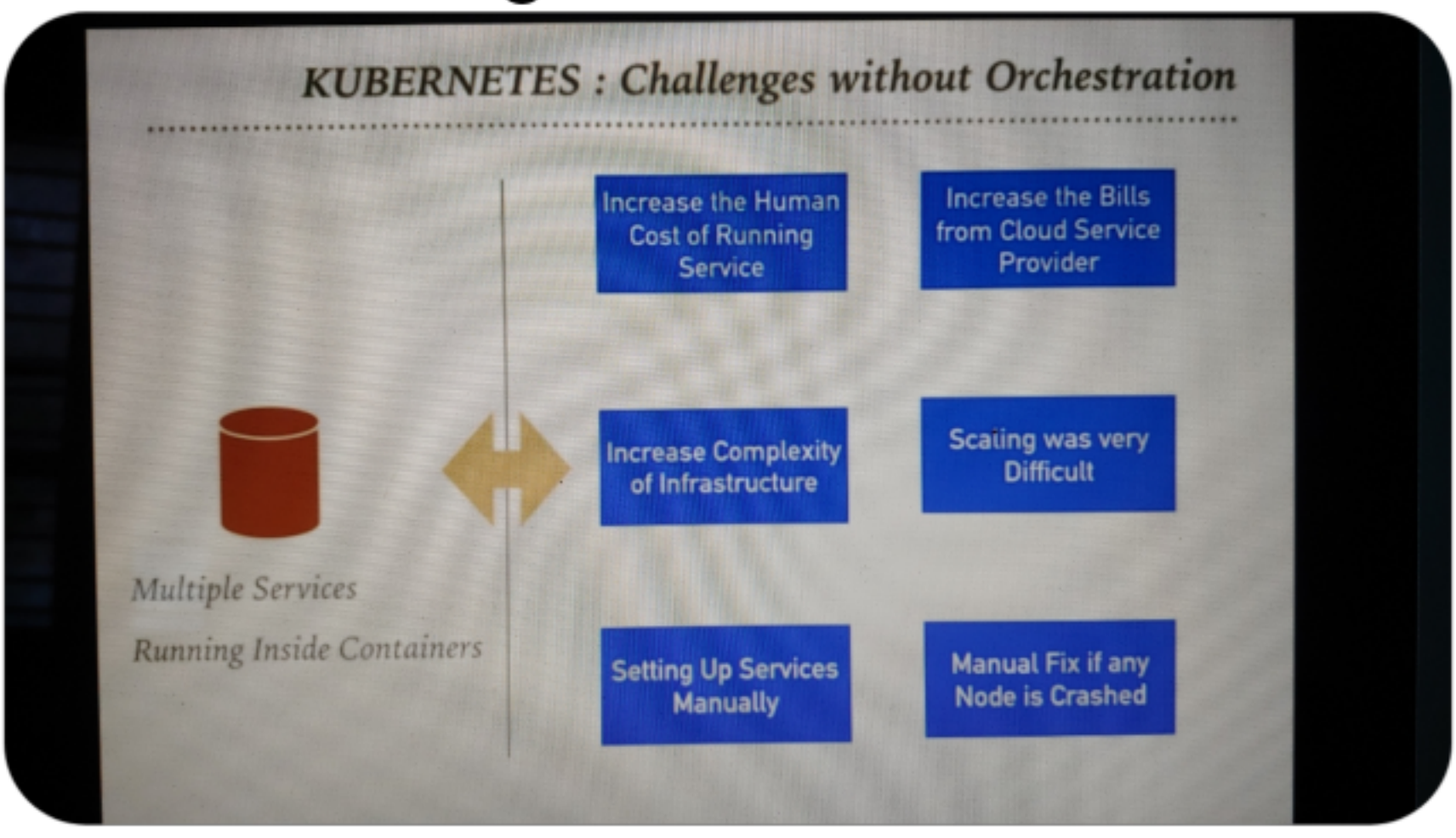
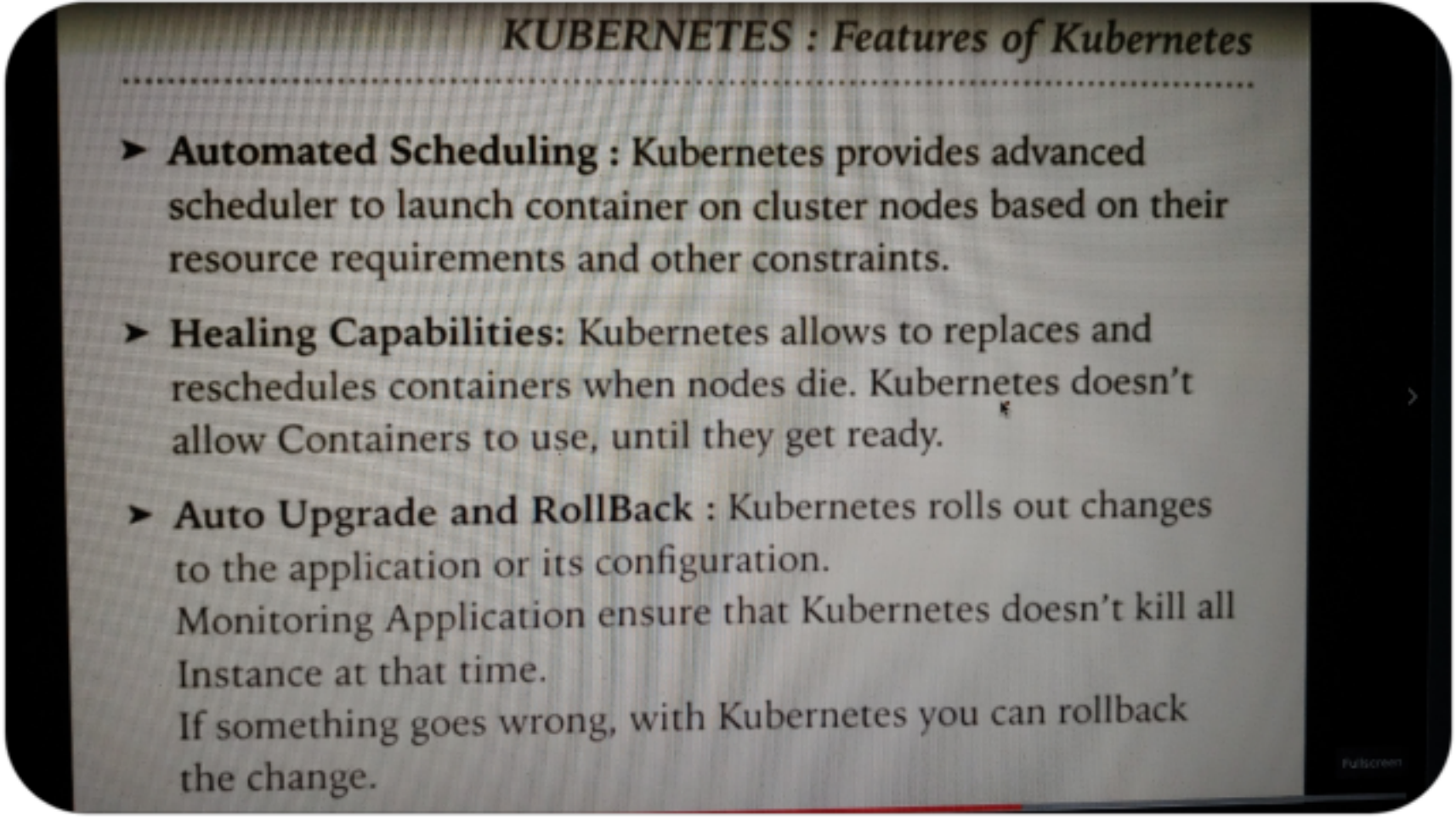
Kubernetes
(Container Orchestration System)

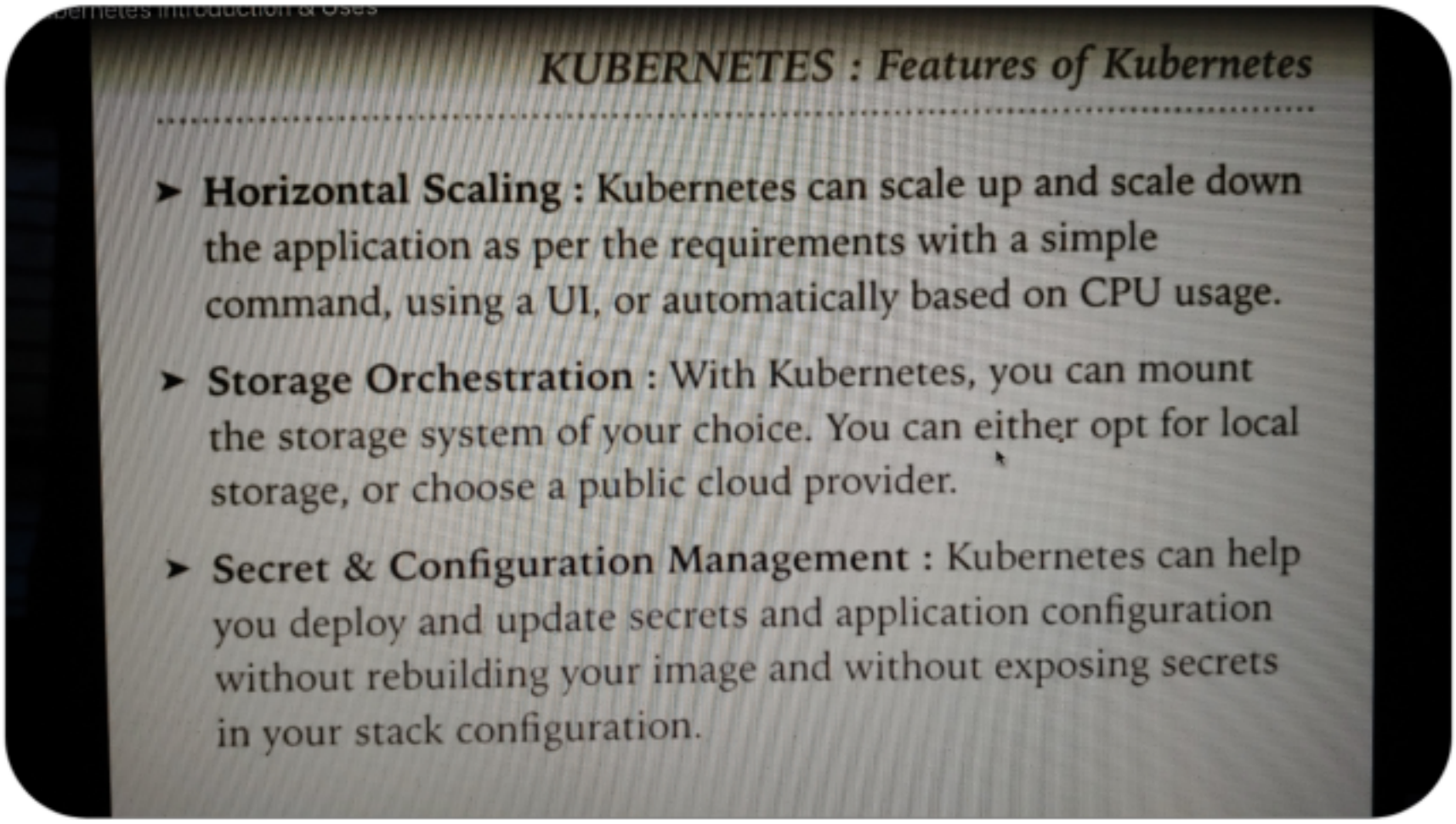
Is an open-source system for automating deployment, scaling, and management of containerized applications

≡ ○ Challenges without Orchistration:



≡ ○ Features of K8S:

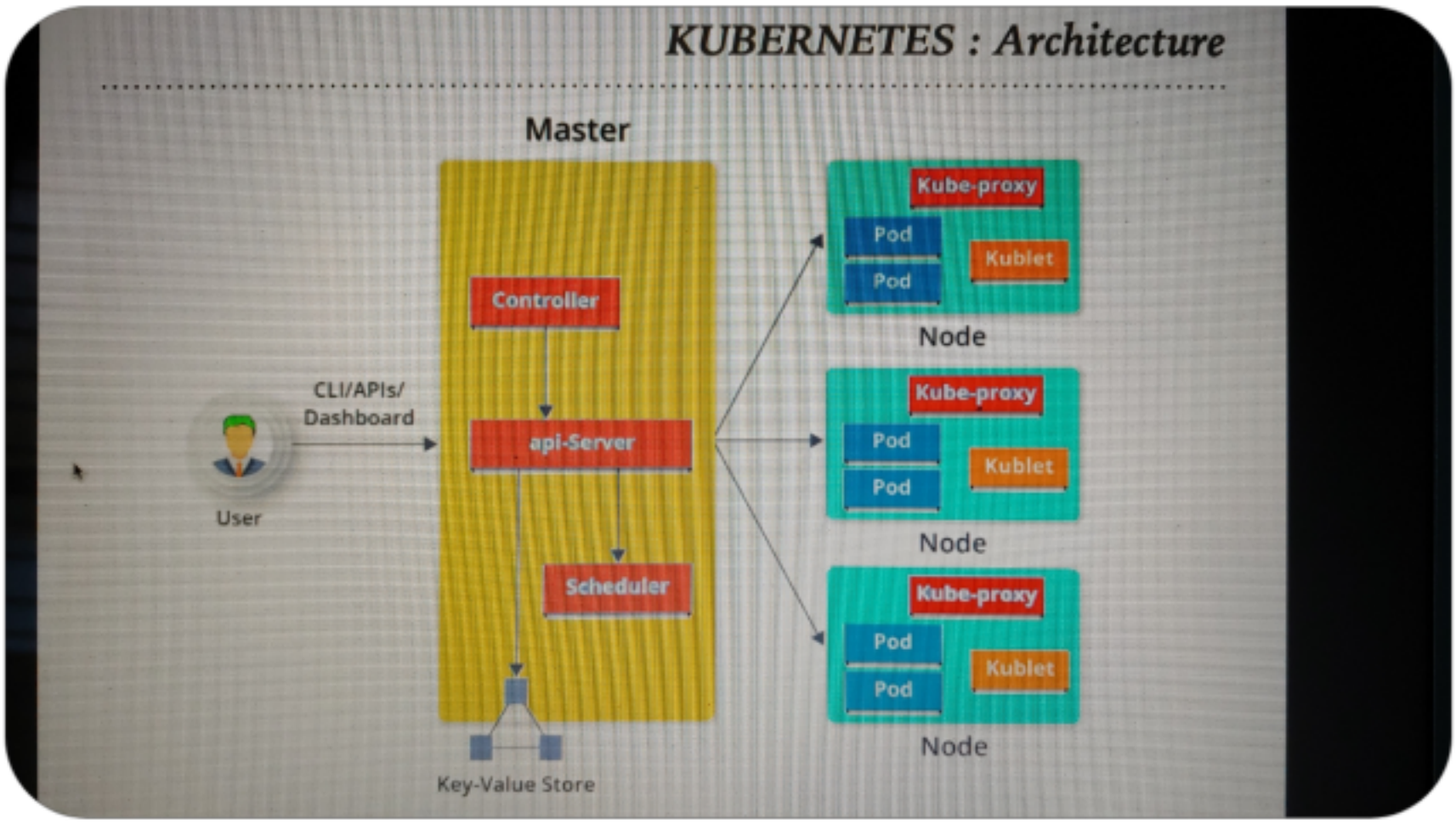




- Also, we can run K8S anywhere like On-Premise(Own DataCenter), Public Cloud or hybrid cloud.

K8S Architecture

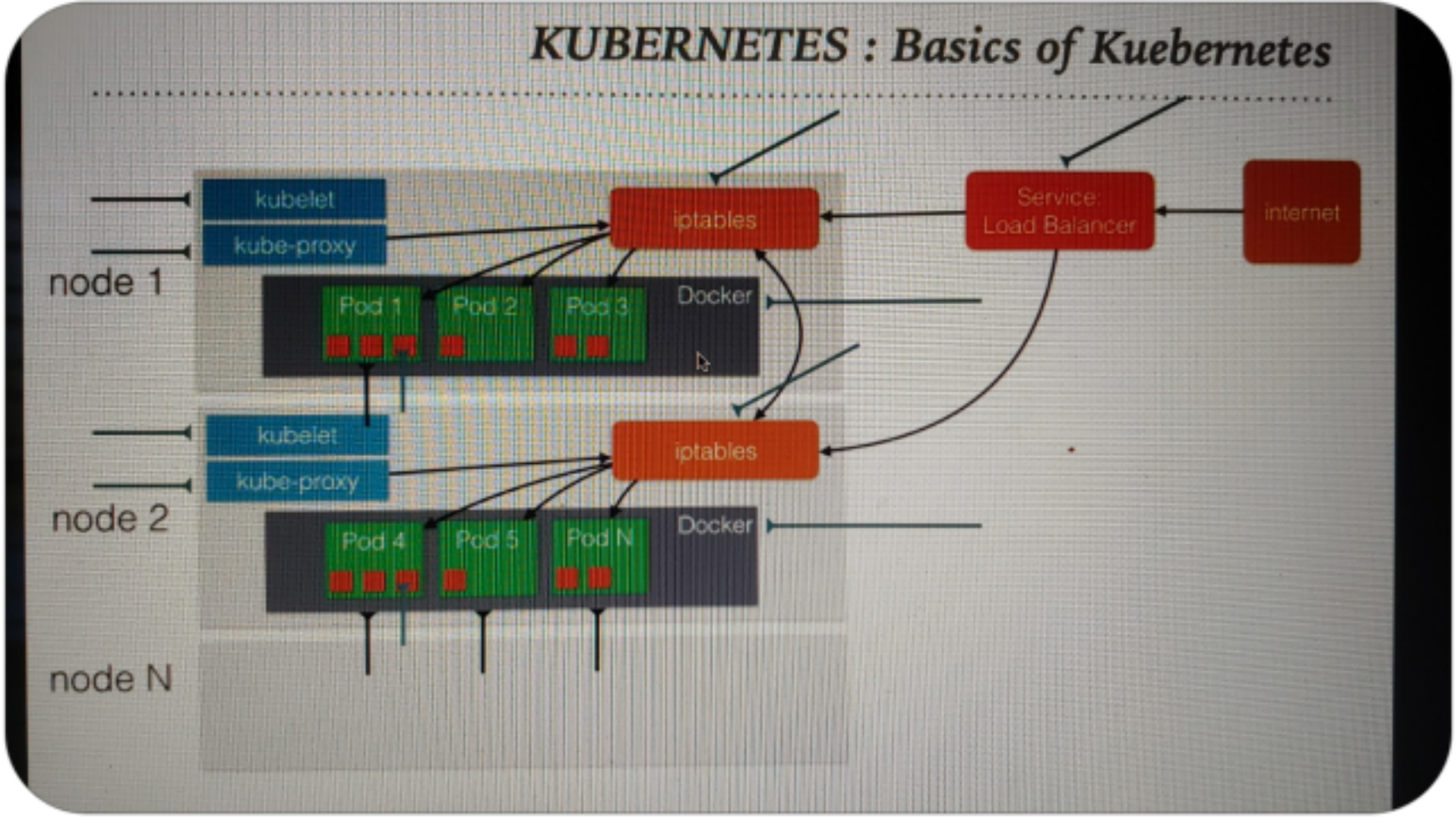
- K8S follow the master-slave(worker) node architecture.



- **Master Node**: Master node is responsible for the management of k8s cluster. More than one master nodes should be there in k8s cluster.
- ETCD/Key-Value store is to store all k8s resources, api server to execute k8s commands by clients, scheduler to schedule a

new pod onto any available node, controller which manages actual and desired state of the cluster.

- By default, k8s runs an api endpoint rest service named 'service/kubernetes', so everytime we enter kubectl command, the command will be posted to this rest api provide by api-server.
- **Worker Node:** A node may be VM or physical machine. Each node contains the services necessary to run pods. Nodes are managed by k8s master. Single master can manage nearly 5000 nodes.
- Pods responsible for running containers, kubelet responsible for monitor the nodes and communicate back to master node, kube-proxy responsible for networking in k8s and container runtime to run containers



Pods in K8S

- These are the smallest, most basic deployable objects in k8s. It can contains one or more containers, such as Docker containers. When a Pod has multiple containers all the containers shares the same cpu, memory and IP address. It is advised to run only one container in each pod so that managing of container will be easy. But in some situation we can run

additional helper container(for example monitoring or logging) in the same pod where main container is running.

