the pods connect to. Through this network ~~are~~, they are able to communicate

03/06/24

with each other. There are many solutions available for deploying such a network.



In this case, a web app is deployed on the first node and a db app is deployed on the second node. The web app can reach the db, simply by using the ip of the db pod. But there is no guarantee that the ip of the db will always remain the same.

A better way for the web application to access the db is using a service. So a service is created to expose the db application across the cluster. The web app can now access the db using the name of the service db. The service also gets an ip address assigned to it whenever a pod tries to reach the service using its ip or name, it forwards the traffic to the backend pod in this case it is db.

03/06/24

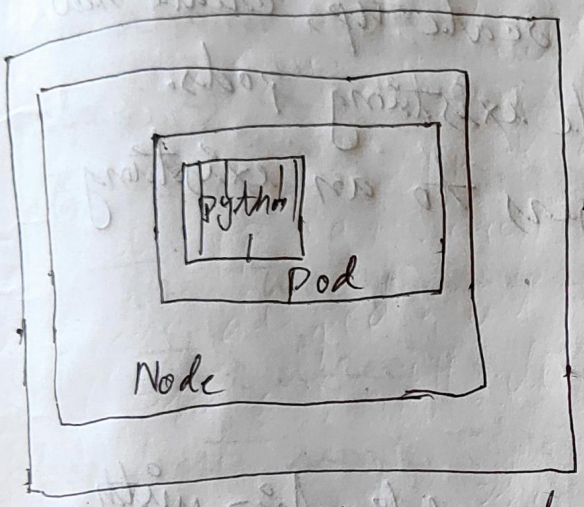
The service can't join the pod network because the service is not an actual thing, it doesn't have any interfaces or an actively listening process. It is a virtual component that only lives in the cabinet as memory. But the service should be accessible across the cluster from any node. That's where kube-proxy comes in.

kube-proxy is a process that runs on each node in the kubernetes cluster. Its job to look for new services and every time a new service is created, it creates the appropiate rules on each node to forward traffic to those services to the backend pods. One way it does this, is using iptables rules. In this it creates an IP tables rule on each node in the cluster to ~~for~~ forward traffic heading to the ip of the service which is 10.96.0.12 to the ip of the actual pod which is 10.32.0.15.

03/06/24

Pod

Kubernetes doesn't deploy containers directly on the worker nodes. The containers are encapsulated into Kubernetes object known as pods. A pod is a single instance of an application. A pod is the smallest object that can be created in Kubernetes.



In the pic, a single node Kubernetes cluster with a single instance of an application running in a single docker container encapsulated in a pod.

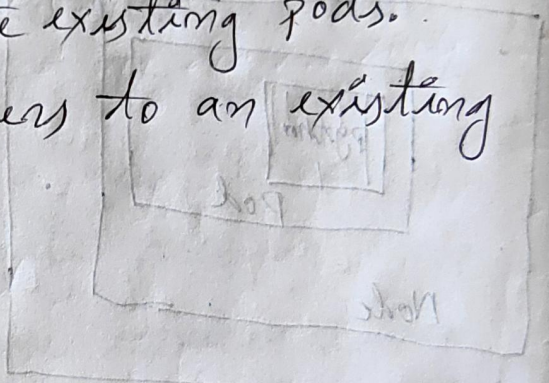
What if the number of users accessing the app increases? and you need to scaling the app? is needed? Additional instances of web app are needed to add to share the load.

Now, where the additional instances would spin up? A new pod with a new instance of the same app has to bring up. At present, there are two instances of the web app running on two seperate pods on the

same Kubernetes ~~says~~ system or node.

What if the user-base further increases and the current node has no sufficient capacity? The solution is, deploy additional pods on a new node in the cluster.

Pods usually have one-to-one relationship with containers running app. To scale up, create new pods, and to scale down, delete existing pods. Don't add additional containers to an existing pod to scale the app.



Multi-Container Pods

Generally, pods have one-to-one relationship with the containers. A single pod can have multiple containers except for the fact that they are usually not multiple containers of the same kind. If the intention is to scale the app, we would need to create additional pods.

Sometimes, a scenario is, there is a helper

03/06/24

container that might be doing some kind of supporting task for the web application, such as processing a user-entered data, processing a file uploaded by the user, etc, and these helper containers to live alongside the app container. In that case, both of these containers part of the same pod, so that, when a new application container is created, the helper is also created. When it dies, the helper also dies since they are part of the same pod.

The two containers can also communicate with each other directly by referring to each other as localhost since they share the same network space. Also, they can easily share the same storage space as well.

—o—

04/06/24

pod-definition.yml

apiVersion, kind, metadata, spec

sudo vim pod.yaml

04/06/24

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx

spec:
  containers:
  - name: nginx  # '-' means the first element of this list
    image: nginx # 'image' - docker image, same name
                 #as mentioned in docker hub
```

:wq

```
kubectl apply -f pod.yaml
pod/nginx created

kubectl get pods
NAME    READY   -  -  -
nginx   0/1     -  -  -  -  -

kubectl describe pod nginx
Name: nginx
-  -  -  -  -  -  -
```

04/06/24

Attend in the exam. (KodeKloud)

Replica Sets

To prevent users from losing access to the app, more than one pod or instance required to run at the same time.

The replication controller helps to run multiple instances of a single pod in the Kubernetes cluster thus providing high availability. Even if, there is a single pod, the replication controller can help by automatically bringing up a new pod when the existing one fails. Thus the replication controller ensures that the specified number of pods are running at all times whether it is just 1 or 100.

Another reason of the necessity of the replication controller is to create multiple pods to share the load across them. For example, ~~for~~ ~~a~~ single pod serving a set of users. When the number of users increase, an additional pod can be deployed to balance the load

04/06/24

across the two pods. If the demand further increases and resources are running out on the first node, additional pods can be deployed across the other nodes in the cluster. The replication controller spans across the multiple nodes in the cluster. It helps to balance the load across multiple pods on different nodes, as well as scale the app when the demand increases.

It is important to note that there are two similar terms - replication controller and replica set. Both have the same purpose, but they are not the same. Replication controller is the older technology that is being replaced by replica set. (2:51)

~~Rept~~ ~~The~~ In the pod definition file, replication controller is supported in api version 1.

For any kubernetes definition file, the spec section defines what's inside the object is being created. In this case, the replication controller creates

04/06/24

multiple instances of a pod. create a template section under spec to be used by the replication controller to create replicas.

rc-definition.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
```

04/06/24

kubectl create -f rc-definition.yml

kubectl get replicationcontroller

ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: front-end

spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

(Same level as replicas)

04/06/24

The major difference between replication controller and replica set is replica set requires a selector definition. The selector section helps the replica set to identify what pods fall under it. It's because replica set can also manage pods that were not created as part of the replica set creation.