- Writing a Pod: Create a new yaml file pod.yml with the below detail

7 June 2021 at 7:44 PM

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
    release: "0"
spec:
  containers:
  - name: webapp
    image: richardchesterwood/k8s-fleetman-webapp-angular:release0
---
apiVersion: v1
kind: Pod
metadata:
  name: webapp-release-0-5
  labels:
    app: webapp
    release: "0-5"
spec:
  containers:
  - name: webapp
    image: richardchesterwood/k8s-fleetman-webapp-angular:release0-5
```

- Running a Pod: type the command 'kubectl apply -f pod.yml' and hit enter. This will create a Pod and run the provided image as a container in it. By default services running inside pods are not visible outside of k8s cluster. This is because pods are short lived and each time they

restart they will be assigned with new IP address.

- States of a Pod:

States of a Pod

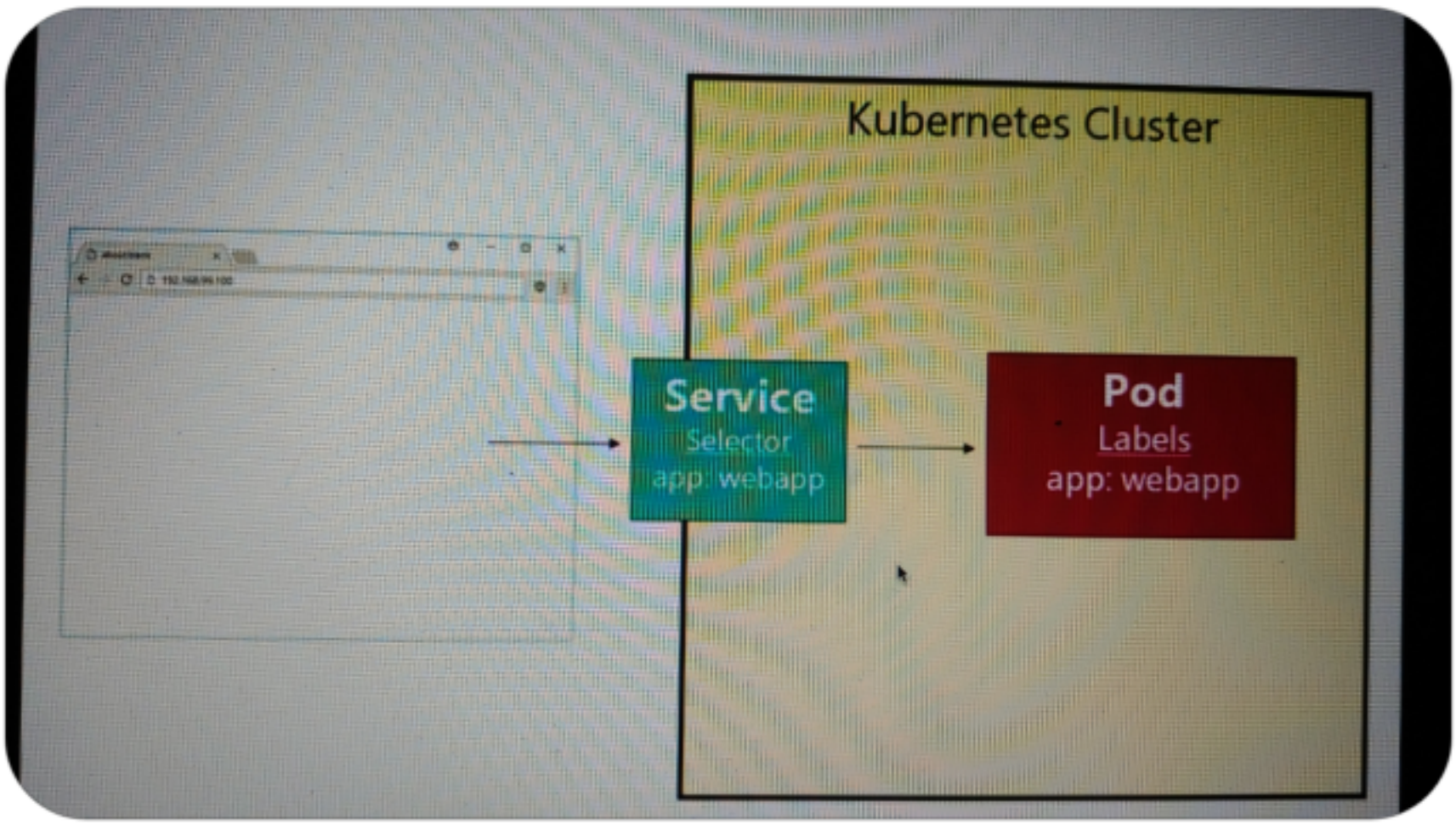
- **Pending:** The pod is accepted by the Kubernetes system but its container(s) is/are not created yet.
- **Running:** The pod is scheduled on a node and all its containers are created and at-least one container is in Running state.
- **Succeeded:** All container(s) in the Pod have exited with status 0 and will not be restarted.
- **Failed:** All container(s) of the Pod have exited and at least one container has returned a non-zero status.
- **CrashLoopBackoff:** The container fails to start and is tried again and again.

Services in K8S

- Services are long running object in k8s. Also, it will be having a static service name and a stable fixed pods. We can attach a service to pod. Now client can call this service which internally calls the container running in a

pod and send the response back to client.

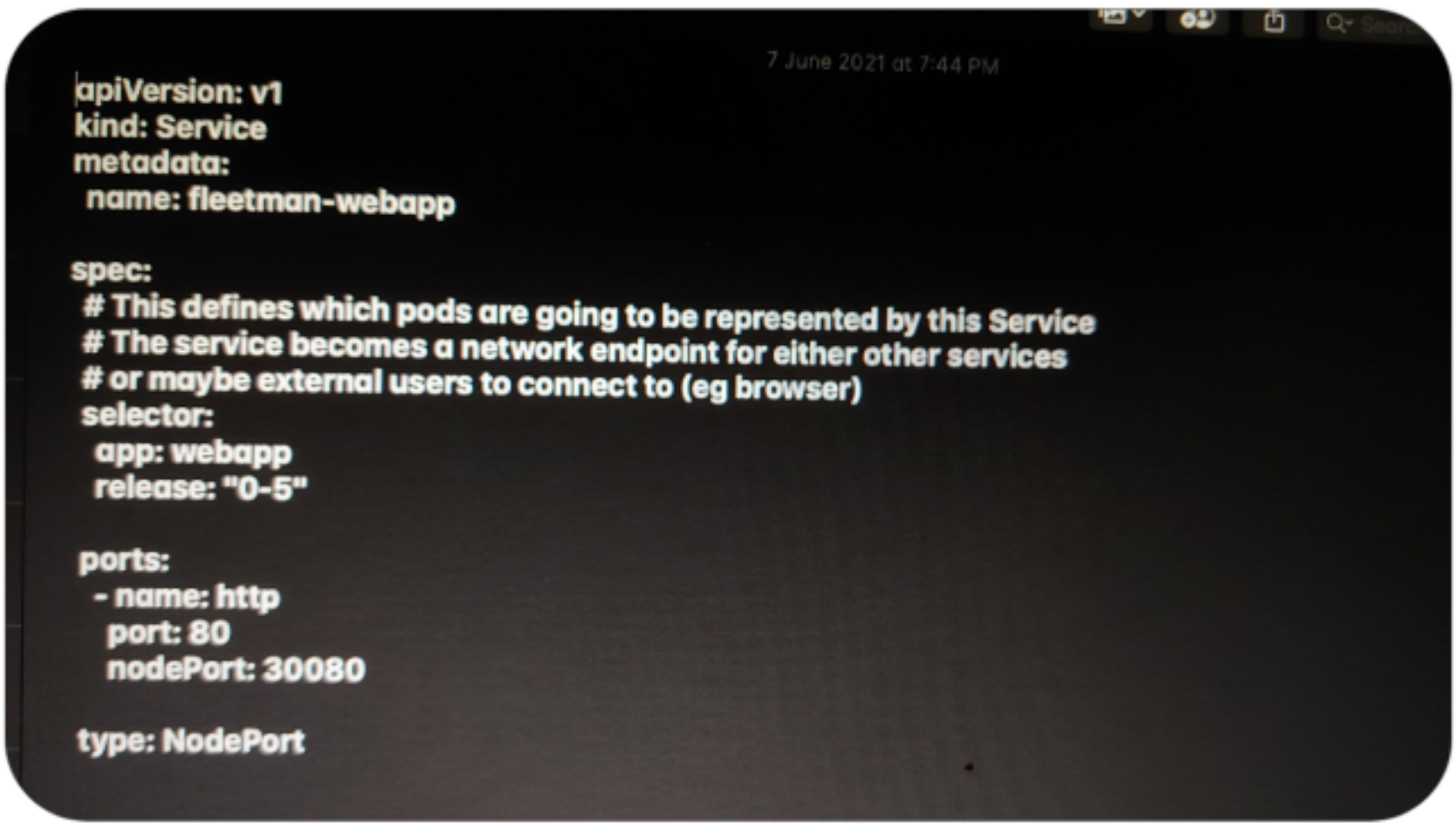
- Labels in Pod: We can set a Label in every Pod which has single/ multiple key value pairs.
- Selector in Service: We define a Selector in every service which will have a key value pair. This key value pair will point to any one of the running pod. In this way we bind a service to a pod through Label and Selector.



- **Types of service**: There are basically 3 types. ClusterIP, NodePort and LoadBalancer.

- ClusterIP: If a service is defined as this type, then it will be internal service and can be accessed only within the k8s cluster mainly by other services.
- NodePort: These type of services will be exposed through the node and can be accessed by any external application along with other internal micro services. As a developer we need to decide on which port we can expose this service. This will be configured in yml file under the ports section with nodePort option. Note that we should choose any port which are more than 30000 for this service of type NodePort.
- LoadBancer: By default k8s local cluster will not support this. Many of the cloud provider provides this option to use. In production we should use this type of service

- Note: NodePort should be used only for development purpose as we need to mention ports above 30000. For production another option is to use Ingress service instead of these above seevices.
- Writing a service: Create a new yml file service.yml and add below details:



- Running a service: Run the kubectl apply command for this seevice.yml file.
- Question: How can we deploy new version of the image with zero downtime. Answer is by

using new label and selector. Create another new pod which runs this nee image version. Now go to service.yml file and change the release selector value from previous 0 to current 0-5. Now service will direct the traffic to new pod automatically. The better way is to use k8s Deployments.

ReplicaSets in K8S

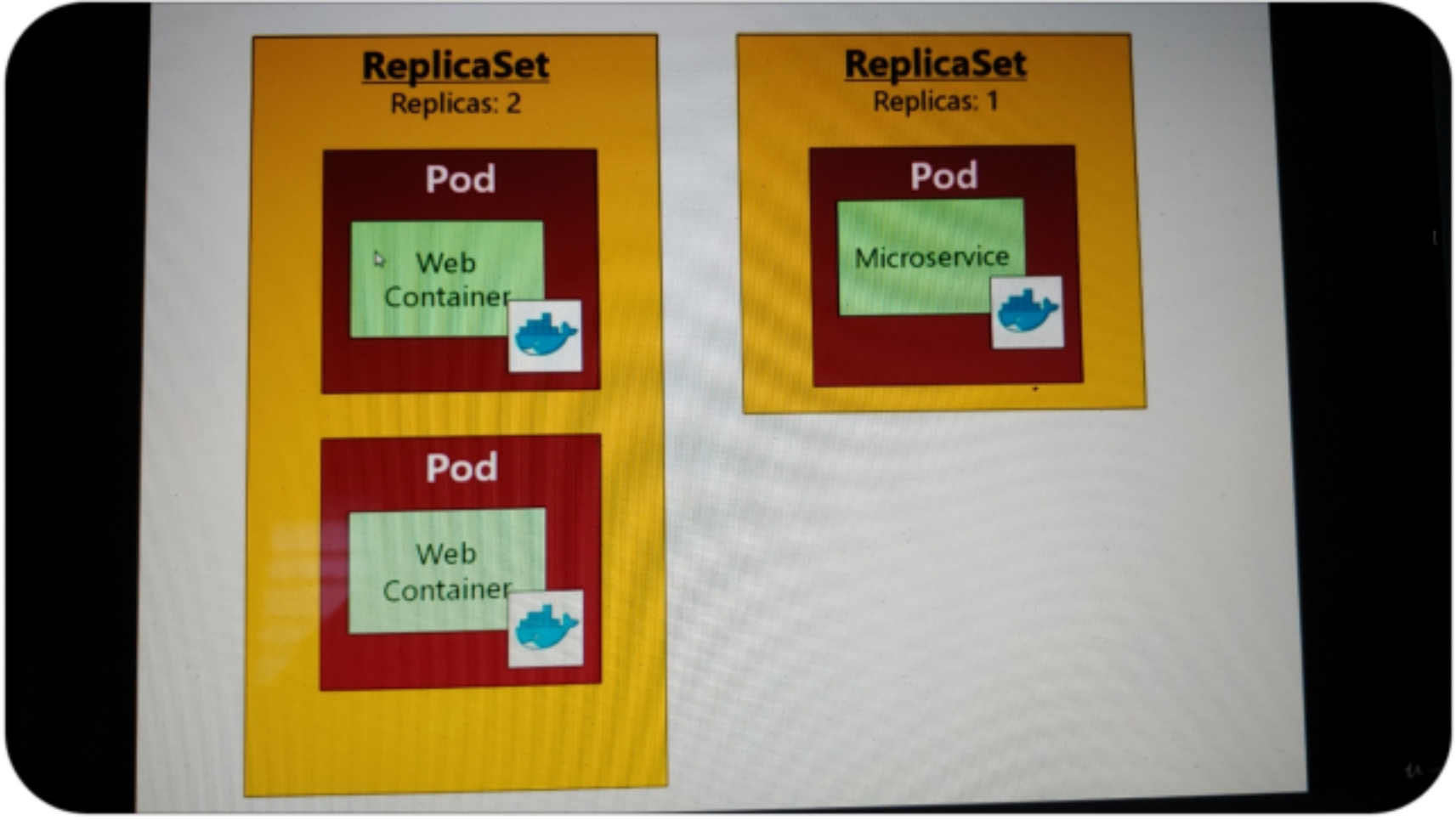
- If we deploy a pod in k8s then its your responsibility for the life cycle management of that pod. If that pod dies then we need to make sure to restart it manually. Now we wrap these pods by a ReplicaSet with some extra configuration, which will inform k8s to maintain specified count of instances of this pod running at any time. So now even though

pod dies, ReplicaSet automatically detect it and spins new pod automatically.

- Two important pieces in ReplicaSet is to mention how many replicas needed and Selector which will bind this ReplicaSet to a Pod.
- Writing a ReplicaSet: Update the previously created pod.yml with below details:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: webapp
spec:
 selector:
   matchLabels:
     app: webapp
 replicas: 2
template: # template for the pods
  metadata:
    labels:
      app: webapp
  spec:
    containers:
    - name: webapp
      image: richardchesterwood/k8s-fleetman-webapp-angular:release0-5
```

- Running ReplicaSet: Run kubectl apply command on pod.yml file which will create this ReplicaSet.



- Note: When we delete a ReplicaSet all the pods of this ReplicaSet also terminates automatically.

Deployments in K8S

- Deployments are similar to ReplicaSet with one more additional feature for automatic rolling update with zero downtime.
- Writing a Deployment: Update the same pod.yml file with below details:

8 June 2021 at 7:18 AM

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
 minReadySeconds: 30
 selector:
   matchLabels:
     app: webapp
 replicas: 2
 template: # template for the pods
  metadata:
    labels:
      app: webapp
  spec:
    containers:
    - name: webapp
      image: richardchesterwood/k8s-fleetman-webapp-angular:release0-5
```

8 June 2021 at 6:37 PM

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: position-simulator
spec:
  selector:
    matchLabels:
      app: position-simulator
  replicas: 1
  template: # template for the pods
    metadata:
      labels:
        app: position-simulator
    spec:
      containers:
      - name: position-simulator
        image: richardchesterwood/k8s-fleetman-position-simulator:release1
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: production-microservice
```

- When we create a Deployment, it automatically creates a ReplicaSet and pods instances associated with this RS.
- Managing Rollouts: We can

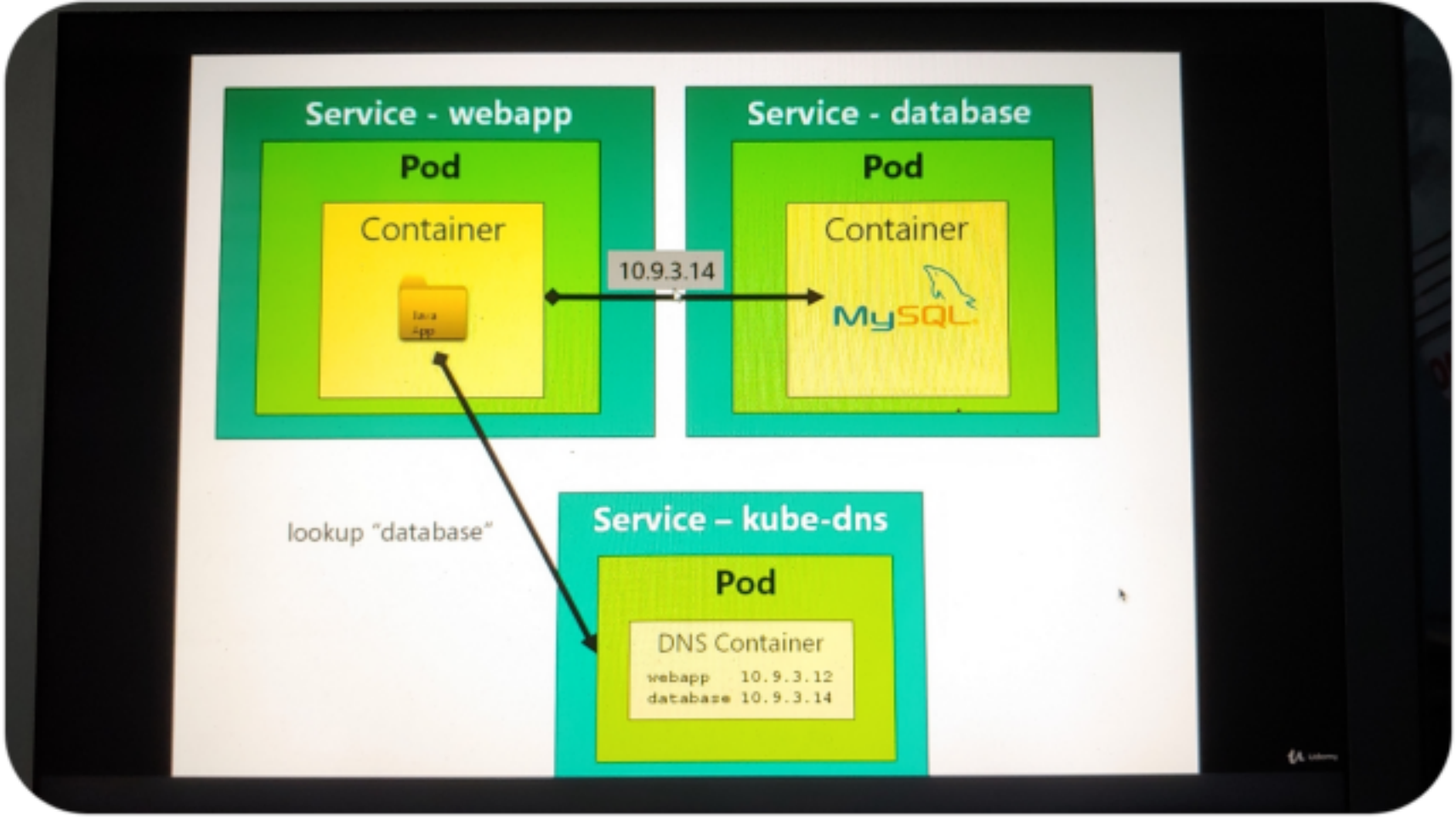
rollout between multiple deployments which has different versions of image running inside a pod. For ex: Rollout from deployment which is running an application of version 2 to its previous version 1, which was deployed previously. K8s can remember last 10 rollout and we can rollout to any revision of it.

Networking in K8S

- If we deploy two containers of different images in a single same pod, then these 2 containers can communicate between each other through localhost. Now if we run 2 different containers in a 2 different pods then accessing through localhost will not work.
- **K8S DNS service**: Solution for the above issue is, k8s maintains its

own private DNS service called kube-dns. Kube-dns is a database containing multiple key-value pair. Key is the name of the k8s services and value is its corresponding IP address. K8S takes the complete responsibility of maintaining this DNS service list up to date always. This service always will be running in the background.