The match labels selectors simply matches the labels specified under it to the labels on the pod.

Labels and Selectors

In a scenario, three instances of front-end web app are deployed. A replication controller or replica set to ensure that I have three active pods at any time. Replica set can be used to monitor existing pods. The role of replica set is to monitor the pods and if any of them fails, deploy new ones.

The labels can be used as filter for the replica set.

04/06/24

under the 'selector' section, the 'matchLabels' filter is used and provide the same label that I used while creating the pods. This way the replica set knows which pods to monitor. The same concept of labels and selectors is used in many other places throughout Kubernetes.

Scale

1) Update the number of replicas in the definition file to the desired number. Then

```
Kubectl replace -f replicaset-definition.yml
```

2) Kubectl scale --replicas=6 -f replicaset-definition.yml

3) Kubectl scale --replicas=6 replicaset myapp-replicaset.

04/06/24

vi replicaset-definition-1.yaml

```
apiVersion: apps/v1
kind: Replicaset
metadata:
  name: replicaset-1
spec:
  replicas: 2
  selector:
    matchLabels
      type: frontend
  template:
    metadata:
      labels:
        type: frontend
    spec:
      containers:
      - name: nginx
        image: nginx
```

Deployment

The ~~depl~~ deployment provides us with the capability to upgrade the underlying instances seamlessly using

04/06/24

rolling updates, undo changes and pause and resume changes are required.

First, create a definition file. The content of the deployment definition file are exactly similar to the replica set definition file, except for the 'kind' which is going to be deployment. deployment-definition.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  (Rest same level as template)
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

04/06/24

kubectl create -f deployment-definition.yml

kubectl get deployments

The deployment automatically creates a replica set.

kubectl get replicaset

—o—

05/06/24

Services

I deployed a pod having a web application running on it. The Kubernetes node has an ip address and it is 192.168.1.2. The internal pod network is in the range 10.244.0.0 and the pod has an ip 10.244.0.2. Doing a 'curl' from the node network (192.168.1.0) will be successful.

```
curl http://10.244.0.2
```

But this is from the node, ~~out~~ for the webserver outside access is necessary.

The Kubernetes service is object like pods. One of its use case is to listen to a port on the node and forward request on that port to a port

on the pod running the web app. This type of service is known as a node-port service because the service listens to a port on the node and forward requests to the pods. There are other kind of services available.

Service Types

NodePort

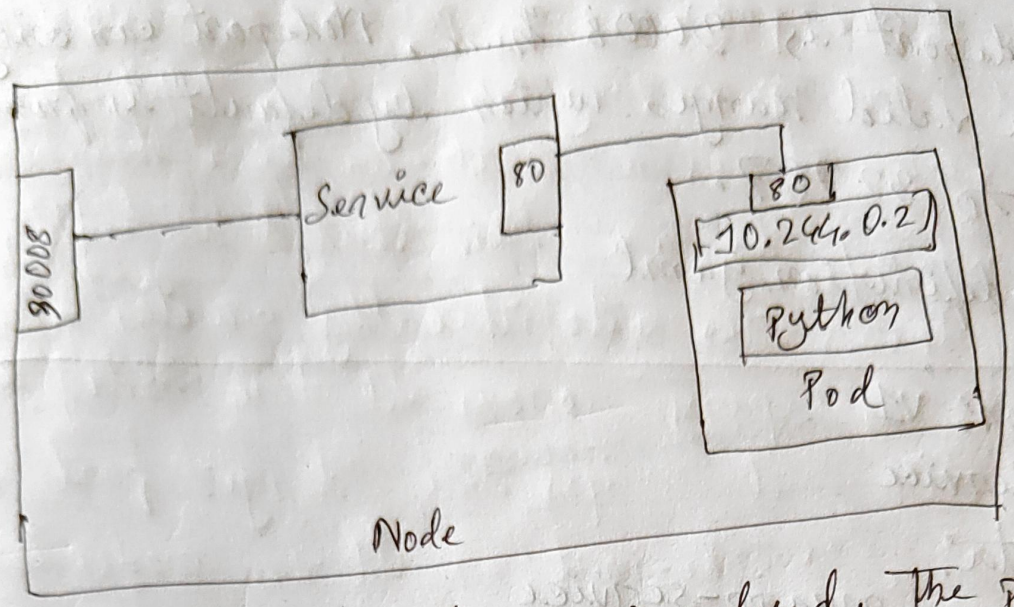
ClusterIP – The service creates a virtual ip inside the cluster to enable communication between different services, such as a set of front-end servers to set a back-end servers.

Load balancer: Provisions load balancer in supported cloud providers. Example: distribute load across the different web servers in the front-end tier.

NodePort :-

This service helps by mapping a port on the node to a port on the pod.

05/06/24



In the image, three ports are involved. The port of the pod where the actual web server is running is 80. It is referred to as the target port because that is where the service forwards the request to. The second port is the port on the service itself. It is simply referred to as the port. The service is infact like a virtual server inside the node. Inside the cluster, it has its own ip address, that ip address is called the cluster ip of the service.

Finally, the port on the node itself, which is used to access the web server externally, and that is known as NodePort.