- Now to communicate with a service which is running in another container, we can simply use its name in the current container.



- **K8S Namespace**: Namespace is a way to partitioning resources in k8s into separate areas. When we create a new resources in k8s without providing the namespace details, then these resources will be created in 'default' namespace
- By default k8s creates 2 namespaces, one is kube-public and another one is kube-system and adds many system level resources under them.
- Note: When we create any resources under a custom namespace other than default

one, then while querying these resources through kubectl command we must need to pass additional parameter '-n <custom_namespace_name>.

Persistence in K8S

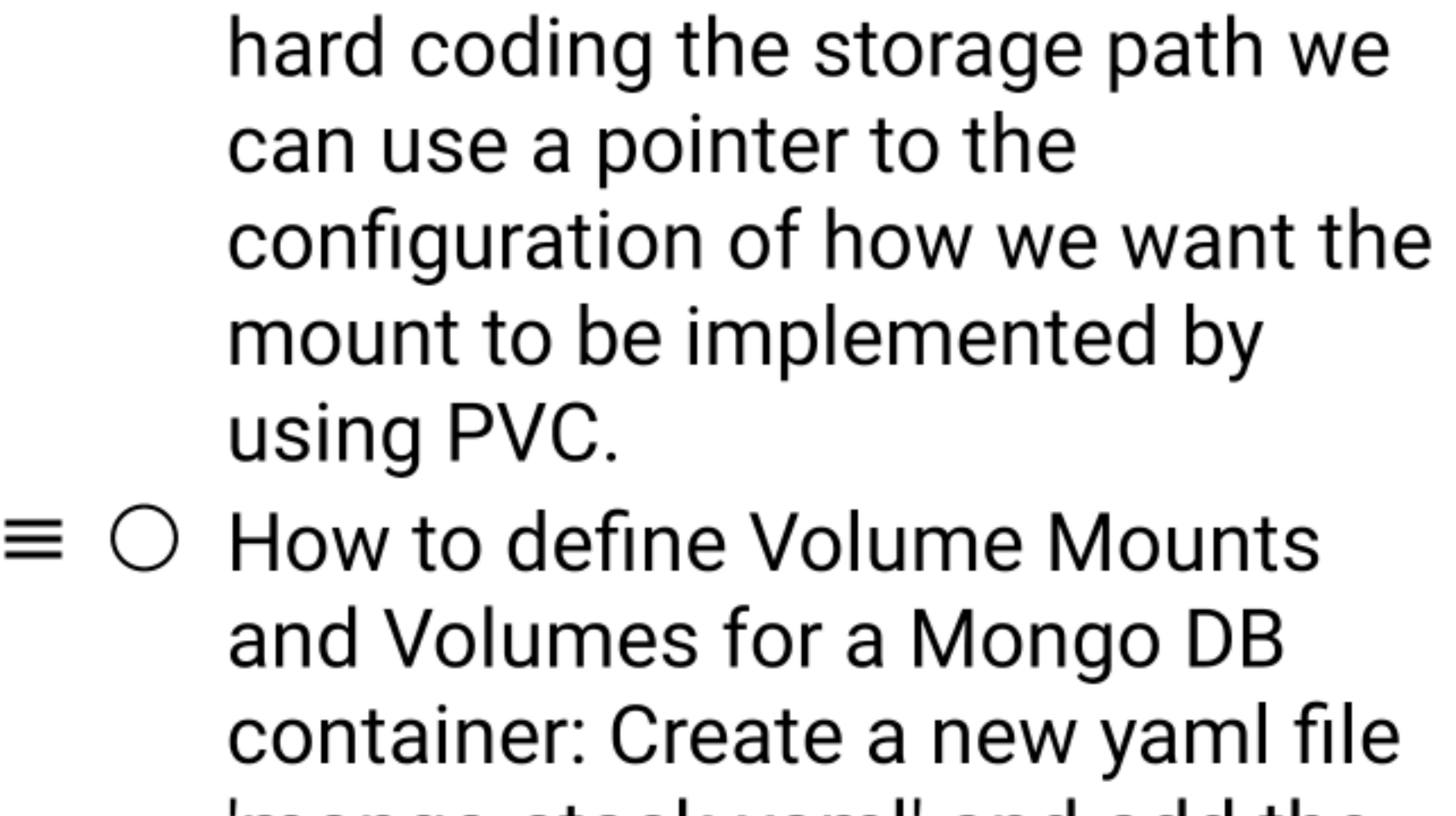
≡ ○ When a pod is destroyed all the data of the pod also will get deleted. For example if we run a Mongo DB inside a k8s pod through a k8s deployment, and connect application to this Mongolia DB from other pod to save some record, by default Mongo will save all those data in a file system inside the running pod. So now if this pod deleted, then all those application data also will be get deleted. Hence we must store those application specific data of Mongo in a

specific directory of host machine for development or in a cloud for production.

- **Volume Mounts**: Docker has a concept called mount which allows mapping a folder inside a container to the folder inside a host machine. We have the same kind of concept in k8s. K8s has a configuration called volumeMounts where we can specify these mappings along with another config field in k8s yaml file named mountPath which is the directory path inside the container where the data will be stored.
- **Volumes:** Through this Volumes configuration we will provide the details of host machine storage path or any cloud provide cluster path to store the application specific Mongo data. Instead of

hard coding the storage path we can use a pointer to the configuration of how we want the mount to be implemented by using PVC.

- How to define Volume Mounts and Volumes for a Mongo DB container: Create a new yaml file 'mongo-stack.yaml' and add the below details for VolumeMounts for the image and Volumes -



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
spec:
  selector:
    matchLabels:
      app: mongodb
  replicas: 1
  template: # template for the pods
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: mongo:3.6.5-jessie
        volumeMounts:
          - name: mongo-persistent-storage
            mountPath: /data/db
      volumes:
        - name: mongo-persistent-storage
          # pointer to the configuration of HOW we want the mount to be implemented
          persistentVolumeClaim:
            claimName: mongo-pvc
---
kind: Service
apiVersion: v1
metadata:
  name: fleetman-mongodb
spec:
  selector:
    app: mongodb
  ports:
    - name: mongoport
      port: 27017
  type: ClusterIP
```

- **PersistenceVolumeClaims(PVC)**: This will be a separate new yaml file which contains configuration on where the data of pods should

be stored through a reference to PV. In PVC we define what do we want. It can be a directory in a host machine or some storage path in cloud. In this PVC file we also define how much space we request on the Persistence Volume.

- **PersistenceVolume(PV)**: In this configuration we define how do we want actual storage to be implemented.
- **Storage Classes and Binding:** It is used to link PV with PVC by adding a new field storageClassName in both PVC and PV definitions.
- How to define PVC and PV required for a Mongo DB container and link both of them together: Create a new k8s yaml file 'storage.yaml' with the below details -

8 June 2021 at 7:34 PM

```
# What do want?
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  storageClassName: mylocalstorage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
# How do we want it implemented
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-storage
spec:
  storageClassName: mylocalstorage
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/some new/directory/structure/"
    type: DirectoryOrCreate
```

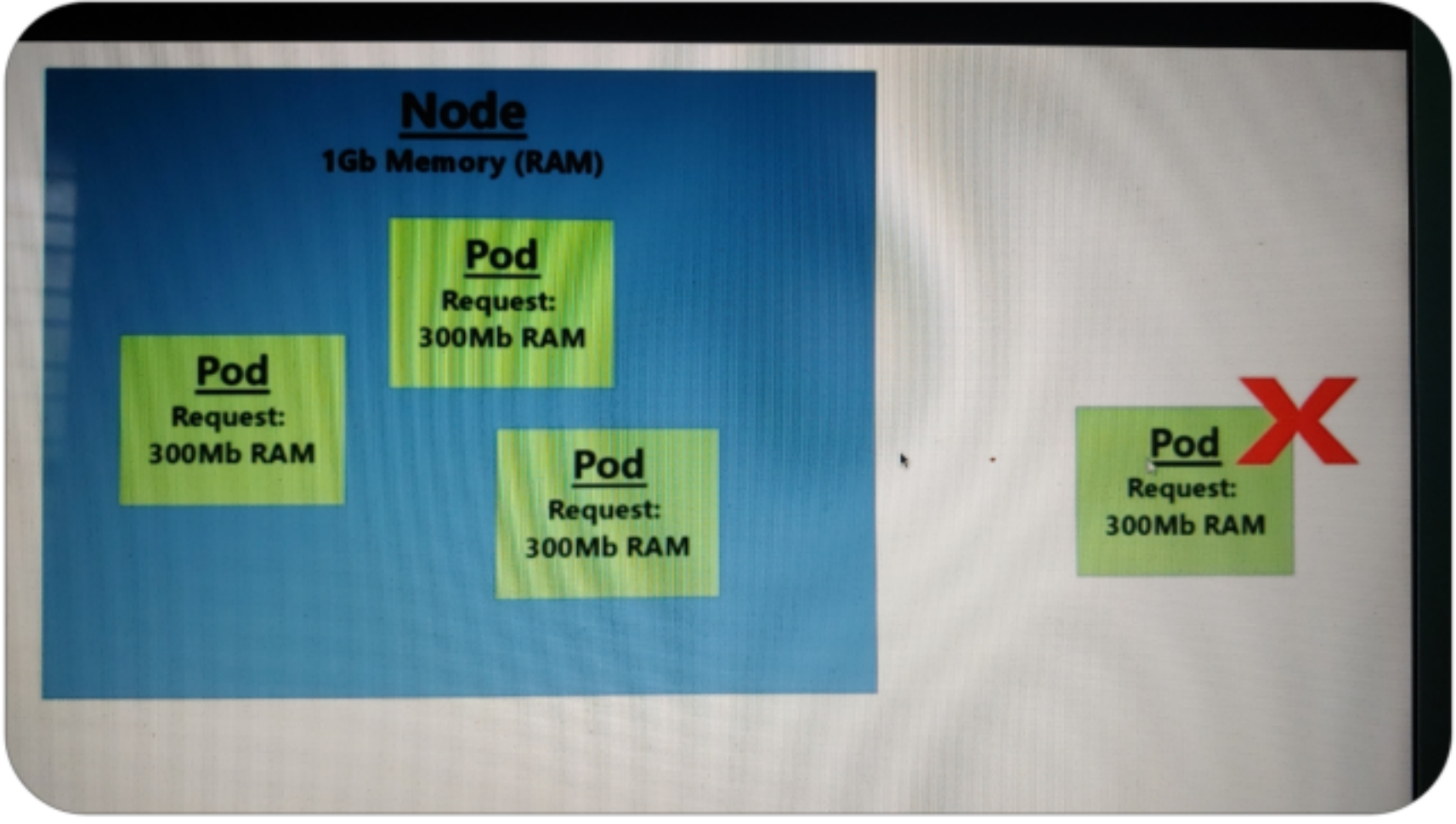
≡ ○ Finally apply both of these new files storage.yaml and mongo-stack.yaml to k8s. This will create all the resources related to k8s persistence amd store the Mongo DB data in the host machine path '/mnt/some new/directory/structure'.

Requests and Limits in K8S

≡ ○ In the deployment.yaml file we can define resource request which can be either a memory request or CPU request. This will

request k8s to provide requested memory and CPU for this pod to run. It is always good to define this request, so that when we try to deploy this to a real k8s cluster, it will allows cluster manager to make sure that there is enough memory on the server on the node to run this pod comfortably.

- **Memory Request**: For example see the below screenshot. The 4th pod can not be deployed in the current Node. So now cluster manager will try to find any other node in the current cluster which has the enough memory to deploy this pod, and if it finds such node then this pod will be deployed there. Else pod will shutdowns with error.



- **CPU Request:** Similar to memory request, we can also request for required CPU for the pod to run.
- Memory and CPU requests are just a hint for the k8s Scheduler to decide, on which running nodes this pods can be deployed so that it can run comfortably. Also, its a guess by developer to determine how much CPU and memory can be needed by a pod with some extra calculation.
- **Memory Limits:** If the actual memory usage of the container at run time exceeds the limit, then

the container will be killed. But, the pod will remain so the container will attempt to restart. So if we set a memory limit for a pod, and if in case pod uses more than this set limit, then automatically this pod will be killed and it attempts to **restart**.

- **CPU Limit:** If the actual CPU usage of the container at run time exceeds the configured CPU limit, then the CPU will be clamped(Means CPU will not allow container to use more than the set CPU limit), but the container will **continue** to run.
- So basically limits will have dramatic impact on the actual running of the containers, and this can be very helpful in handling cases of memory leaks which can be happens in containers. This way we can have

some safety measures to protect overall health of the k8s cluster and if there are any misbehaving containers are found then it will have to pay the price either by having limited CPU or if its gone over memory it will be killed.

- How to write CPU, memory requests and limits:


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: queue
spec:
  selector:
    matchLabels:
      app: queue
  replicas: 1
  template: # template for the pods
    metadata:
      labels:
        app: queue
    spec:
      containers:
      - name: queue
        image: richardchesterwood/k8s-fleetman-queue:release2
        resources:
          requests:
            memory: 300Mi # 1Mi = 1024Kb   1Kb = 1024 Bytes
            cpu: 100m
          limits:
            memory: 500Mi
            cpu: 200m
```

- Note: In Minikube, we can check the CPU and Memory used through metric server which is provided by Minikube by default.

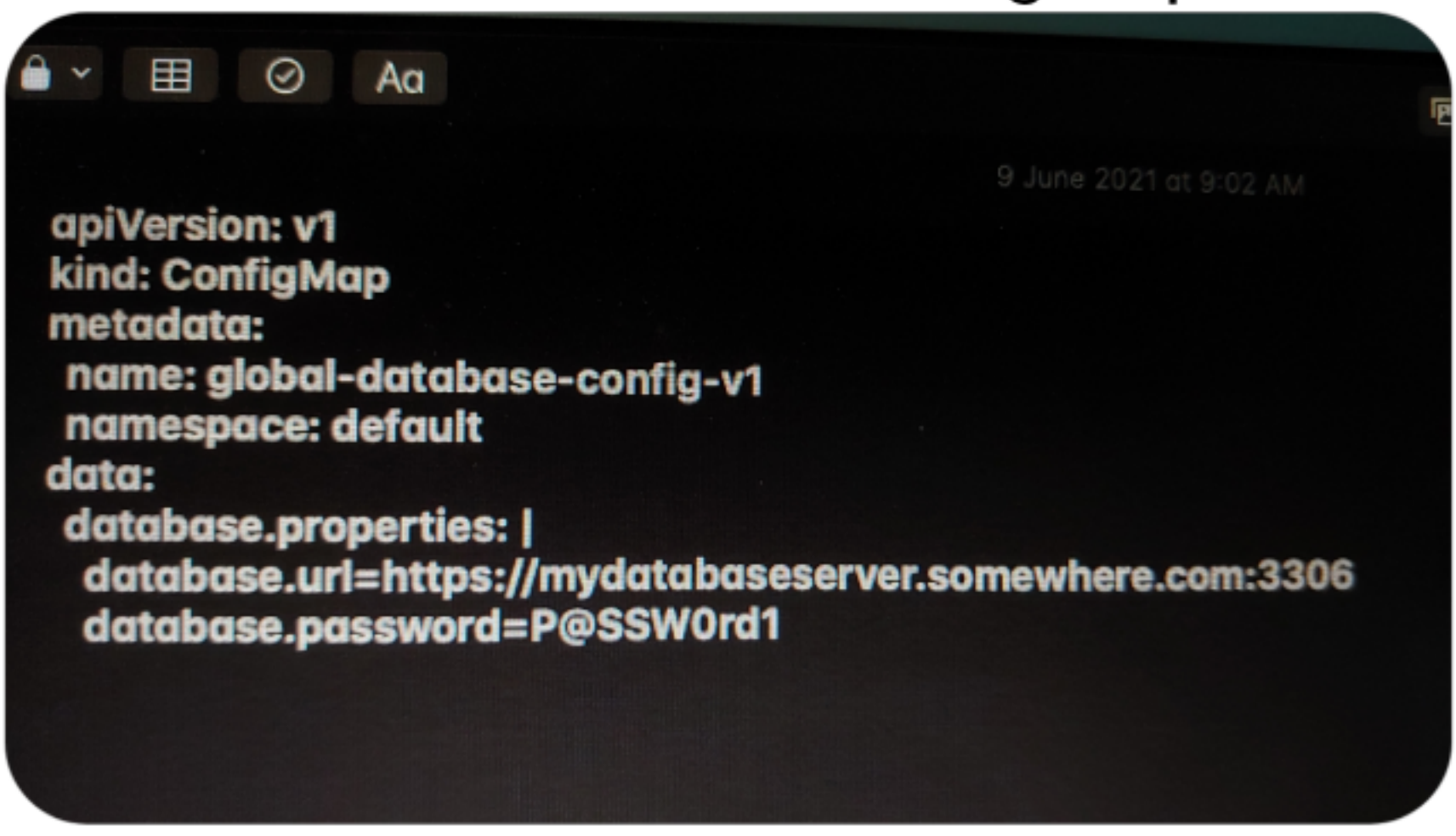
ConfigMaps and Secrets in K8S

- In most of the docker images, they expose some environment variables as decided by developer of that micro service, the value of which we need to pass while running them inside k8s pods, depending on the environment.
- Now such kind of environment variable names and values will be spread across multiple k8s yaml configuration files which will be a bad practice as it contains duplicate codes. It would be better if we could extract such repeated environment variables and store it in k8s.
- For this, we can define ConfigMap and inject the data from it to any pod either by converting the data into environment variables or probably preferred way is to mount the ConfigMap as a

volume and then use it as regular file on images file system.

- **ConfigMaps**: We can create a separate yaml file of kind ConfigMaps and define set of key value pairs. Here key is the environment variable name and value is the actual value of it. Once ConfigMaps are defined, we can use them in any other k8s resource files through a field called configMapKeyRef.

- How to define a ConfigMap:



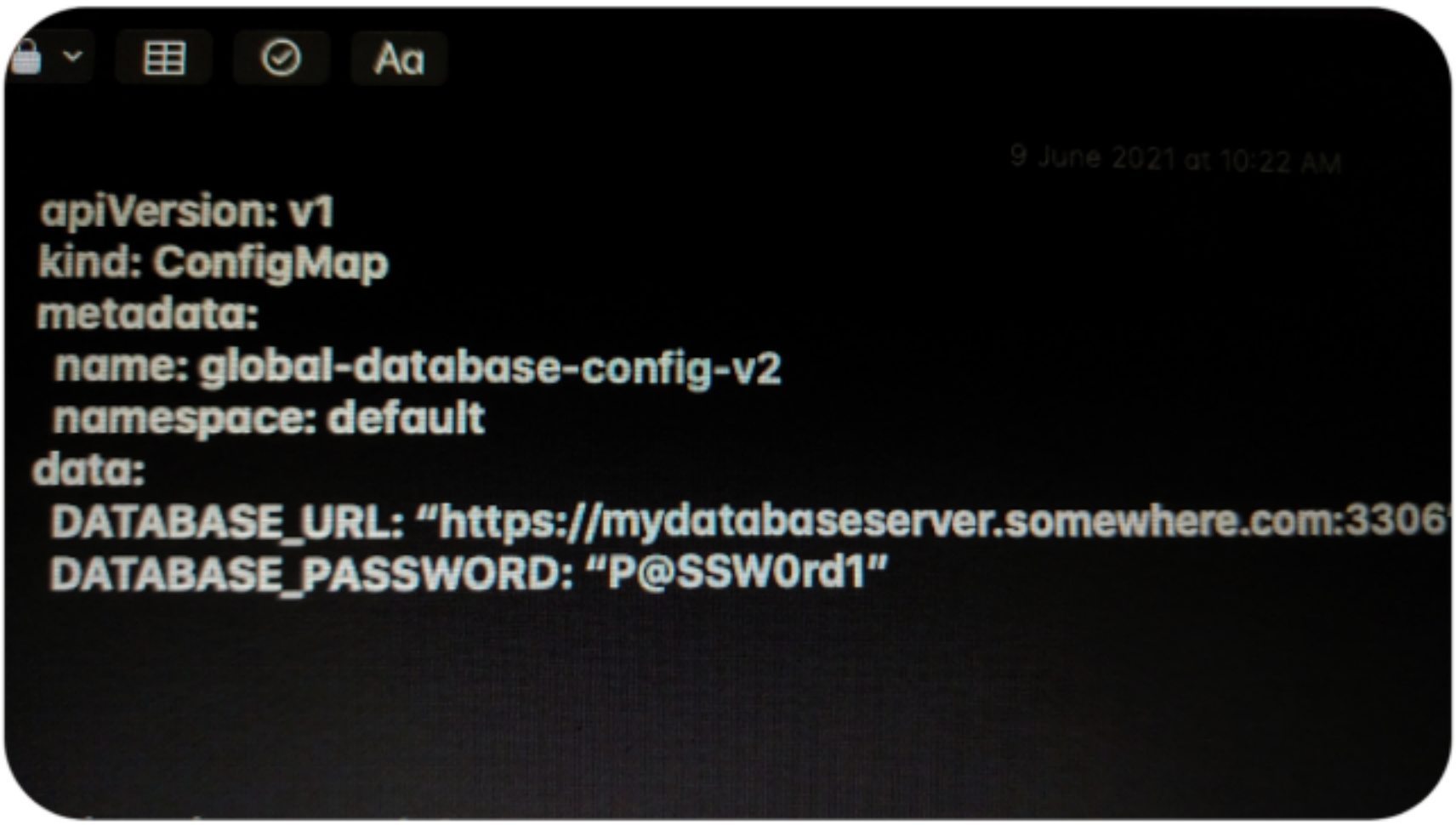
- How to use ConfigMap in Pod:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: position-simulator
spec:
 selector:
  matchLabels:
   app: position-simulator
 replicas: 1
 template: # template for the pods
  metadata:
   labels:
    app: position-simulator
  spec:
   containers:
   - name: position-simulator
    image: richardchesterwood/k8s-fleetman-position-simulator:release2
    env:
    - name: DATABASE_URL
      valueFrom:
       configMapKeyRef:
        name: global-database-config-v1
        key: database.properties.database.url
    - name: DATABASE_PASSWORD
      valueFrom:
       configMapKeyRef:
        name: global-database-config-v1
        key: database.properties.database.password
```

≡ ○ Do changes to a ConfigMap get propagated to a running Pod? - At the time of recording the course the changes done to any key's value will not propagated automatically to running pod. Only available option was to kill the pod so that ReplicaSet will create a new pod automatically where this ConfigMap changes will be available. Another option is to create a new version of ConfigMap and refer to it from pod and apply it. Note: Is this fixed in latest k8s version? Need

to verify it.

- Note: If we use spring cloud kubernetes plugin, that will provide an option to hot reload ConfigMaps without needs for restarting the running pods.
- **Easy way to define ConfigMaps and use it in resources by using EnvFrom option(BETTER THAN PREVIOUS OPTION):** Define all the environment variables in ConfigMap and then load all of them in each resource.

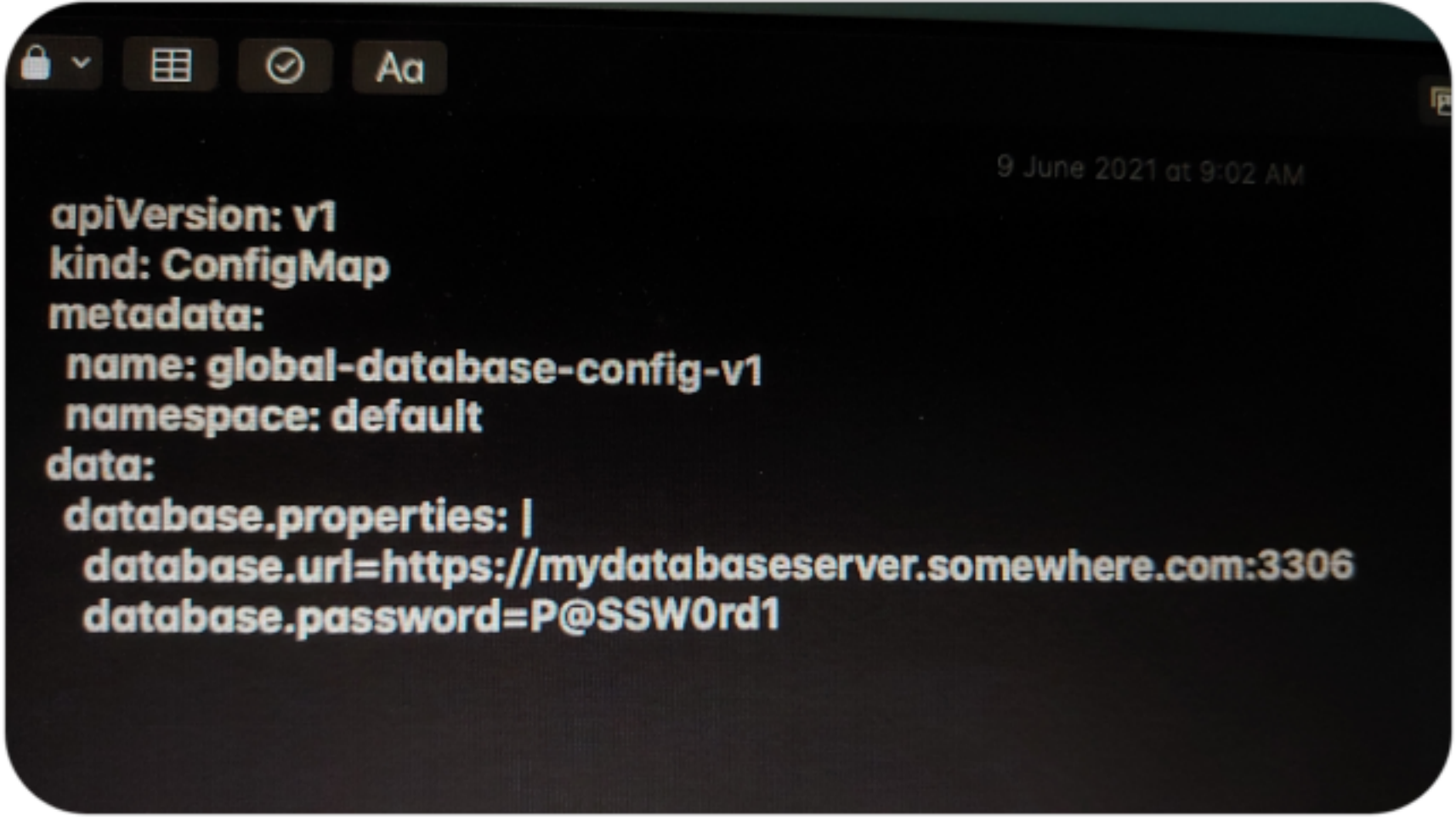


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: position-simulator
spec:
 selector:
  matchLabels:
   app: position-simulator
 replicas: 1
 template: # template for the pods
  metadata:
   labels:
    app: position-simulator
  spec:
   containers:
   - name: position-simulator
    image: richardchesterwood/k8s-fleetman-position-simulator:release2
    envFrom:
    - configMapRef:
        name: global-database-config-v2
```

≡ ○ **Mounting ConfigMaps as Volumes instead of Environment variable values:** If our image does not want to work with environment variables, then we can also inject contents of ConfigMap in the form of a File which is mounted to a directory inside the image through VolumeMounts and Volumes. With this, all of the values in the ConfigMap will be available as Files on the images file system.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: position-simulator
spec:
 selector:
  matchLabels:
   app: position-simulator
 replicas: 1
 template: # template for the pods
  metadata:
   labels:
    app: position-simulator
  spec:
   containers:
   - name: position-simulator
     image: richardchesterwood/k8s-fleetman-position-simulator:release2
     volumeMounts:
     - mountPath: /etc/any/directory/config
       name: database-config-volume
   volumes:
   - name: database-config-volume
     configMap:
       name: global-database-config-v2
```

- For each of the keys defined in the ConfigMap there will be separate files created in the provided path /etc/any/directory/config, which contains the corresponding value of it. Now to create one single file which has all the keys of ConfigMap, instead of creating multiple files for each key, we can define the ConfigMap in the following way:



- Now this will create a single file 'database.properties' in the above mentioned path, which contains all the key value pairs as mentioned in the above ConfigMap.
- **Secrets:** This k8s objects let you store and manage sensitive information like passwords, OAuth tokens and ssh keys.
- **Creating Secrets:** To define such secrets we can define a new yaml file of kind Secret with multiple keys and base64 encoded values under the data section like:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-credentials
data:
  accessKey: YmFyCg==
```

- Now if we dont want to keep base64 encoded value, then we can use stringData field to directly enter the values so that k8s will automatically convert them to base64 encoded format and uses it

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-credentials
stringData:
  accessKey: "12345678790"
  secretKey: "SECRET1234"
```

- **Using Secrets:** By default secrets are not that secured. We can use the command 'kubectl get secret <secret_name> -o yaml' to get the yaml format of the secret definitions. This will provide us the encoded format of all the secret values which we can decode easily and find the actual secret values. Through secrets we can add a small layer pf protection. Its a sort of first line of protection against system administration mistakes.

Replication and Scaling in K8S

- **Replication**: Replication is the process of creating multiple pods which runs a container from same image. So if there are more traffic on a pod, we replicate this pod so that traffic will be distributed and served. We need to design micro services in such a way that when multiple instances of the same services are created through Replication, it should not cause any side effect like duplicate records, corruption of data etc.
- Horizontal auto scaling of a cluster means adding additional nodes to it to serve the request. And vertical auto scaling of a cluster means making existing nodes more powerful. Horizontal Pod auto scaling means adding

additional pods through pod replication. Usually we can Auto scale either deployment or replica sets depending on requirement like CPU utilization, memory utilization, number of request per second, heap size limit etc.

- **Horizontal Pod Auto-scaler(Hpa):** In k8s we can define a new object called Horizontal Pod Auto Scaler which is used for implementing Horizontal Pod Auto Scaling. In Hpa, we can configure some rules, such that if CPU utilization of a pod exceeds 50% of requested CPU, then go and modify the deployment to add other maximum of 5 more pods. In the same way through Hpa we can define other rules such that if maximum utilization of CPU by a pod reduces to less than 50%, then we can reduces number of

pods automatically. For **each** of the deployment we need to define and create such Hpa object.

- We can configure such Hpa through a command 'kubectl autoscale deployment api-gateway --cpu-percent 200 --min 1 --max 4'. This command will create a new Hpa object for the mentioned deployment name. Here cpu-percent tells if the current CPU usage goes beyond 200% of requested CPU. Min means minimum number of allowed replicas and max means maximum number of replicas.
- Now if we stop and restart the cluster these Hpa objects will be get deleted. We can save this Hpa object definition to a yaml file through the command 'kubectl get Hpa api-gateway -o yaml', and later we can apply this yaml file to

recreate it.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: api-gateway
 namespace: default
spec:
 maxReplicas: 4
 minReplicas: 1
 scaleTargetRef:
  apiVersion: apps/v1beta1
  kind: Deployment
  name: api-gateway
 targetCPUUtilizationPercentage: 400
```

9 June 2021 at 2:12 PM

- **Cluster AutosScaler(CA):** It is used in k8s to scale cluster I.e number of nodes dynamically. CA watches the Pod continuously and if it finds that a pod cannot be scheduled, then based on the Pod condition it chooses to scale up. The CA requests a newly provisioned node if the cluster or node poll has not reached the user defined maximum node count.

Readiness and Liveness Probes in K8S

- If we configured horizontal pod auto scaling, if the CPU usage on a particular running pod exceeds a particular value, then horizontal pod auto scale will automatically increases the number of replicas in the deployment, therefore we are going to have some new pods created in our cluster.
- Lets say there are 2 new pods are created which will internally starts 2 new containers running some services. Now as soon as the containers inside these pods are successfully running then k8s considers these pods are in service and ready to accept traffic. Now here is the problem. The software inside these containers although it is running, it might not yet to ready to service the request. If in case any

request comes in, it will go timeout and throws error to user.

- **Readiness Probe:** We can configure some readiness probe in k8s configuration file for each pod, so that k8s will send some http request to the mentioned http endpoint of this container. And when it gets successful value back from container, only then it considers this container as ready to serve requests and the traffics will be routed to it. If readiness check fails pods will not be restarted but pod IP address will be removed from services.
- **Liveness Probe:** This also a http request similar to readiness probe, but this will continuous to run for the duration of your pods lifetime. For every defined time intervals, the mentioned service

health will be called. And if for any reason if this http call fails then k8s mark that pod as failed and restarts the container.

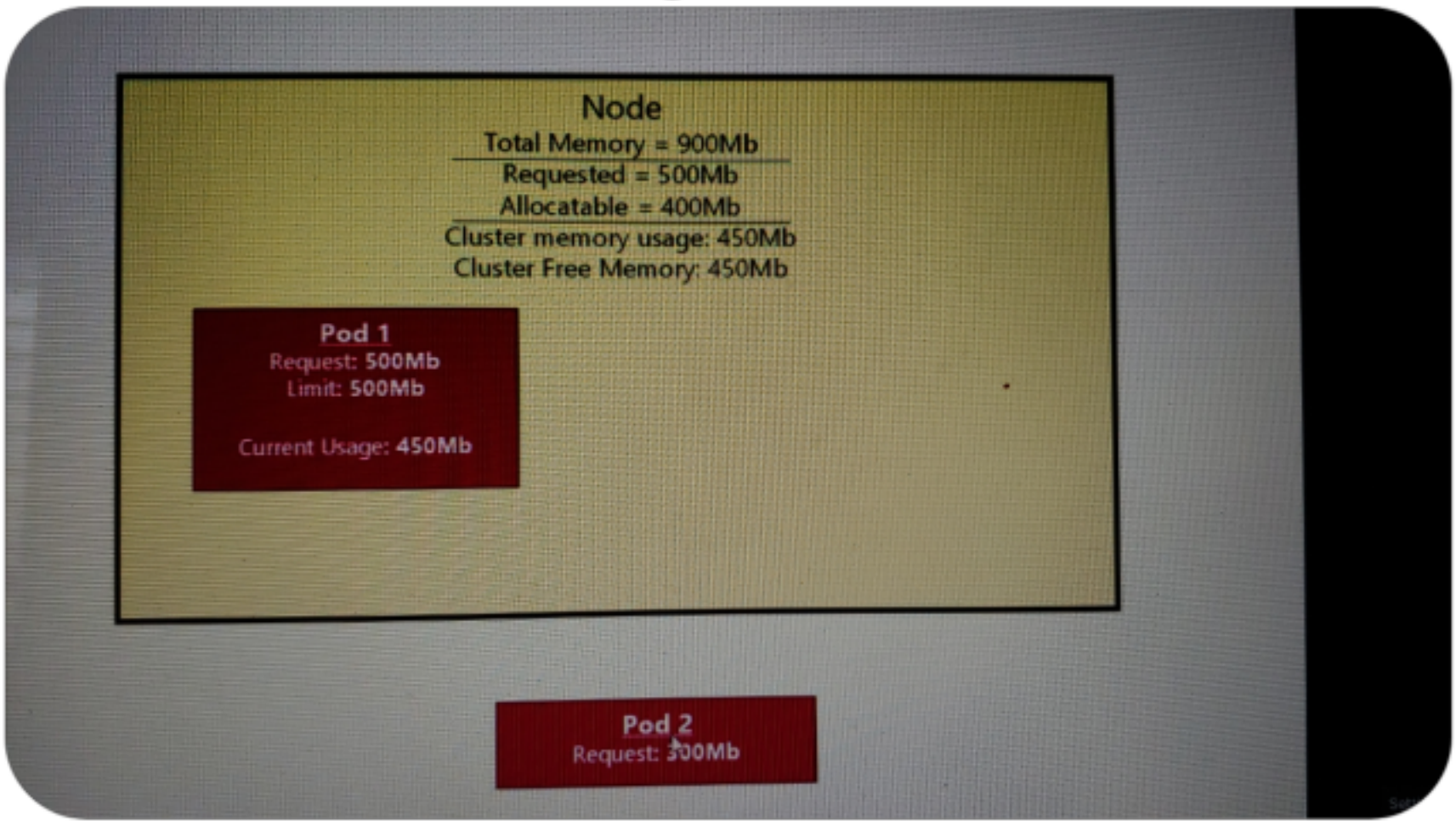
```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: position-simulator
spec:
 selector:
  matchLabels:
   app: position-simulator
 replicas: 1
 template: # template for the pods
  metadata:
   labels:
    app: position-simulator
  spec:
   containers:
   - name: position-simulator
     image: richardchesterwood/k8s-fleetman-position-simulator:release2
     readinessProbe:
       httpGet:
         path: /
         port: 8080
     livenessProbe:
       httpGet:
         path: /
         port: 8080
```

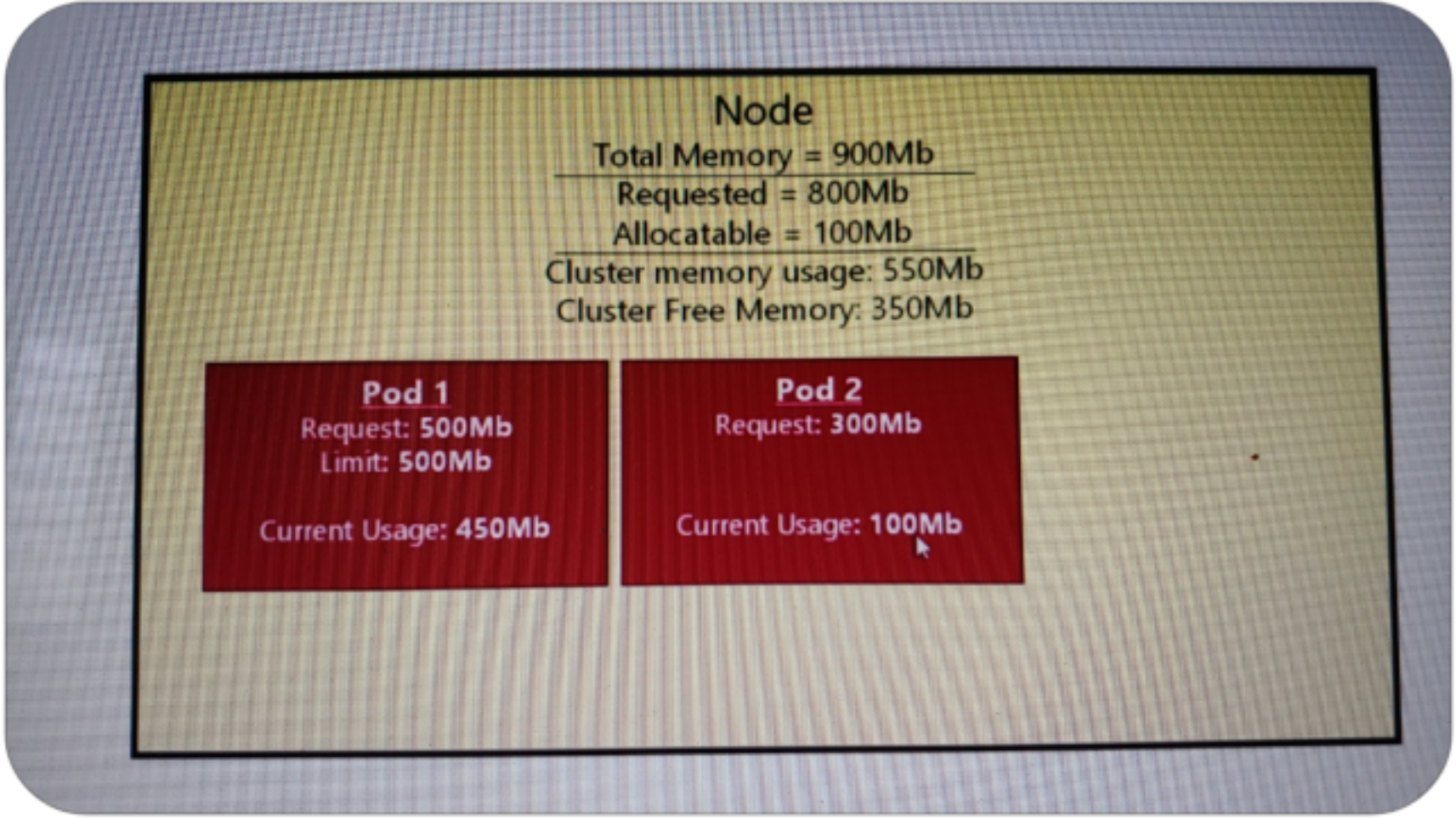
≡ ○ These two configurations will help in not having any downtime when pods restarts. Old pods will keep running until new pod's readiness probe is success.