05/06/24

The nodeport is 30008 here. Nodeport can only be in a valid range, which by default is from 30,000 to 32767.

service-definition.yaml

```
apiVersion: v1
kind: Service
metadata:
    name: myapp-service
spec:
    type: NodePort
    ports:
      - targetPort: 80
        port: 80
        nodeport: 30008
    selector:
        app: myapp
        type: front-end
```

(Same level as ports)

If 'targetPort' is not mentioned, it will be the same as the 'port', if the 'nodeport' is not mentioned, a port from 30000 - 32767 is automatically allocated. Multiple port mapping within a single service is ~~pass~~ a way of conf. There is nothing in the definition file that connects

05/06/24

the service to the pod. The target port on which pod is not mentioned. Labels and selectors will be used to link these together. ~~---~~ create a new section 'selector', copy the 'app' and 'type' from the pod definition file. This links the service to the pod.

```
kubectl create -f service-definition.yml
```

```
kubectl get services
```

```
curl http://192.168.1.2:30008
```

In the production environment, multiple instances of web application running for high availability and load balancing purposes. In my example, multiple instances (pods) are running with same labels (same key:value). The same label is used as 'selector' during the creation of the service. ~~When the~~ s

In a scenario when the service is created, it looks for a matching pod with the label and finds three of them. The service then automatically selects

all the three pods as endpoints to forward the external requests coming from the user. Thus the service acts as a built-in load balancer.

When the pods are distributed in multiple nodes, Kubernetes automatically creates a service that spans across all the nodes in the cluster and maps the 'target port' to to the same 'node-port' on all the nodes in the cluster. This way one can access the app using the ip of any node in the cluster and using the same port number which in this case 30008.

Cluster IP

A Kubernetes service can help group the pods together and provide a single interface to access the pods in a group. For example, a service created for the backend pods will help group all the backend pods together and provide a single interface for other pods to access this service. The requests are forwarded to one of the pods

05/06/24

under the service randomly. Similarly create additional services for other service like redis and allow the backend pods to access the redis systems through the service.

This enables to easily and effectively deploy a microservices based app on Kubernetes cluster. Each layer can now scale or move as required without impacting communication between the various services. Each service gets an ip address and name assigned to it inside the cluster and that is the name that should be used by other pods to access the service. This type of service is known as cluster ip. To create such a service as always use a definition file.

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
    name: back-end
spec:
    type: ClusterIP
    ports:
    - targetPort: 80
      port: 80
```

(same level as ports) label as pod)

```
  selector:
    app: myapp
    type: back-end
```

05/06/24

If the type is not mentioned, it will automatically set as 'ClusterIP', it's the default type.

```
kubectl create -f service-definition.yml
```

```
kubectl get services
```

This service can be accessed by other pods using the cluster-ip or the service name.

Service - Load Balancer

Kubernetes has support for integrating with the native load balancers of certain cloud providers and configuring that.

In a full stack app, there can be multiple nodes and pods. For example, frontend apps are deployed with node-port 30035 and backend apps with 31060. But to the end users, ~~the URL is~~ these are accessible using node ~~urls~~ ips but users need a single url. Load balancer service can do this.

service-definition.yml

05/06/24

```
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
  type: LoadBalancer
  ports:
   - targetPort: 80
     port: 80
     nodePort: 30008
```

If it is configured, in the unsupported platform, it will act like node-port service.

—o—

Namespace

The default namespace in the Kubernetes is created when the cluster is first set up. Kubernetes, creates a set of pods and services for its internal purpose, such as those required by the networking solution, the dns service etc. To isolate these from the user and to prevent from accidental deletion or modifying these services, Kubernetes creates them under another namespace created at cluster startup - named kube-system. A third namespace created

05/06/24

by Kubernetes ~~that is~~ automatically, is called kube-public. This is where resources should be made available to all users are created.

If I want to use the same cluster for both dev and production environment but at the same time isolate the resources between them, I can create a different namespace for each of them. Each of the namespaces can have its own set of policies that define who can do what. Quota can be assigned of resources to each of these namespaces. In this way each namespace is guaranteed a certain amount and doesn't use more than its allowed.

The resources within a namespace can refer to each other simply by their names. If required a pod can reach a service in another namespace as well. For this, one must append the name of the namespace to the name of the service. For example, for a web pod in the default namespace to connect to the database in the dev environment or namespace.

05/06/24

Use

```
servicename.namespace.svc.cluster.local
```

When the service is created, a dns entry is added automatically in this format. Looking closely at the DNS name of the service, the last part 'cluster.local' is the default domain name of the kubernetes cluster. 'svc' is the subdomain for service.

```
kubectl get pods --namespace=kube-system
```

A pod is by default created in the default namespace.

To create a pod in another namespace

```
kubectl create -f pod-definition.yml --namespace=dev
```

To mention it inside the pod definition file -

```
metadata:
  -  -
    namespace: dev
```

Create a namespace -

05/06/24

namespace-dev.yml

```
apiVersion: v1
kind: Namespace
metadata:
    name: dev
```

```
kubectl create -f namespace-dev.yml
```

```
kubectl create namespace dev
```

To switch the namespace permanently

```
kubectl config set-context $(kubectl config current-context) --namespace=dev
```

To see all the pods in all the namespaces

```
kubectl get pods --all-namespaces
```

Limit resources in a namespace using resource quota

compute-quota.yml

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-quota
    namespace: dev
spec:
    hard
        pods: "10"
        requests.cpu: "4"
        requests.memory: 5Gi
        limits.cpu: "10"
        limits.memory: 10Gi
```

05/06/24

`kubectl create -f compute-quota.yaml`

—o—

Imperative vs Declarative

In the Infrastructur as Code (IaC), there are different approaches in managing the infrastructure. They are classified into - imperative and declarative approaches.

An example of an imperative approach of provisioning infra, would be a ~~it~~ set of instructions written step-by-step, such as ~~provisioning a VM~~

→ provisioning a vm by the name web-server
→ Install nginx software in it.
→ Edit configuration file to use ~~t~~ port '8080'.
→ Edit configuration file to the web path '/var/www/nginx'
→ Load web pages to '/var/www/nginx' git repo -x
→ Start nginx server

Here, what is required and how to get things done are clearly mentioned.

05/06/24

Example of declarative approach -

VM Name: web-server

Package: nginx

Port: 8080

Path: /var/www/nginx

Code: GIT Repo - X

In the declarative approach, the requirement is declared. Orchestration tools - Ansible, Terraform, puppet, chef.

In the imperative approach what happens, if the first time only half of the steps were executed?

What happens if the engineer provides the same set of instructions again to complete the remaining steps?

To handle such situations, there will be many additional steps involved, such as checking to see if something already exists and taking an action based on the results of that checks.

For example, while provisioning a VM, what

05/06/24

would happen if a vm by the name 'web-server' already exists? The same goes with creating database or important data. Should it fail or should it continue since the vm is already there? What if upgrading the version of software is decided to say nginx 1.18 in the future? It should be as simple as updating the version of nginx in the configuration file, and the system should take care of the rest. Ideally, the system should be intelligent enough to know what has already been done and apply the necessary changes only. That's the declarative way of doing things.

In the Kubernetes world, the imperative way of managing infrastructure using commands like the 'kubectl run' command to create a pod. The 'kubectl create deployment' to create a deployment, the 'kubectl expose' command to create a service to expose a deployment and the 'kubectl edit' command may be used to edit

an existing object. For scaling a deployment or replicaset, use the 'kubectl scale' command. Updating the image on a deployment, use the 'kubectl set image'.

I have also used object configuration files to manage objects such as creating an object using the "kubectl create -f' command, with '-f' option to specify the object configuration file, and editing an object using the 'kubectl replace' command, and deleting an object using the 'kubectl delete' command.

The declarative approach would be to create a set of files that defines the expected state of the applications and services on a Kubernetes cluster. With a single 'kubectl apply' command, Kubernetes should be able to read the conf. files and decide by itself what needs to be done to bring the infra to the expected state

05/06/24

So in the declarative approach, I will run the 'kubectl apply' command for creating, updating or deleting an object. The apply command will look at the existing configuration and figure out what changes need to be made to the system.

—o—

Within the imperative approach there are two ways

→ The first is using imperative commands such as the run, create or expose commands

kubectl Apply

The apply command takes into consideration the local configuration file, the live object definition on Kubernetes and the last applied configuration before making a decision on what changes are to be made. When the apply command is run, if the object doesn't already exist, the object is created. When the object is created, an object configuration similar to what is created

05/06/24

locally within Kubernetes but with additional fields to store status of the object. This is the live configuration of the object on the Kubernetes cluster. This is how Kubernetes internally stores information about an object no matter what approach is used to create the object.

When the 'kubectl apply' is used to create an object, the yaml version of the local object configuration file is converted into a json format and it is then stored as the last applied configuration. Going forward for any updates to the object, all the three are compared to identify what changes are to be made on the live object. For example, when the nginx image is updated to 1.19 in our local file, and the 'kubectl apply' command is run, this value is compared with the value in the live configuration and if there is a difference, the live configuration is updated with the

05/06/24

new value. After any change, the last applied json format is always updated to the late test so that its always up-to-date.

Why do then last applied configuration needed?
If a field is deleted, for example the 'type' label is deleted and now when the 'kubectl apply command' is run, anyone can see the last applied configuration had a label, but it is not present in the local configuration. This means that the field needs to be removed from the live configuration. If a field was present in the live configuration and not present in the local or the last applied configuration, then it will be left as is. If a field is missing from the local file, and it is present in the last applied configuration, that means that in the previous step or whenever the last time, the 'kubectl apply' command is run, the particular field was

there, and it is now being removed. So, the last applied configuration helps to figure out what fields have been removed from the local file. That field is then removed from the actual live configuration.

Where is the json file that has the last applied configuration stored?

It's stored on the live object configuration on the Kubernetes cluster itself as an annotation named last applied configuration. This is only done when the apply command is used. The 'kubectl create' or 'kubectl replace' commands don't store the last applied configuration like this.

Once the apply command is used, going forward, whenever a change is made, the apply command compares all three sections, the local pod definition file, the live object configuration and the last applied configuration

05/06/24

stored within the live object configuration file for deciding what changes are to be made to the live configuration.

—o—

07/06/24

Git

D:/DevOps (Poridhi.io)/git

12/06/24

Kubernetes - Class 1

Docker container can also be run using api.

self healing - if a docker container is down, it will automatically up.

Kubernetes

Features to set up a system nearly similar to

1) container launch
2) self-healing
3) Health check → Agent → run the agent in an infinite loop, it will check the container state

if → HTTP 200 → alive

→ HTTP 404 → down; so up the container, remove the down container

⇒ req count - curr count = 0

12/06/24

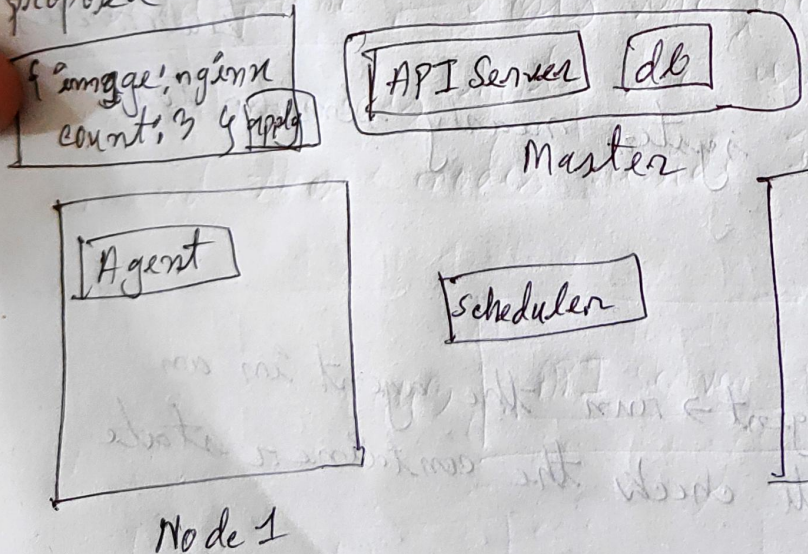
If requested count > current count, launch container

13/06/24

But all the applications doesn't http connection e.g. mysql, in that case tcp connection is needed to build (tcp probe).

TCP probe or HTTP probe can also be called as liveness probe.

But what happens if ~~one~~ the server is down? The whole operation is down. To solve this problem use multiple servers. The following scenario is proposed



The Agent(s) will send health checks, if an agent

13/06/24

won't send heartbeat check the respective node is down.
The json file will look like -

```
{ image : nginx
  count : 3
}
```

In the pic, after 'apply', it will communicate to API server. The api server will ask scheduler, where the instances can be launched. The scheduler will take data from db to know the number of container in the respective node. The scheduler after getting info from db will tell to launch number of containers ~~on~~ in the node. For example, it can say ~~have~~ among 3 containers launch 2 on node 1 and 1 in node 2.