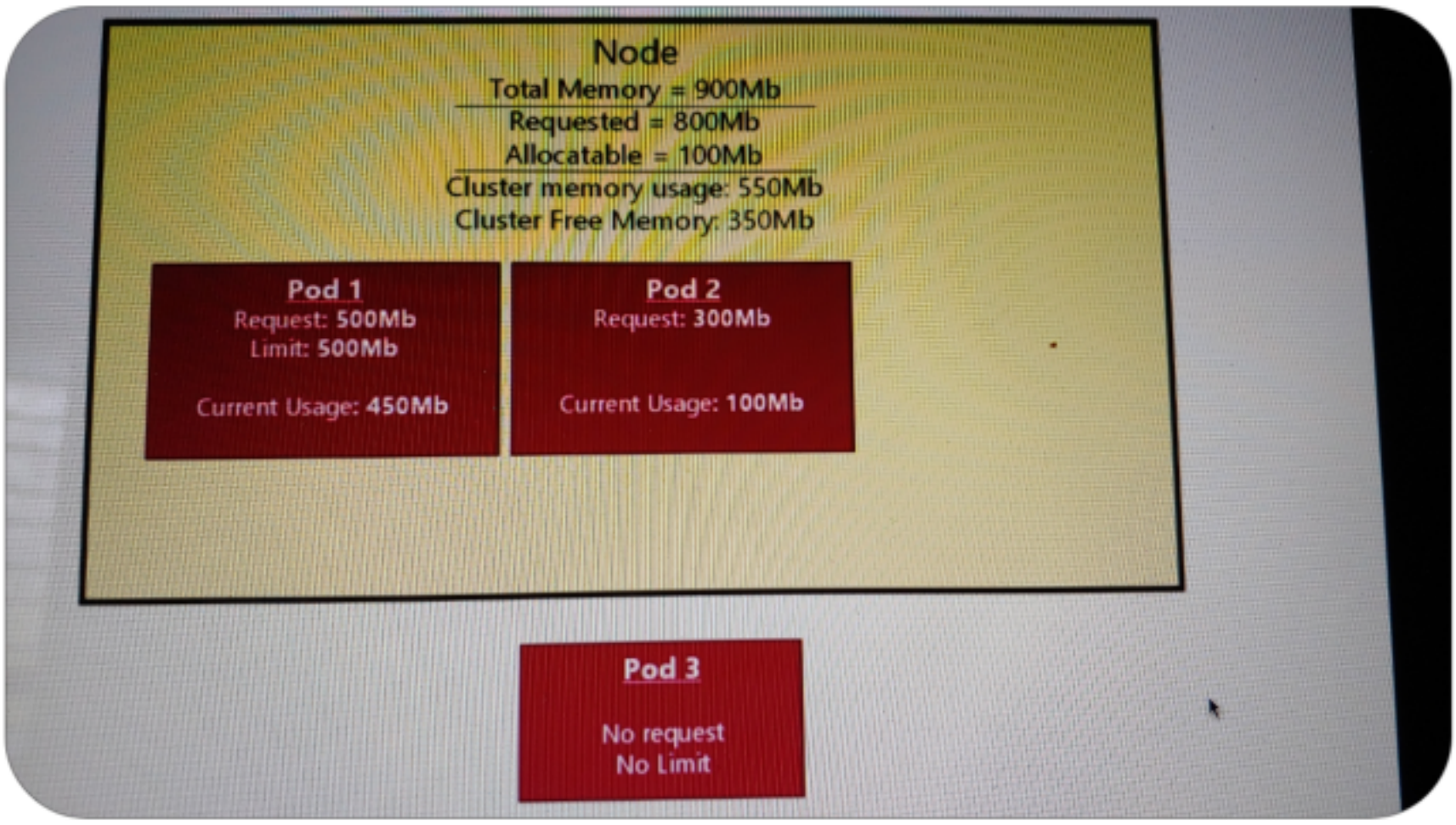
Quality Of Service and Eviction

≡ ○ **Scheduler in K8S:** Job of the scheduler is to decide on which node a particular pod should be

deployed. Scheduler is responsible for ensuring any one node is not overloaded which will be a disaster if that happens. For that sometime scheduler has to evict some pods. Scheduler does the following calculations:

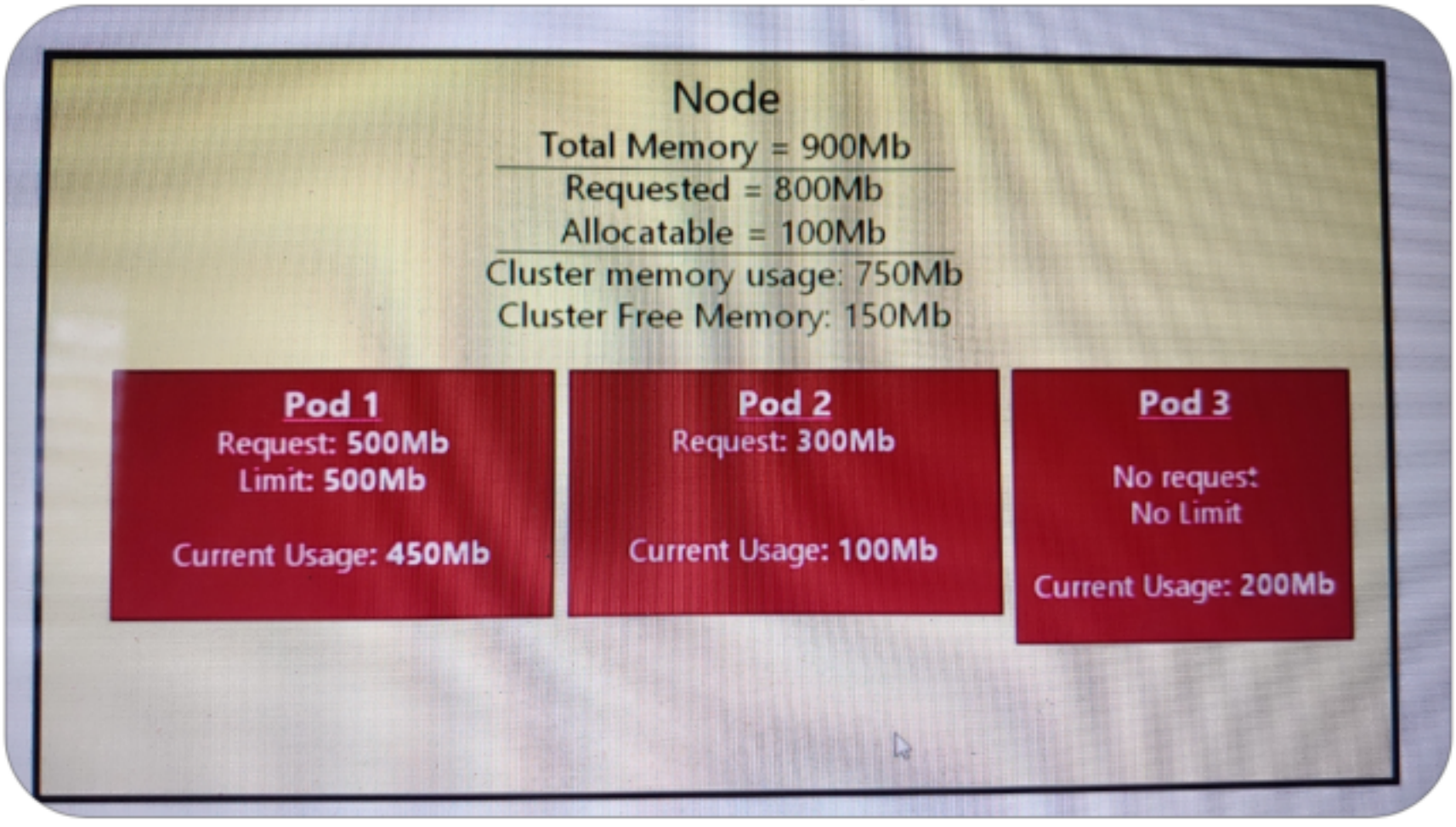






≡ ○ Now when Pod 3 comes, scheduler does not know how much memory is needed for this pod. So its going to deploy it anyway. Now this pod uses

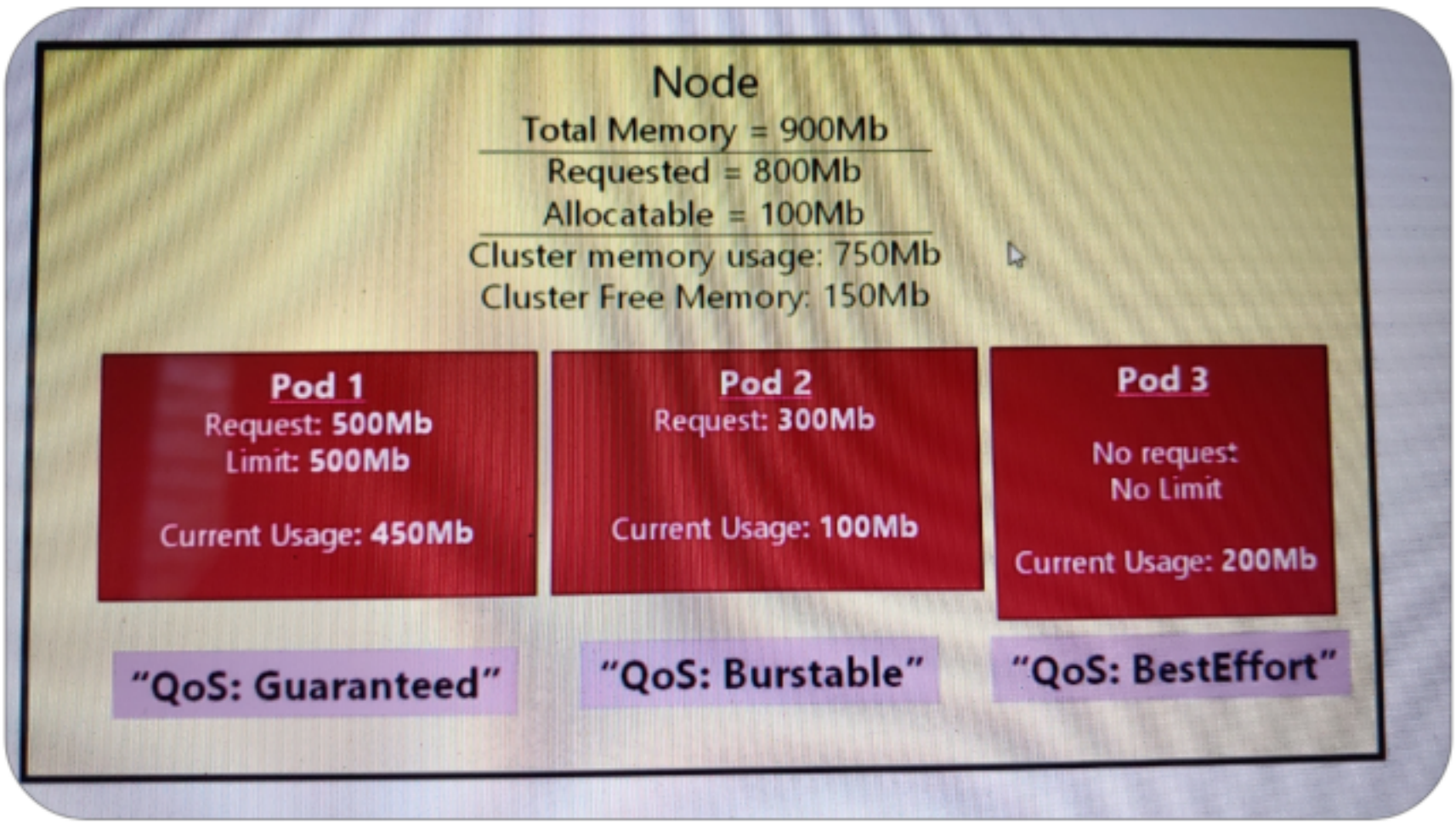
200Mb of memory while running.



- Now in this node we have left with 150Mb of memory and memory usage of any pod can go high at any time.
- In case of Pod 1 developer has defined both request and limit. In case of Pod 2 developer has mentioned only request. And in case of Pod 3 both request and limit are not defined and we have no clue on the same
- Now, Scheduler will uses QoS labels which exists on these pods to decide which pods to evict in

case of nodes are going out of resources.

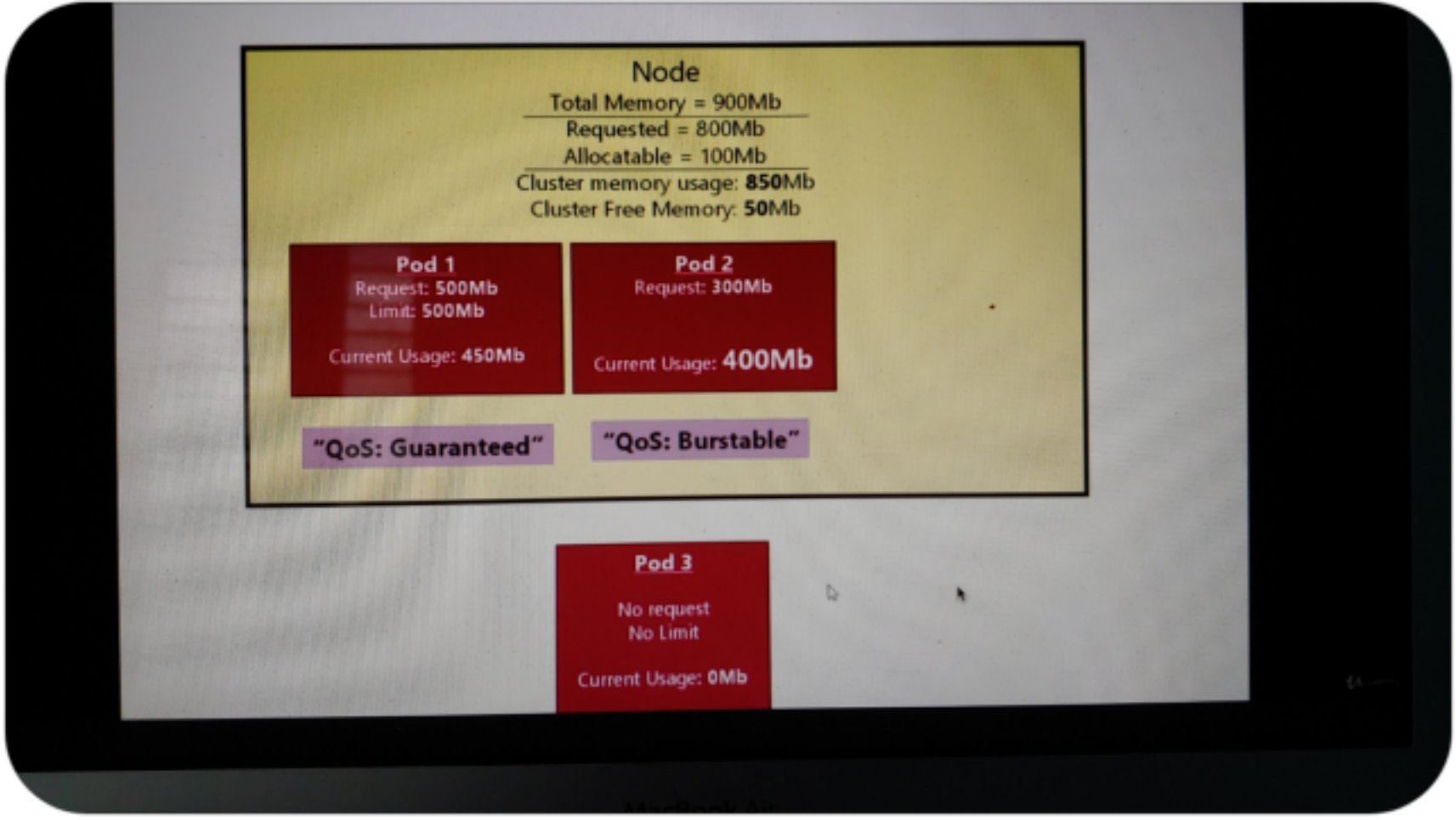
- **Quality Of Service(QoS) labels:** When pods are deployed by a scheduler, every pods will be assigned by a kind of QoS class labels depending on its configuration as explained before There are 3 QoS labels as below:



- **Evictions:** Eviction is the process of deleting a running pod by a scheduler when a node is running out of resources. Scheduler will uses the QoS labels which are exists on every Pod, to decide

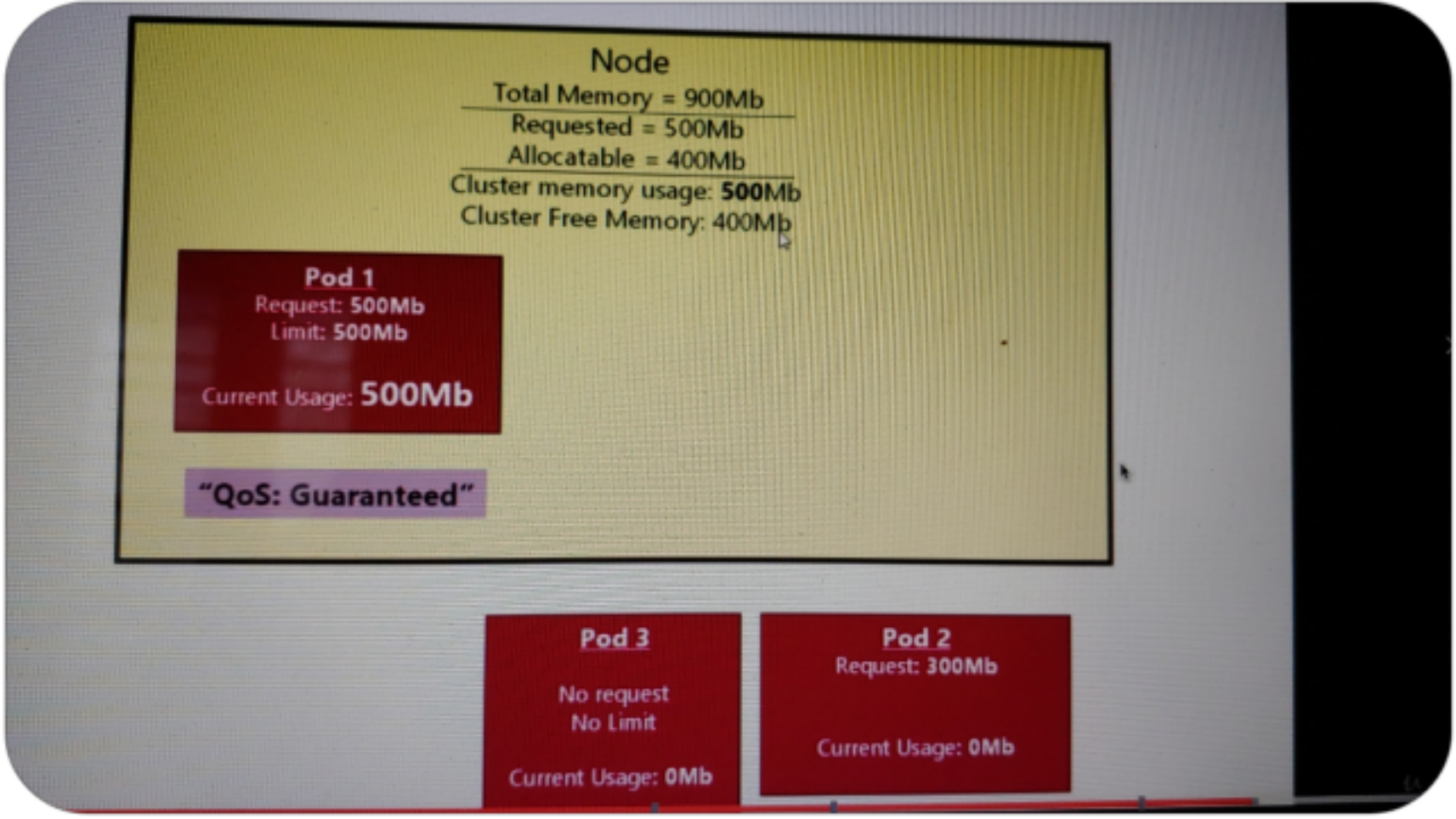
which pod to evict first.

- For the Pod 1 since we defined limit, if in case this pod goes over the limit then it will evicted immediately.
- For the Pod 2, if in case its current usage goes up such that there are no more free memory exists in a node, then scheduler has to evict one or more pods. In such cases, scheduler will first try to evict pods which has QoS label as BestEffort (Here its Pod 3). If still shortage of resource, only then it will evict pods with label Brustable (Here it is Pod 2).