15/06/24

What happens if a container (in a pod) inside the node exists and the node is down? How the data will be retrieved?

The process which is used for data retention is

15/06/24

stateful. making the node stateful.
In general stateful apps (e.g. mysql) is not deployed in Kubernetes.

—o—

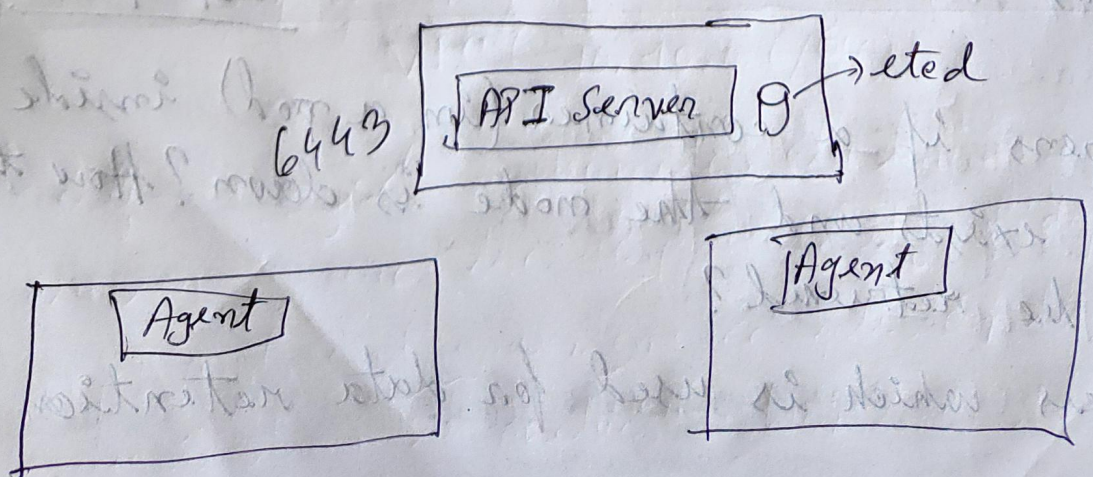
18/06/24

Class 2

Kubernetes has two parts - a) control plane
b) Data plane

Control plane: control all the nodes, master resides in the control plane.

Data plane: hosts the worker nodes, specifically the worker nodes hosts the user applications - and those nodes reside in the data plane.



16/06/24

Agent sends reports periodically to the API Server, api server stores report in a db named 'eted' which stores value in 'key-value' pair. API server listens on port 6443.

[Using kine, api server can use other dbs. [kine is an interface that lets the api server uses mysql, postgres etc.]

kubectl is a client, feed the yaml configuration to it. The kubectl converts the yaml into json. After that kubectl makes a GET/POST/PUT request to the API Server.

kubectl is a tool that uses a binary file written on 'Go'. If 'kubectl get pods' command is applied, it will send a 'GET' request to the api server. The api server will collect pod information from worker nodes and then 'kubectl' will show the response.

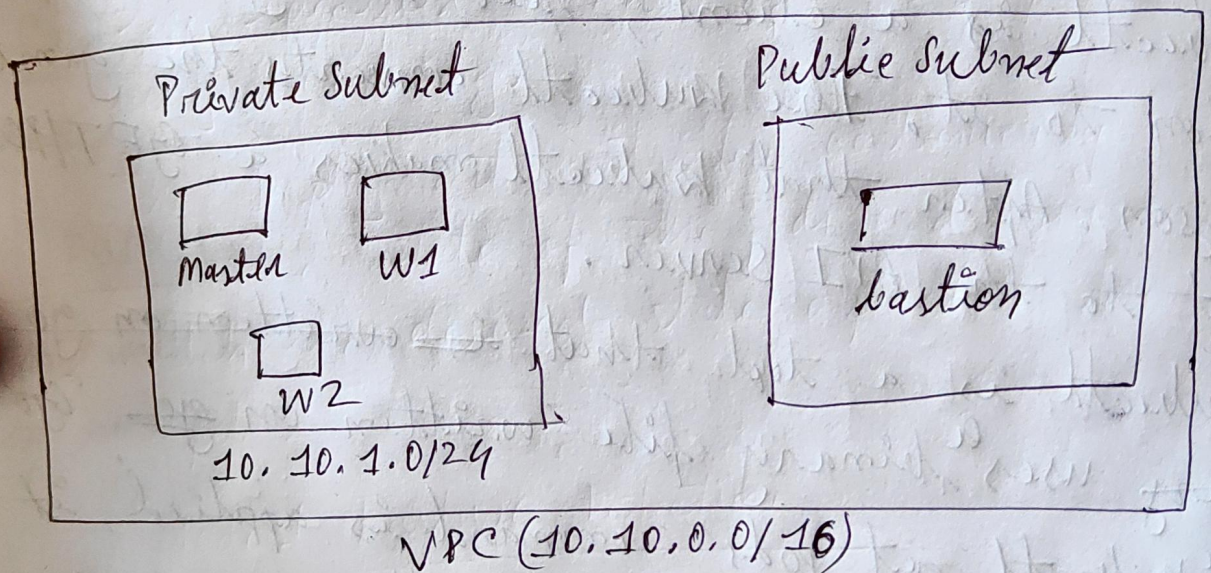
19/06/24

in poridhi lab it is K3S - lightweight version of Kubernetes. In K3S, cloud provider and storage provider, those two options are removed.

In EKS there is an opensource version and closed-source version.

In Kubernetes, Agent is known as Kubelet.

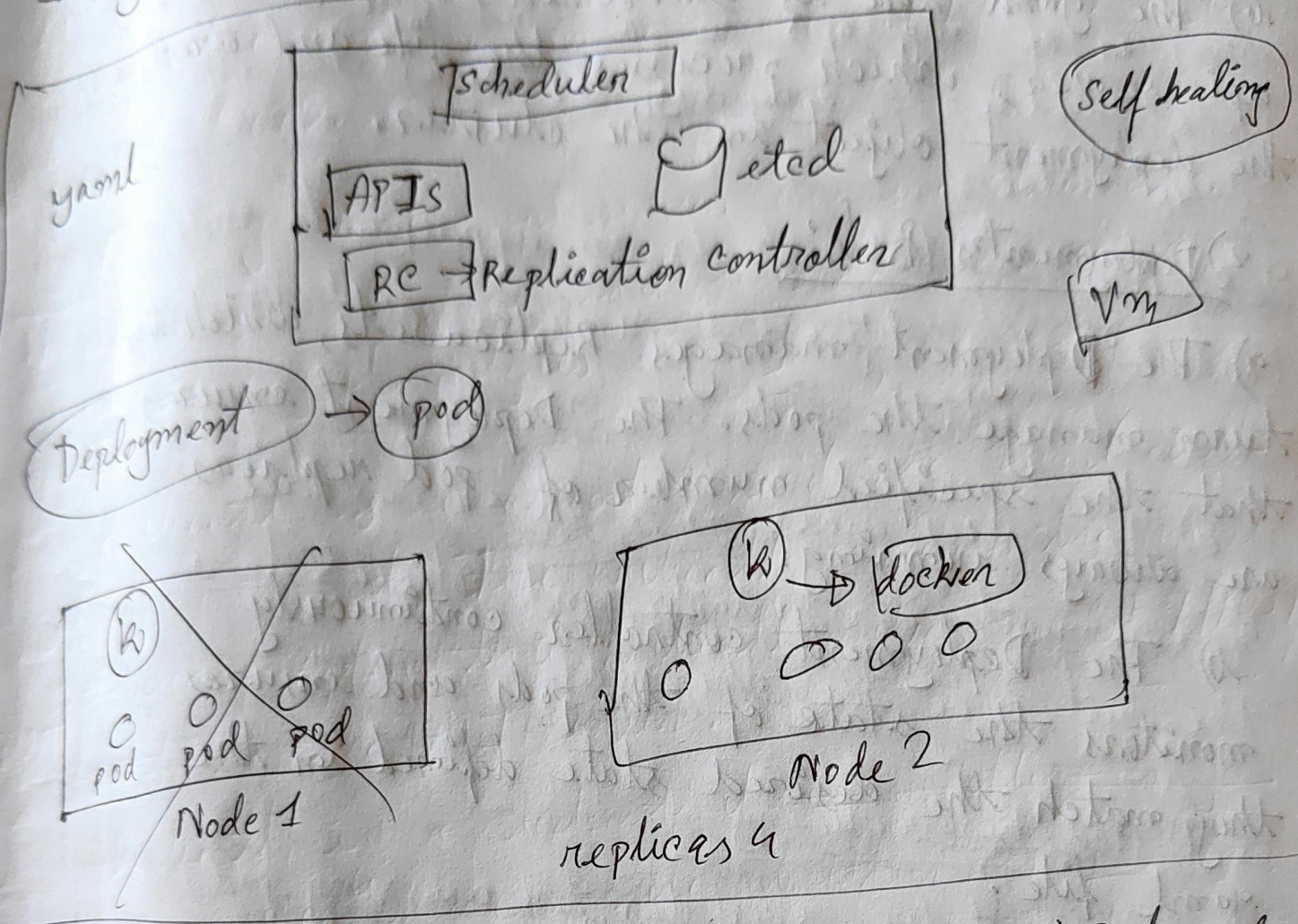
self-managed cluster



A self-managed Kubernetes cluster is a type of cluster where the responsibility of deployment, management, and maintenance falls on the user or organization themselves, rather than being

19/06/24

by a cloud provider on a managed service.
In general, db is prohibited ~~in~~ to run in kubernetes.



Detailed Steps and Role of Deployment, YAML File and Scheduler

1) YAML file and Deployment:

a) The user defines a 'Deployment' using a yaml file. This yaml file specifies the desired state of

19/06/24

the application, including the number of replicas, the container image to use, labels, etc.

b) The yaml file is submitted to the kubernetes api server, which processes the file and creates the deployment object in the cluster.

2) Deployment's Role:

a) The 'Deployment' manages ReplicaSets, which in turn manage the pods. The Deployment ensures that the specified number of pod replicas are always running.

b) The Deployment controller continuously monitors the state of the pods and ensures they match the desired state defined in the yaml file.

3) Node 1 Failure Detection

a) Kubernetes continuously monitors the health of all nodes. When Node 1 goes down, Kubernetes components (like the kubelet and node controller) detect the failure. Node 1 is marked as "NotReady".

19/06/24

4) Replication Controller's Role

a) The Replication Controller operates under the control of the Deployment. It ensures that the specified number of pod replicas are running at any given time.

b) When Node 1 goes down and the pods running on it are lost, the current state no longer matches the desired state defined by the Deployment.

5) Etcd and Desired State

a) The Deployment maintains the desired state information in etcd, which is key-value store all cluster data.

b) When Node1 goes down and the pods running on it are lost, the ~~curre~~ current state no longer matches the desired state stored in etcd.

6) Scheduler's Role

a) The Scheduler is responsible for placing pods onto nodes in the cluster based on resource

19/06/24

availability and requirements.

b) When the RC identifies that the desired number of replicas is not met due to the node failure, it triggers the Scheduler to take action.

c) The Scheduler finds an appropriate node (Node 2) with sufficient resources to run the required pods.

d) It makes decisions based on resource requirements, constraints, and policies to select the best node for the pods.

7) Pod Rescheduling

a) The Scheduler schedules the missing pods to run on Node 2.

8) Pod Creation on Node 2

a) The kubelet on Node 2, which is the agent responsible for managing pods on a node, receives the scheduling decision from the Scheduler.

19/06/24

b) The Kubelet then communicates with the container runtime (e.g. Docker) on Node 2 to start the pods.