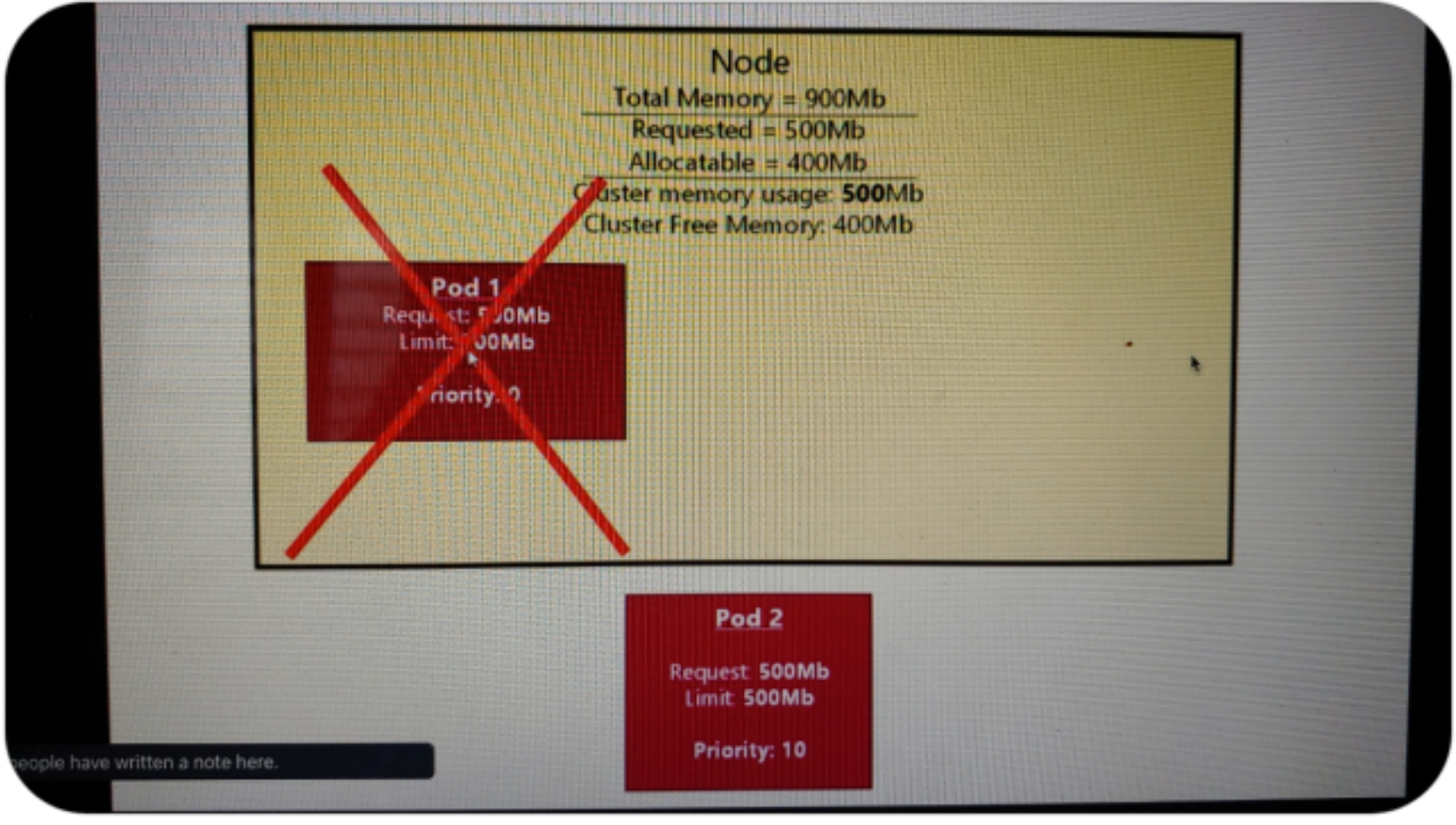


≡ ○ Now scheduler will try to deploy the evicted Pod 3 in some other node which has the memory. If there are no nodes available and if we defined cluster auto scaling then new nodes will be created automatically where this pod will be deployed.

≡ ○ Over time, Pod 1 started to use more memory and there is a shortage of resource. Now scheduler will start to evict pods of label Brustable which is Pod 2

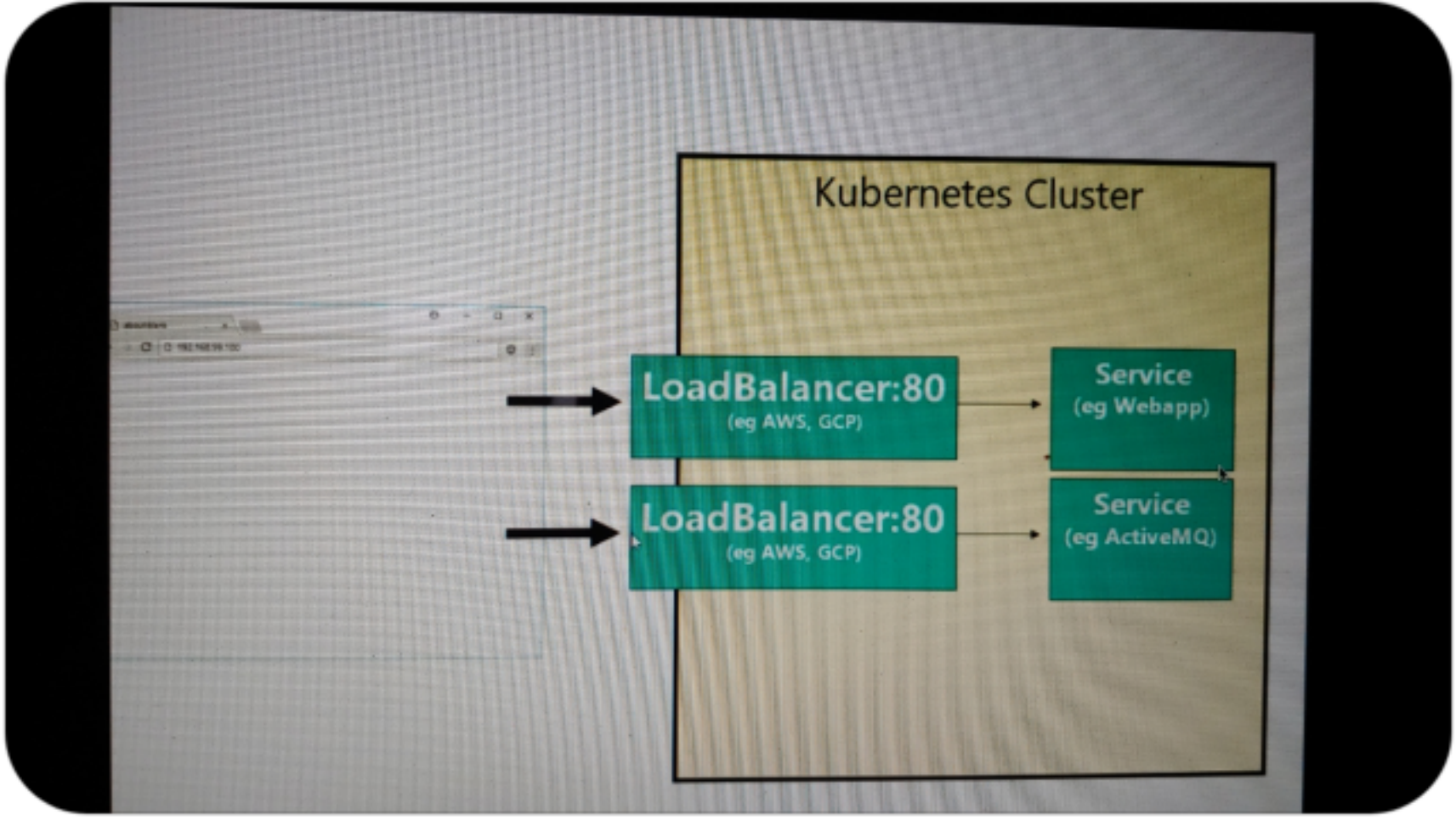


- **Pod Priorities:** While creating a Pod, we can mention priority for each pods. So later, while scheduler tries to evict a Pod, instead of looking for the QoS labels now it decide it based on the priority of the pods. So pods with lower priority will be evicted and pods with higher priority will be deployed.



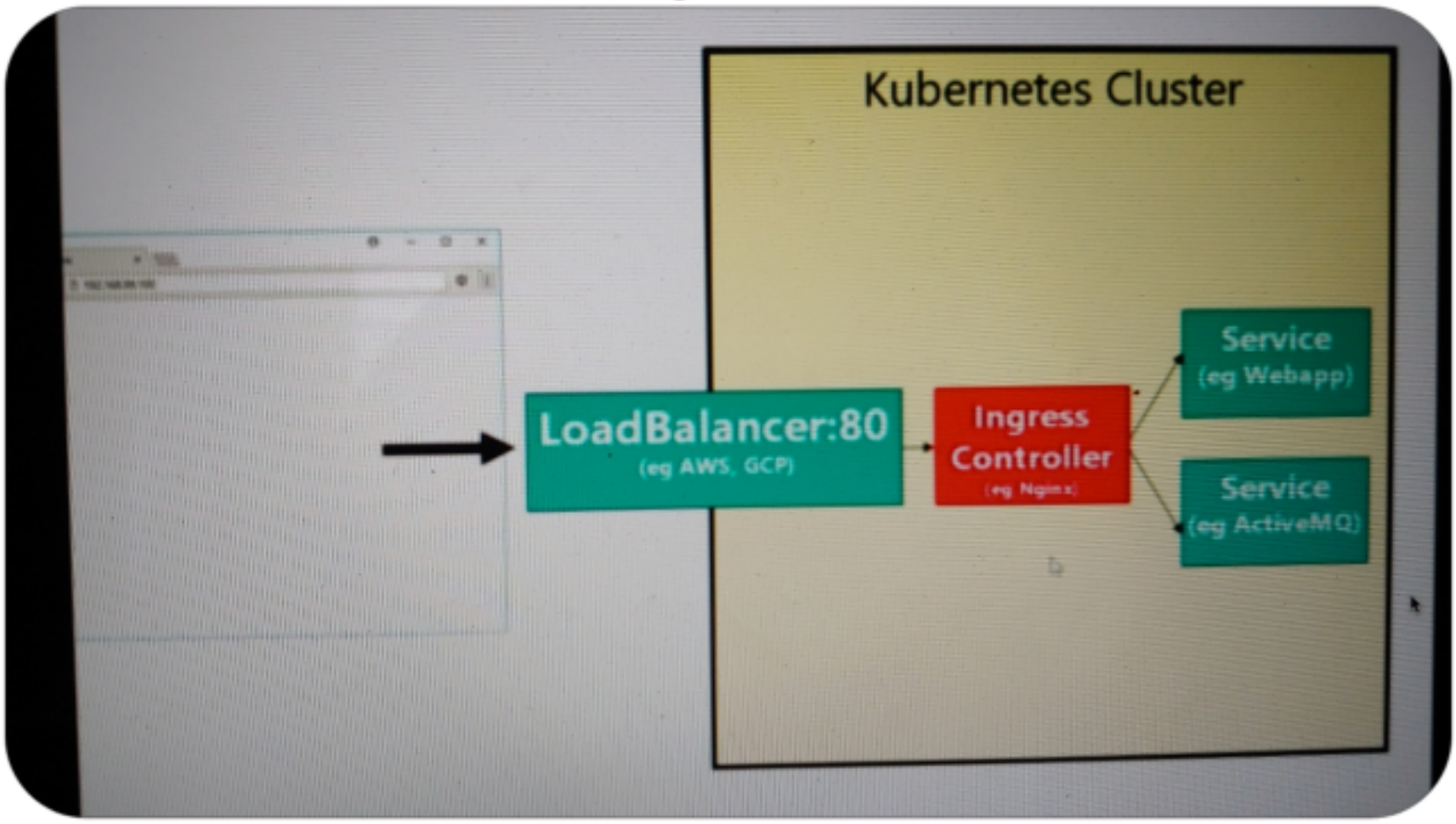
Ingress Controller

- Only restrictions we have with Services of type NodePart is to use the port only in the range of 30000 or above. Another option is to use services of type LoadBalancer in cloud. But usually these are quite expensive and if we have multiple web apps which meeds to be exposed to client then we end up with buying multiple Load Balancer one for each web apps.



- **Ingress and Ingress Controller**: The general solution for this problem is adding Ingress. Now we add another component called Ingress between multiple services and a single Load Balancer. This ingress is an object that allows access to your k8s services from outside the k8s cluster. Traffic routing is controlled by rules defined on the Ingress resource and are controlled by Ingress Controller. There are different types of implementation of Ingress

Controller and the most popular one is the Nginx.



- **Defining Routing Rules:** We can define a new yaml file of type Ingress with below details to create an Ingress Controller:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: basic-routing
spec:
 rules:
  - host: fleetman.com
    http:
     paths:
      - path: /
        backend:
         serviceName: fleetman-webapp
         servicePort: 80
```

- We can define multiple host definition for each service with its corresponding host name, service name and service port.
- Now we can apply this file and access the application with url: fleetman.com:80
- Note: We can also add basic authentication in Ingress Controller to restrict direct access to lets say ActiveMQ console which is exposed by this Ingress Controller.

Other Workload Types

- **Batch Jobs**: A job will create a pod and K8s ensures, that pod successfully runs to completion. Basically other kind of services generally long running process. But if we have a requirement where we need to do some kind

of batch job which needs to run frequently and then should complete. For example creating some report

```
apiVersion: batch/v1
kind: Job
metadata:
 name: test-job
spec:
 template:
  spec: # Pod
   containers:
   - name: long-job
    image: python:rc-slim
    command: ["python"]
    args: ["-c", "import time; print('starting'); time.sleep(30); print('done')"]
   restartPolicy: Never
 backoffLimit: 2
```

- Restart Policy: If a pod completes or faces any error then it uses this setting. If this is set to Never then pod never restarts, if it is set to Always then pod will restart always and if it is set to OnFailure then it restarts only of it is failed.
- BackoffLimit: If this job fails how many times you want k8s to retry
- **Cron Jobs**: To schedule a job on a regular basis. For example billing

process we need to run every day.

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cron-job
spec:    # CronJob
  schedule: "* * * * *"
  jobTemplate:
    spec:    # Job
      template:
        spec: # Pod
          containers:
          - name: long-job
            image: python:rc-slim
            command: ["python"]
            args: ["-c", "import time; print('starting'); time.sleep(30); print('done')"]
          restartPolicy: Never
      backoffLimit: 2
```

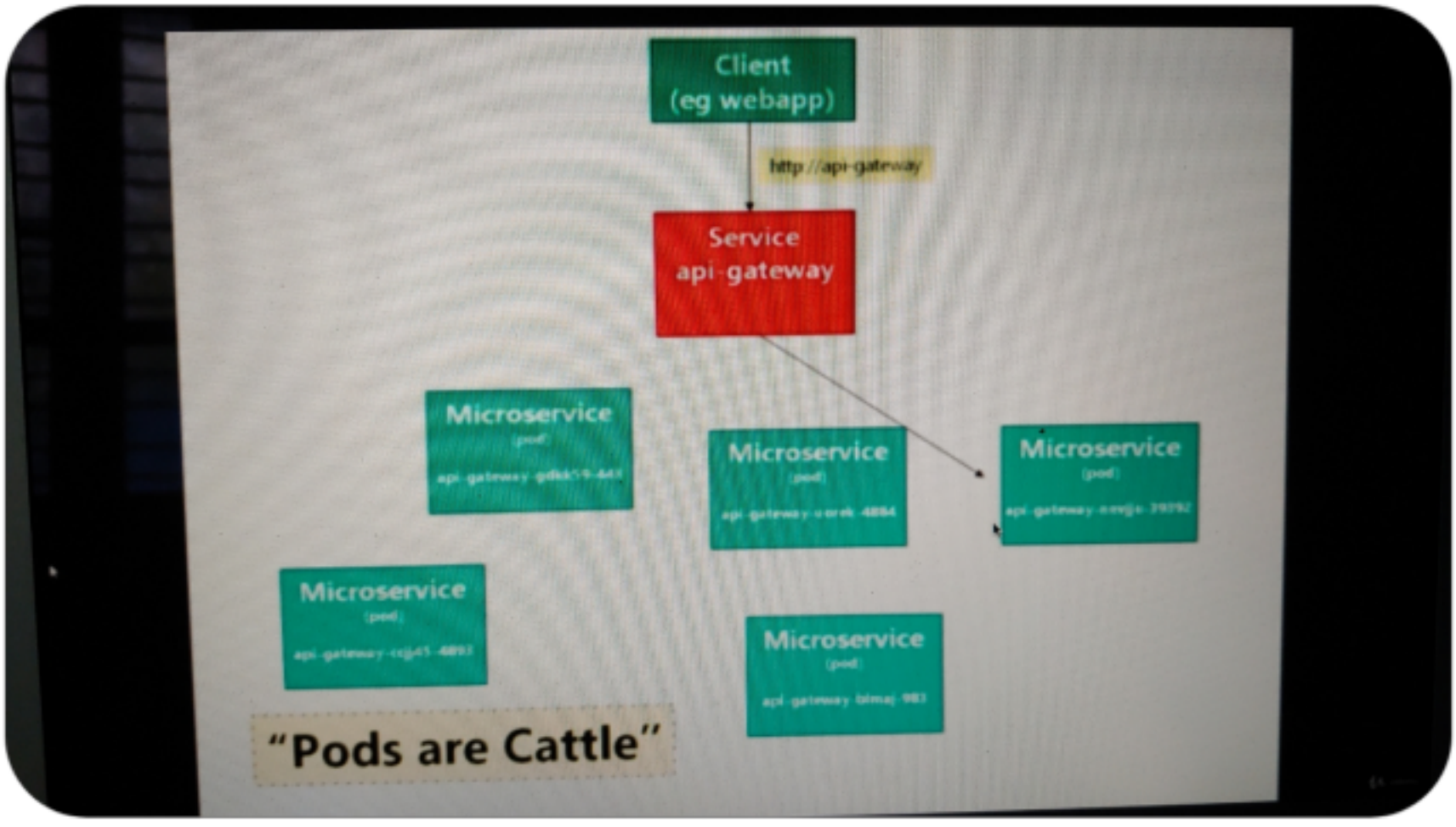
- **DaemonSets:** If we have a requirement where we need to run a pod in all available nodes, then we can replace the deployment by DaemonSet in yaml file and delete the replicas option. So that, k8s ensures that this pod will be running in all available nodes.
- For example running a logs collection daemon on every node such as fluentd or logstash.