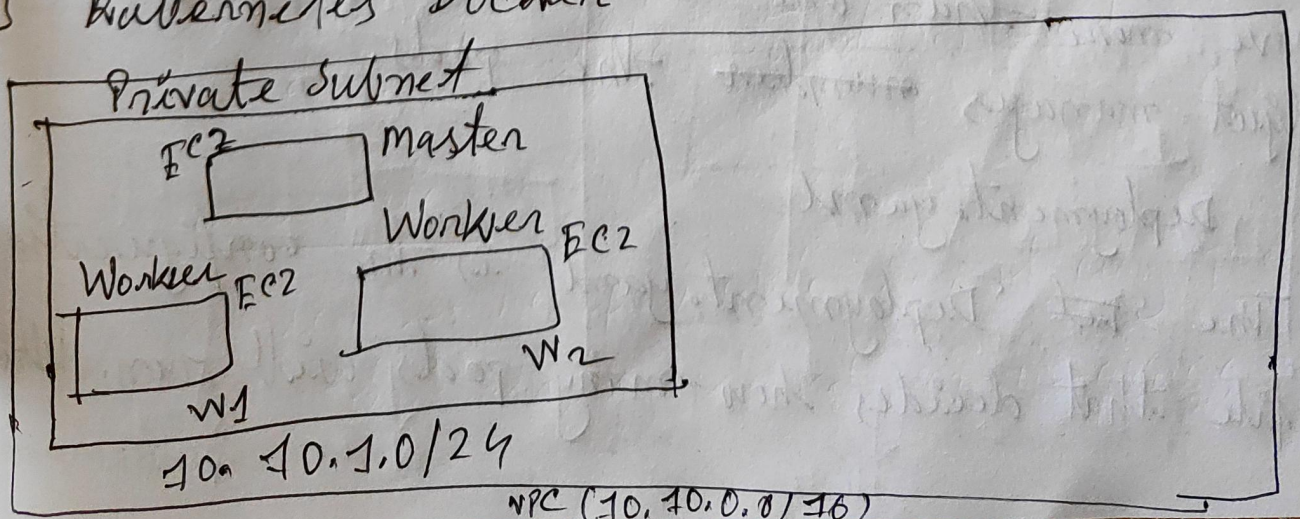
g) Restoration of Desired State

a) Once the pods are up and running on Node 2, the Deployment checks the cluster state again.

b) If the number of running pod replicas matches the desired state defined in the yaml file, the Deployment controller does nothing further. If there is still a discrepancy, the Deployment controller continues to instruct the Scheduler to create new pods until the desired state is achieved.

20/06/24

AWS Kubernetes Docker



20/06/24

Build a docker image and push it to the dockerhub. Kubernetes doesn't know containers, it knows pod.

In the pic, in a vpc, in the subnet, there are three EC2, one is master and other two are workers. Pull the customized nginx image that's uploaded in the dockerhub. Run the image, so the container starts. The nginx container runs inside a pod. So in the pod, there can be an nginx container and fluentd container. The task of fluentd is to send the nginx logs into the S3 bucket.

~~Pod can be~~ is considered Pod is an object. ~~It's~~ Its task is to manage container. A pod can have more than one containers. The Deployment object manages ~~number~~ the pods.

Deployment.yaml

The ~~'Det~~ 'Deployment.yaml' is the configuration file that decides how many pods will run. Which

20/06/24

worker node will run the pods - not that, (can be decided manually). To apply the deployment configuration

```
kubectl apply -f "deployment filename"
```

After applying the ~~above~~ command above, this will 'POST' to the master. Master will notice the deployment name, number of containers and which node ~~has empty~~ will be able to run the pod. Then the pod will be created.

```
kubectl get pods -o wide
```

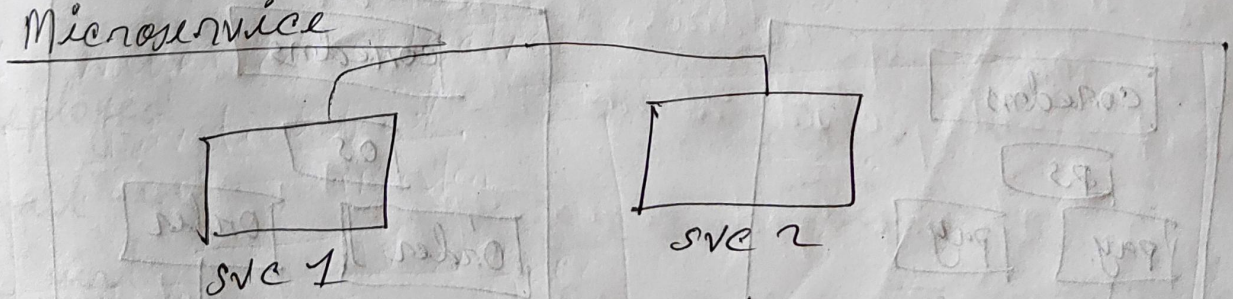
location of the pods ~~on~~ in a particular node.

—o—

24/06/24

Class 7

Microservice



In docker ~~container~~, two containers can communicate via a bridge docker created.

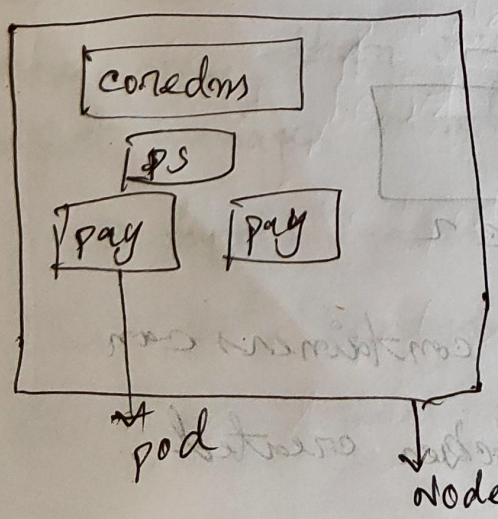
24/06/24

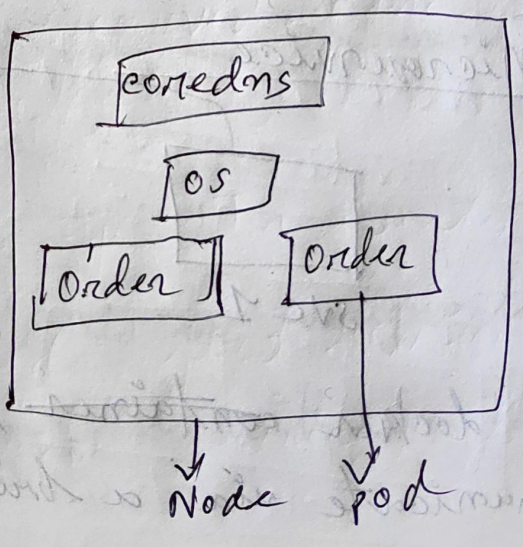
Service Discovery

In regular scenario, if a name resolution is done by dns server. If local dns (/etc/hosts), has an entry, name resolution is done by it. If an ip address of the host is changed or the host is down, the local dns entry needs to be changed. Thing to remember, the local dns is in the gateway. In the container enviroment, also every container configuration has to be updated.