9 June 2021 at 4:40 PM

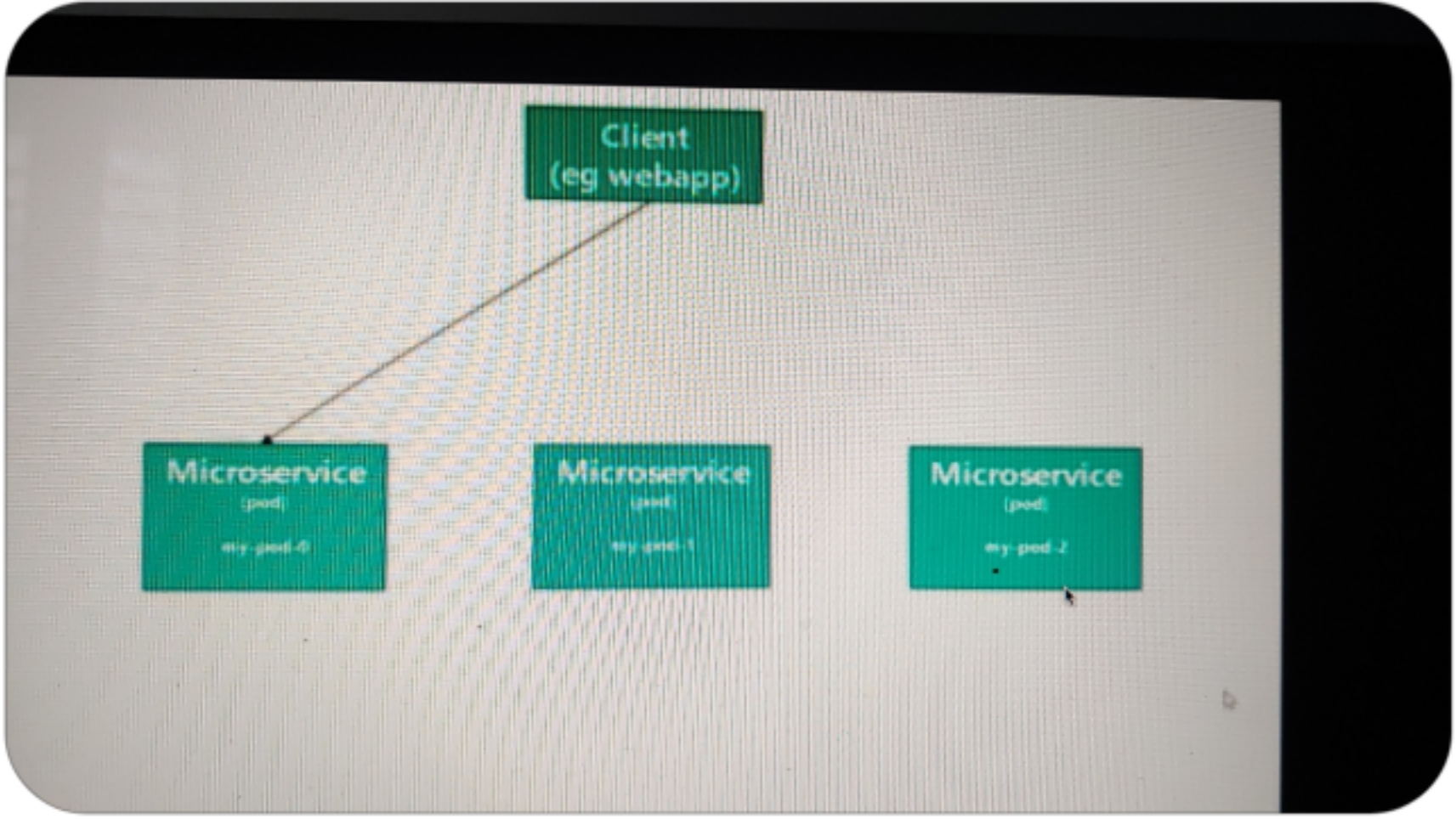
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: webapp
spec:
 selector:
  matchLabels:
   app: webapp
 # replicas: 1
 template: # template for the pods
  metadata:
   labels:
    app: webapp
  spec:
   containers:
   - name: webapp
     image: richardchesterwood/k8s-fleetman-webapp-angular:release2
```

- **StatefulSets:** Used to create pods with static names like pod-0, pod-1 etc. It is not used for persistence. When we create a deployment, the pods created by it will have some nasty character in its name. This is required because we can create multiple instances of this pod with different character for each of those instance names. Now if a client wants to call these pods, it has to call it through a service by service name. Now service will route this request to a random

instance of the pod.

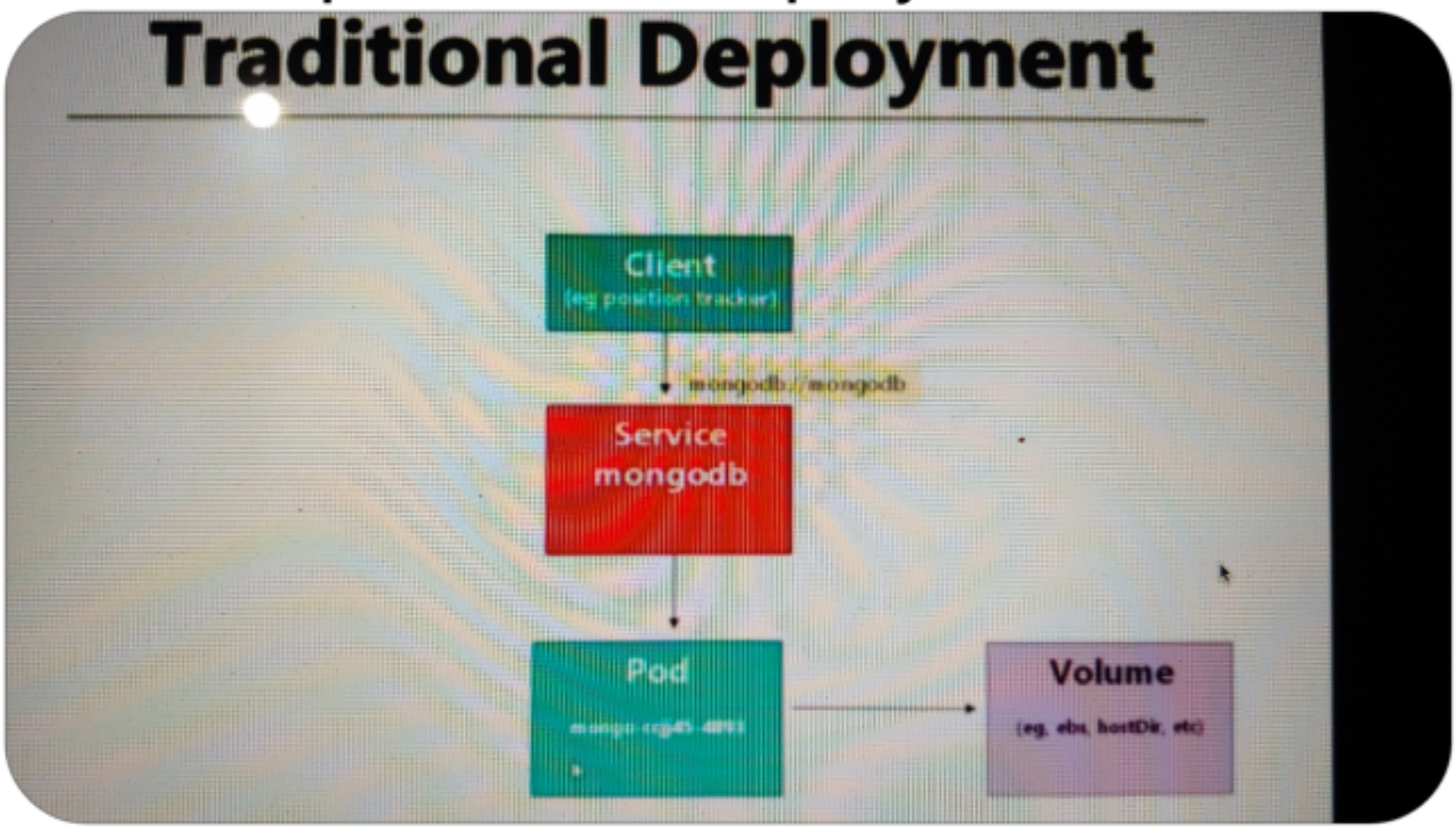


- Now, in some situations we need a set of pods with known, predictable names instead of random characters. And this is required by a client, so that they will be able to call them directly by their pod name. This is what we get from StatefulSets. It helps to create list of pods which have same name.



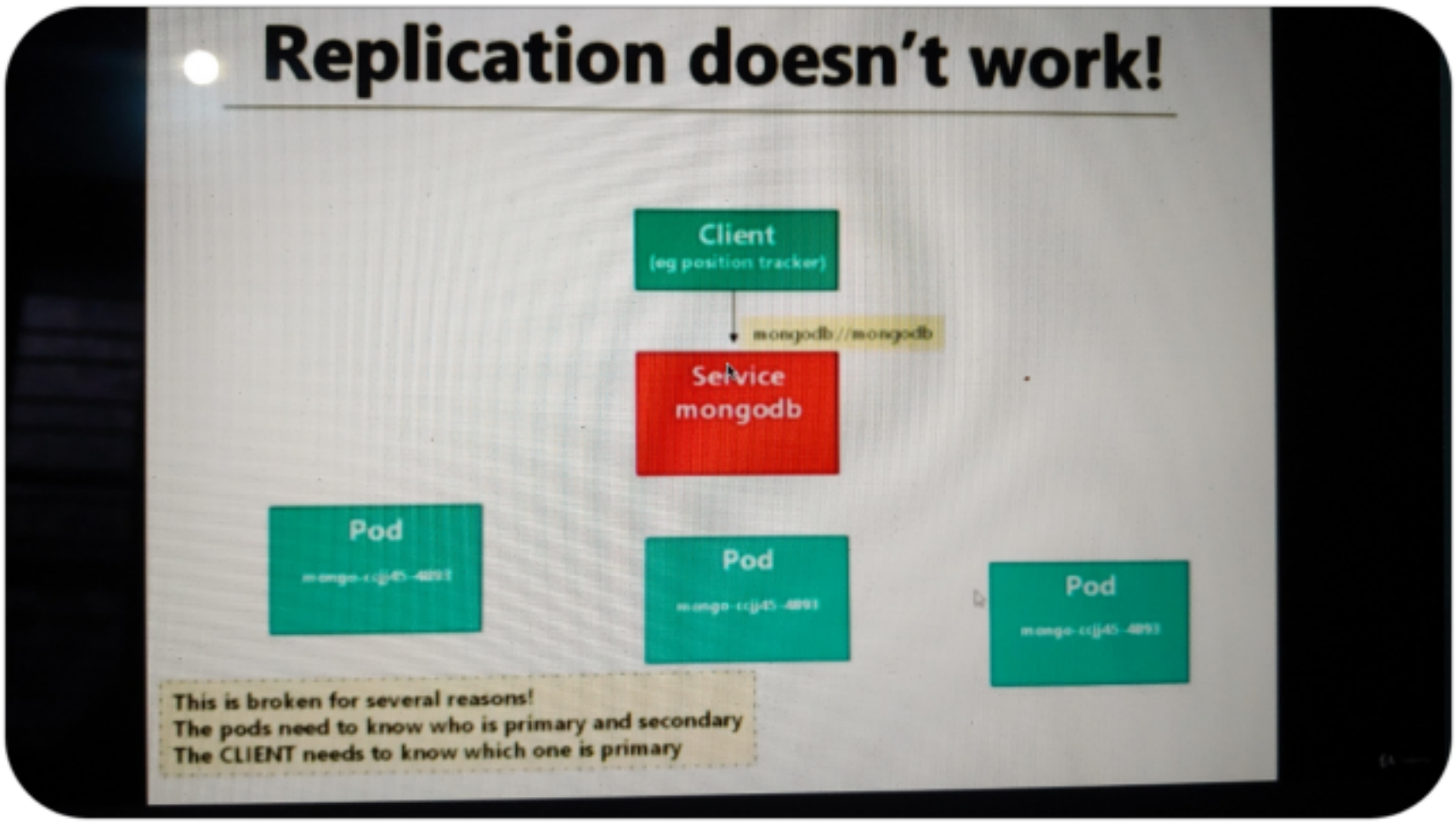
- Here, client will call specifically pod with name my-pod-2 and later it will call another pod with name as my-pod-0 with the help of special service called headless service.
- If we want to create pods which will have a predictable name like 0, 1, 2 then we can use StatefulSets provided by k8s instead of deployment. The pods will always start up in sequence. Clients can directly address them by name. So Stateful really means StarfulName.

- **Use case of StatefulSets**: We usually use this for database replication like Mongo DB replication. Database can't usually be replicated by simple ReplicaSets/Deployments.



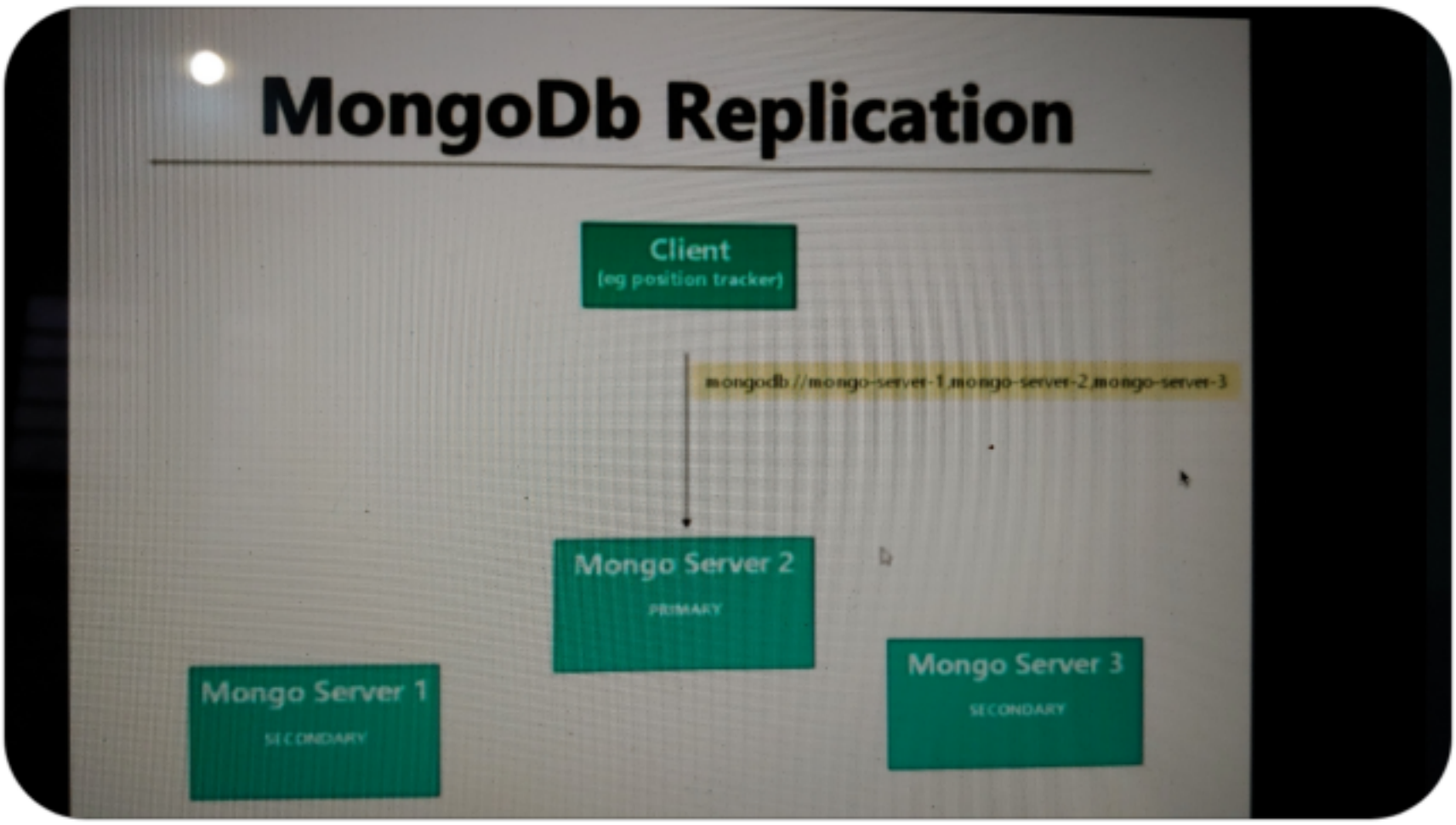
- In above scenario we deployed one instance of Mongo DB Pod and exposed it via a service. Now if we want to increase the replica of this Mongo DB pod from 1 to 3 through deployment replicas option it will not work. This wont work because of the way databases works. Each pod will

not know each other and collaborate each other.



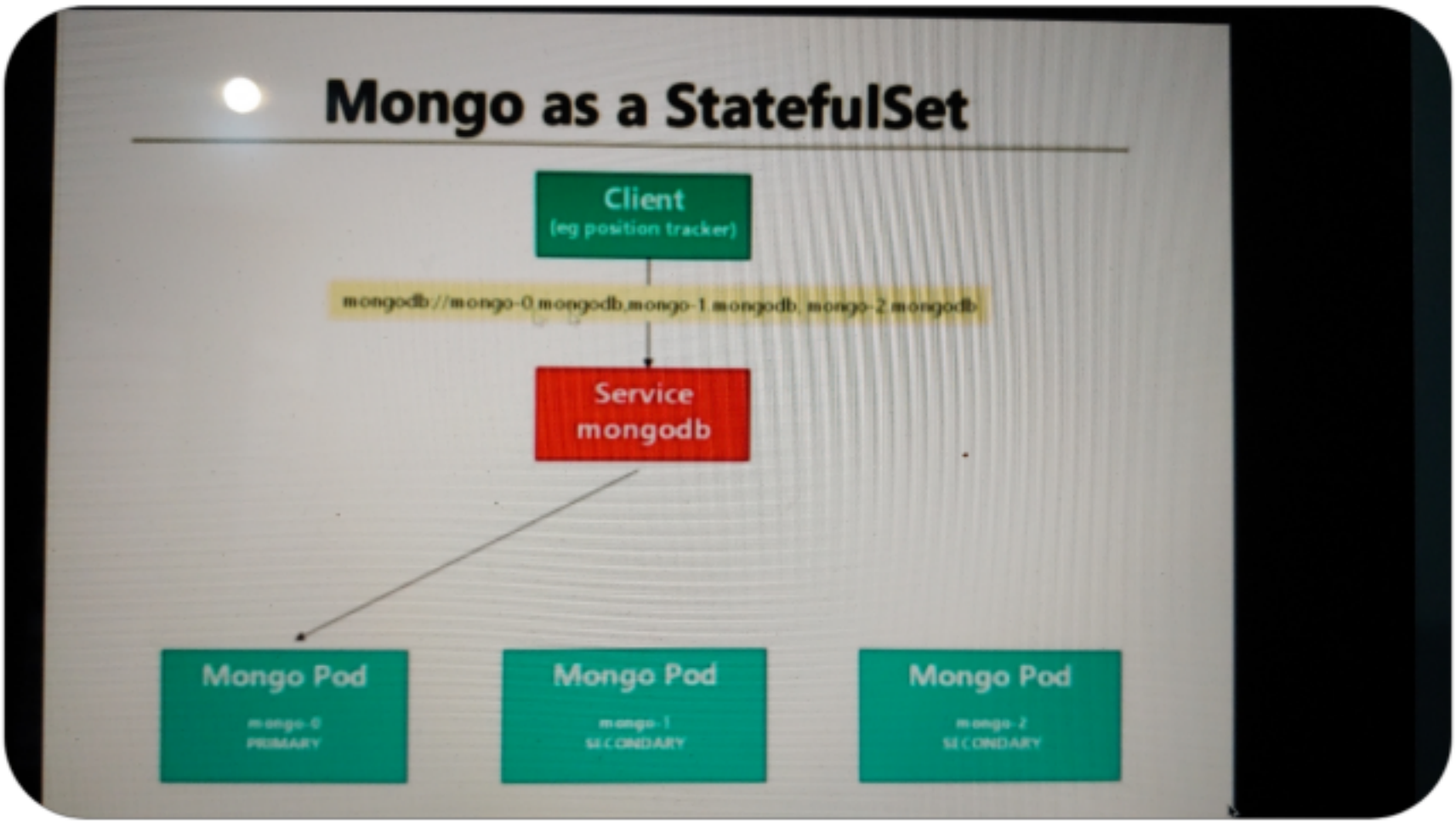
- **Mongo DB Replication**: In a Mongo cluster we can have set of multiple database services. We have to configure these services so that they are aware of each other. There will be one primary DB service and set of other secondary service. So whenever data is inserted to primary data will be copied to all secondary automatically. And client always make a call to a specific primary database instance URL. Normally

we feed the client with comma separated list of services in the cluster. And client code iterate over the service to find the primary. If primary crash, one of the secondary will be elected as primary. So by using traditional k8s model this cant be achieved.



- **Mongo as a StaefulSet**: By using StatefulSet, we can create multiple instances of pod with name like mongo-0, mongo-1, mongo-2. Now client will make a call to headless service with URL of the primary pod. Here client

knows which is the primary pod and send a request to the primary pod by its name. Format of the URL is <headless_seevice_name>:<mongo-instance-name.<headless_service_name>. We can create a yaml file of kind StatefulSet with all other details to create it.



Note: Usually we dont deploy any databases inside k8s cluster as it will be complex to manage, taking backups, recovery etc. We usually use any hosted database services provided by any cloud

providers instead of deploying them inside a pod of k8s.

Role Based Access Control

- Role Based Acccess Control (RBAC): defines role and rolebinding or clusterrole or clustorerolebinding. To provide or restrict accees to specic resources of the k8s. Role and rolebinding are fpr specific namespace and clusterrol for entire k8s cluster.role will have set of rules for set of resources which ll be binding to specific user
- If multiple people are working on a project and we need to restrict permission for each individuals on the k8s resources then we need to use RBAC. It controls what the developer on the cluster

can do with kubectl command.

- **Requirement**: We have a new joined starting tomorrow. As a system admin, I want them to able to get experience on thr live cluster, but safely. They should be able to look at all the resources in the cluster(pods, deployments, service etc) and create their own pods/deployments in their own 'playground' namespace.
- **Defining Roles**: We define set of roles, where each role contains rules that represent set of permissions. Foe ex we set get, list and watch permissions on k8s resources like pods, deployments, services etc which are part of mentioned apiGroups.

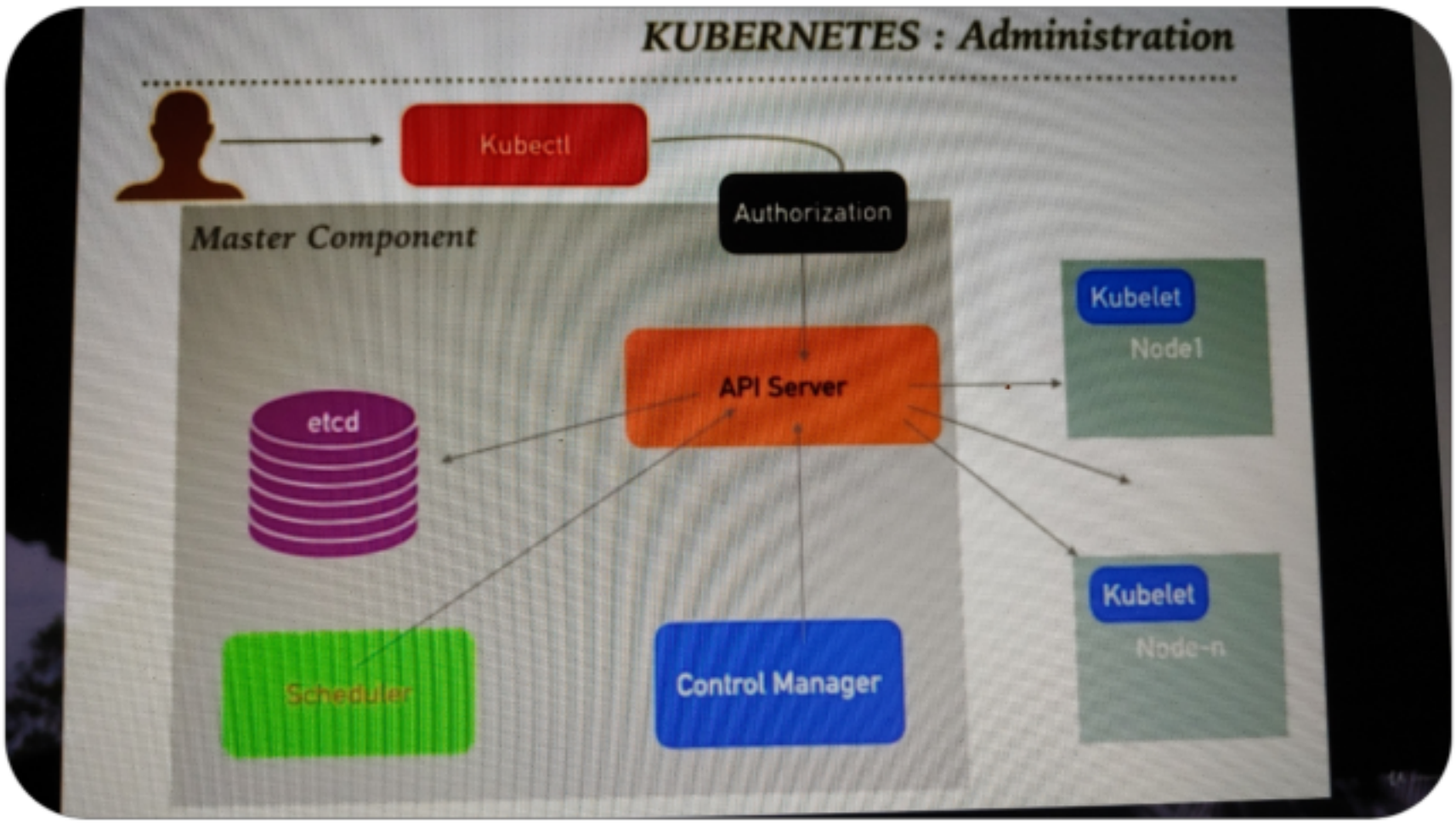
```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: new-joiner
  namespace: default
rules:
- apiGroups: ["","apps"]   # Core API AND apps
  resources: ["*"]     # pods, services, deployments...
  verbs: ["get", "list", "watch"]
```

- **Defining RoleBindings**: We define RoleBindings to put multiple users into each role.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: put-specific-user-or-users-into-new-joiner-role
  namespace: default
subjects:
- kind: User
  name: francis-linux-login-name
roleRef:
  kind: Role
  name: new-joiner
  apiGroup: rbac.authorization.k8s.io
```