CoreDNS (is a pod)

The CoreDNS, can solve the problem above.





24/06/24

In two different nodes, there are payment (pay) and order (Order) pods.

OS (Order service) and PS (payment service) sends packets to the respective pods. In ~~-~~ coredns, service entry is made.

25/06/24

So in the CoreDNS there will be entry like the following -

orderSvc ~~- 'ip address'~~ → 11.10.0.2

paySvc ~~- 'ip address'~~ → 11.10.0.3

Here the ip address is the virtual ip.

service ip cidr

The service is an object. The service object sends traffic and load balances. When the service is deployed, an entry is made in the CoreDNS a mail is sent to the service with an ip address. This ip address is a virtual ip or a service ip. This ip is not attached to a pod. This ip is set on every node. This ip is process iptable or ipvs.

25/06/24

A service has two parts

Service ip address	Pod IP Address
11.10.0.2	1
	2

As the service manages traffic, ~~the~~ packets first come to ~~the pod~~ service ~~first~~. Then the iptables (or ipvs) tells no pod with the service cirdr available. [Every service provider has ~~there~~ own system like 'iptables/ipvs' to manage.] Iptable will tell to go to pod's ip address '1' or '2'.

If a pod goes down, from the iptable that pod's ip address is removed. So the service won't send packet there.

~~If the pod comes up~~ In a sentence -

When the pod is called, first look in coredns, from coredns to service, after

25/06/24

that according to the service's iptable rule the packet is sent to the pod's ip address.

In service, on virtual ip is used for group of pods.

V. V. I - Interview

```
kubectl get ep
```

shows pods ip address.

26/06/24

Create an nginx deployment file -
sudo vim nginx_deployment.yaml

```
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:latest
        ports:
        - containerPort: 80
```

26/06/24

sudo vim nginx_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: clusterIP
```

Create the pods using deployment file

```
kubectl apply -f nginx_deployment.yaml
```

To see the pods

```
kubectl get pods
```

26/06/24

Get the node information ~~in detail~~ with additional details

```
kubectl get nodes -o wide
```

-o : output

Output

```
NAME ..... Internal-IP External-IP OS-Image
                                    Kernel-Version Container-Runtime
kubesinglemasternode2   192.168.0.20  <none>  Ubuntu 24.
                                    6.8.0-35-generic containerd://1.6.33
```

pod info

```
kubectl get pods -o wide
```

Output

```
Name ..... Restarts AGE IP Node .....
```

100 26/06/24

Get specific column specific column info

```
kubectl gets pods -o wide | grep -v NAME |
awk '{print $6, $7}'
```

with ~~stou~~ show information of the column 6 and 7.

'grep -v NAME' : filters out the line containing "NAME".

'| awk '{print $6, $7}' - print 6th and 7th column.

```
kubectl apply -f nginx_service.yaml
```

service

```
kubectl get svc
```

output

Name	.....	Cluster IP
.....		
nginx-service		10.110.232.168

The cluster ip is the virtual ip, this is the ip where the traffic before going to the pod

26/06/24

Get the pod ip address

```
kubectl get ep
```

Output

```
NAME          ENDPOINTS
nginx-service 10.32.0.2:80, 10.32.0.3:80, 10.40.0.3:80
```

Notice, using 'kubectl get svc', the service ip is '10.100.232.168' and from 'kubectl get ep' the ip of pods are '10.32.0.2', '10.32.0.3' and '10.40.0.3:80'. So when the packets come first it will go to '10.100.232.168' and then the pods ip addresses. The pods ip addresses are from kubernetes ~~network~~ CIDR not the service cidr.

For debugging kubernetes cluster, make a ubuntu container. In this ubuntu container, keep some ~~tol~~ tools for debugging purpose.

[The pod ip address, is the ip address of the container inside it. The pod is connected

26/06/24

a virtual
to the docker bridge via an ethernet cable.

Enter in the pod

```
kubectl exec -it nginx-deployment-576....-q.. sh
```

Inside the pod's container

```
apt update
apt install iproute2
```

```
ip a s
```

```
kubectl get pods -o wide
```

If the pods ip addresses are in the same network, they are in the same node.

27/06/24

Enter into ubuntu

```
kubectl exec -it ubuntu-....bash
```

```
sudo apt update
```

```
sudo apt dnsutils
```

```
nslookup nginx-service
```

27/06/24

Output:

```
Server:  10.96.0.10
Address: 10.96.0.10

Name:   nginx-service.default.svc.cluster.local
Address: 10.110.232.168
```

namespace — always same

In another worker node

```
kubectl get svc -n kube-system | grep kube-dns
```

output

```
kube-dns   clusterIP  10.96.0.10  . . . . . .
```

```
kubectl get pods -n kube-system
```

output

```
core-dns . . . . . .
```

The core-dns runs as a pod and works as a part of kube-dns service.

In ubuntu-pod

```
apt install iputils-ping, telnet
```

```
telnet nginx-service 80
Trying . . . .
 connected to nginx-service
```

27/06/24

```
ping 10.110.232.168
```

ping is failed because this ip address is not configured in an interface. For a service, a table create an ip but not belongs to any interface. This is the proof service ip being a virtual ip.

```
curl http://nginx-service
```

* In another worker node

```
kubectl logs -f nginx-deployment-.....
```

The 'service' acts a load balancer, in round-robin, ~~it sends packet to pod~~ fashion.

V, V, I

In production environment, everything has to be done from Master Node, No access to workers.

27/06/24

```
sudo iptables -t nat -L KUBE-SERVICES | grep
nginx-service
```

—o—