- In subjects section we need to list all users that are going into this role. K8s does not provide

any guidance on how we can **create a user** name. We need to use some external **Authentication** for this to work, so that all the kubectl command will be authenticated first in k8s master node before it reach API Server.



- In RoleRef we are linking all the users defined above to a particular role by role name which we created earlier.
- Both Role and RoleBinding are bound to namespaces. Here we have provided the namespace as default.

- To create the users we need to depend on any outside independent service for example by distributing private keys.
- **ClusterRole**: Above Role and RoleBinding we created in a default namespace. So user can do get list and watch in the default namespace but not in other namespaces. For other namespaces to work we need to create similar files for all the available namespaces, which will be duplicate of code.
- We can Create ClusterRole instead of normal Role, since ClusterRole are not part of any namespaces, and it will become cluster wide. For this we can change the kind from Role to ClusterRole and remove namespace option.
- **ClusterRoleBinding**: It is also

similar to RoleBinding but this will make the binding cluster wide instead to a specific namespace.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: new-joiner
rules:
- apiGroups: ["","apps"]    # Core API AND apps
 resources: ["*"]     # pods, services, deployments...
 verbs: ["get", "list", "watch"]

---

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: put-specific-user-or-users-into-new-joiner-role
subjects:
- kind: User
 name: francis-linux-login-name
roleRef:
 kind: ClusterRole
 name: new-joiner
 apiGroup: rbac.authorization.k8s.io
```

≡ ○ Now our requirement is, user should be able to create any resource under the playground namespace. For this we can create a new Role and RoleBinding definition files and with all permissions in the playground namespace only.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: new-joiner
  namespace: playground
rules:
- apiGroups: ["","apps","extensions"]   # Everything
 resources: ["*"]     # Everything
 verbs: ["*"]

---

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: put-specific-user-or-users-into-new-joiner-role
 namespace: playground
subjects:
- kind: User
 name: francis-linux-login-name
roleRef:
 kind: Role
 name: new-joiner
 apiGroup: rbac.authorization.k8s.io
```

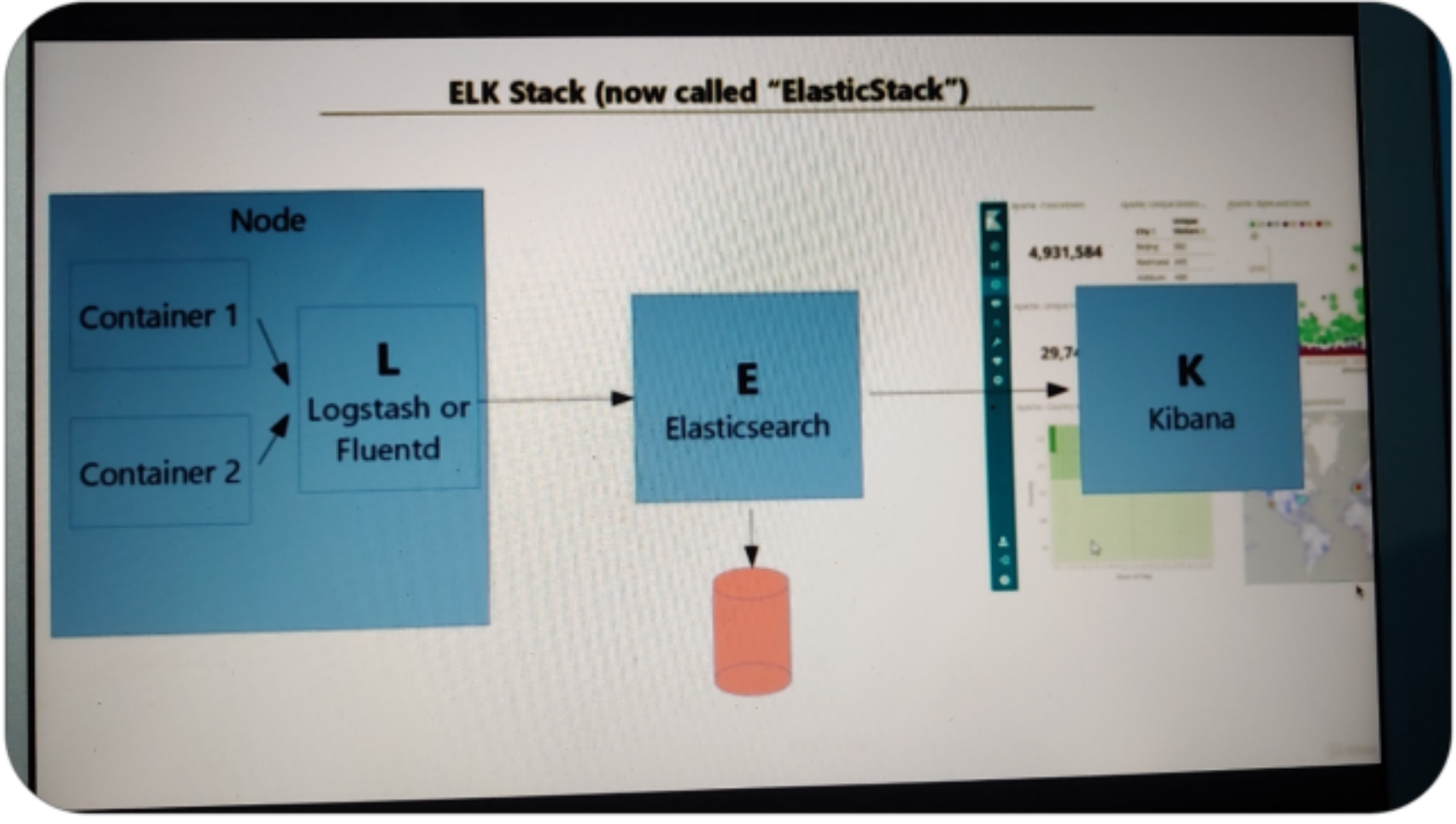
- Note: Instead of using Subjects of kind User, we can also use Subjects of kind ServiceAccount.

Logging in a K8S Cluster

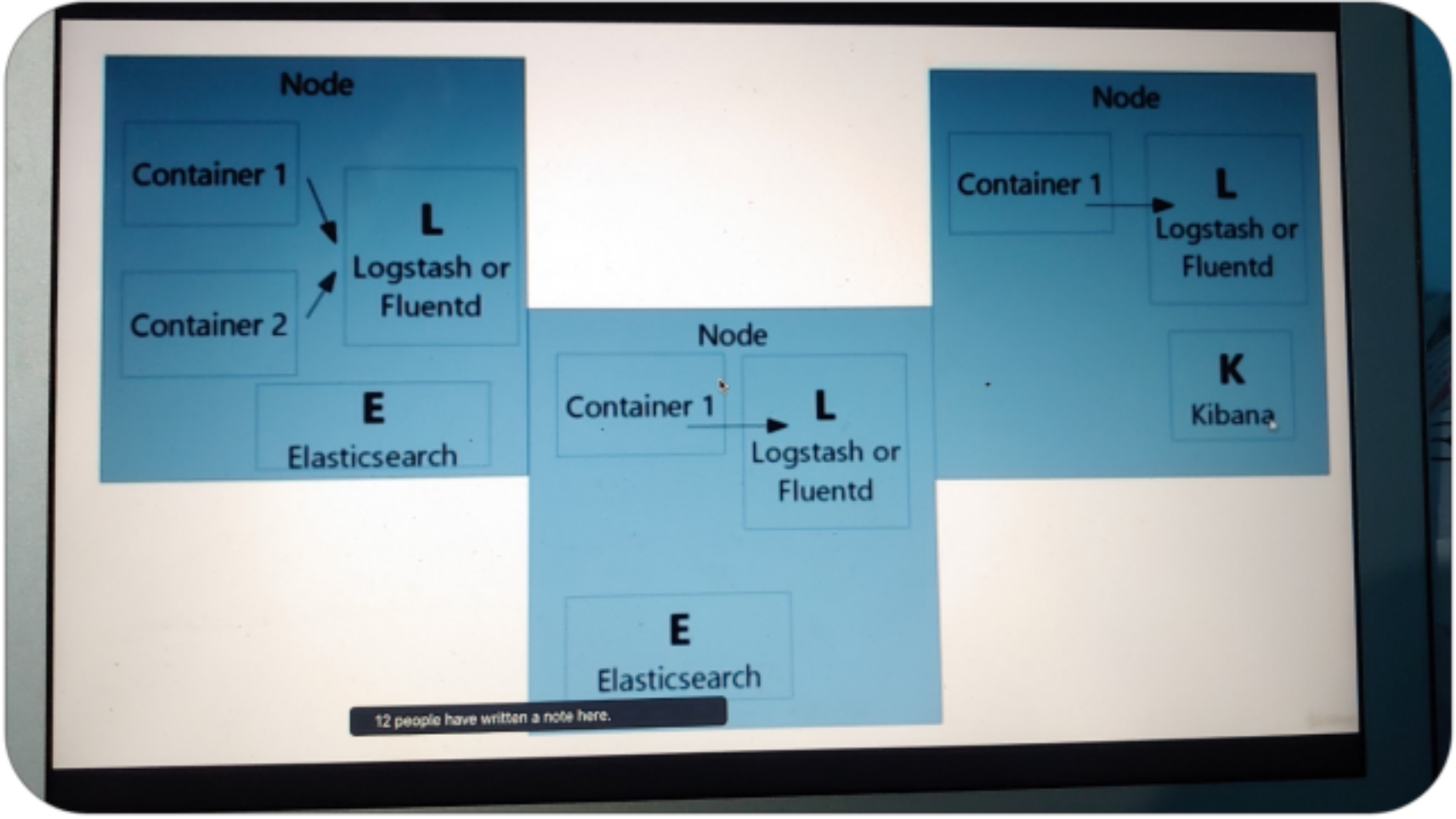
- When a pod restarts in k8s all the logs related to that pod also get deletes. This will be problem since those logs can contains some critical data which we would like analyze later.
- We use the ElasticStack tool to collect, transfer, store and visualize these logs data from all

pods.

- ○ We usually install a software called Logstash or FluentD in each node of the k8s cluster where multiple containers are running. All the containers will be writing their logs and Logstash capable of finding these logs and grabbing or pulling it.
- ○ Then Logstash which don't store logs by itself, will send all those gathered logs to another software called Elastic Search which has a Database and store all these logs.
- ○ Later we install another software called Kibana which will pull the logs data from ES and provide many options to visualize them in Kibana User Interface.



- All the cloud provider provides hosted ElasticStack which we can use.
- Or else, we can deploy these software directly to your k8s cluster. We have to deploy Logstash in every available node. And install replicated ES so there will be 2 instances of ES will be available. Finally instance Kibana in any one available node.



- Go to k8s official page, then go to addons page and find fluentd-elasticsearch to install all the required tools into your k8s cluster.

Monitoring a K8S Cluster

- Monitoring includes checking whether all your nodes healthy, are they consuming right level of CPU, are there enough nodes, are there any pods in stuck state etc.
- We care about smooth running of the hardware rather than looking

at the internal of the application which will taken care by logging.

- If we use any cloud platform we will get monitoring for free. But we can set up our own external monitoring system by using Prometheus and Grafana.
- Prometheus/Grafana:
  Prometheus is a tool which is capable of gathering various kind of metrics from running k8s cluster. It has a basic level of user interface. So to visualize these data of Prometheus we can plugin Grafana tool to it.
- To install these tools we can use kube-prometheus-stack. By following the steps provided in kube-prometheus-stack official page, we can apply bunch of yaml file to our k8s cluster so that it will run multiple pods for prometheus, grafana and alert

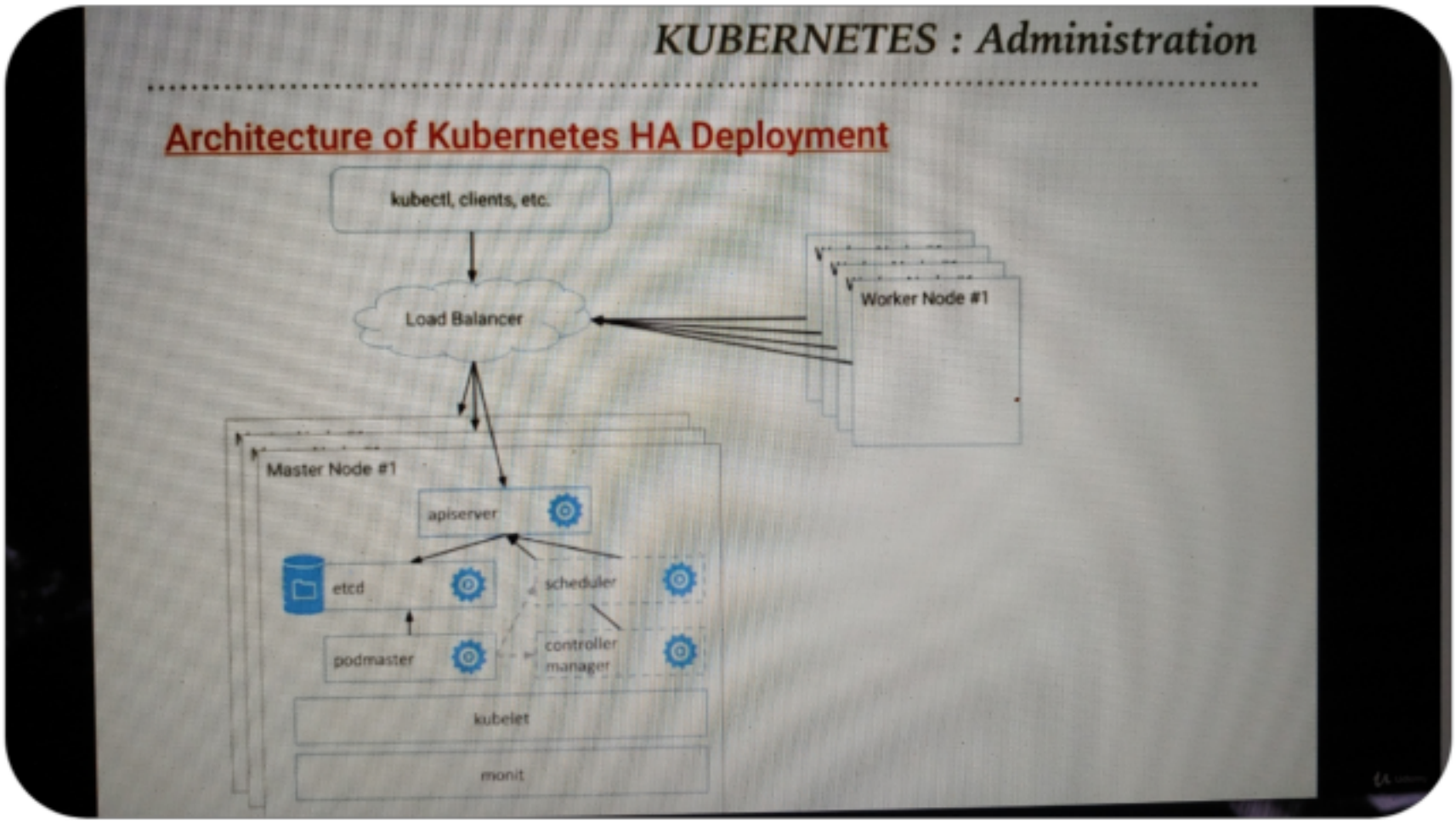
manager components.

- There is one component called **Alert Manager** which can be used to set up alert functionality in k8s cluster if any of the metrics data exceeds the expected maximum limit. We can send an email notification to configured email or push some notification to Slack channel with the details of the alert and its causes.

High Availability of K8S in Production

- Creating a HA k8s cluster means we want to have multiple k8s Master and Worker nodes running across multiple zones. HA is about setting up k8s along with its supporting components in a way that there is no single point of failure.
- In a single master cluster the

important components like API server, Controller Manager lies only on the single master node and if it fails you can not create more services pods etc. However in case of k8s HA environment, these important components are replicated on multiple masters(usually 3) and if any of the master fails the other master keep the cluster up and running.



- Here Scheduler and Controller Manager will be active on only one master node. Rest of the other components are active on

all master nodes.

Serverless Functions on K8S

- ○ Public cloud providers provide server less capabilities in which user can deploy functions directly without setting up server/VM. Serverless capabilities does not require to deploy containers or infrastructure to run.
- ○ Serverless functions are already running on containers behind the scene. It allow you to build applications at a high abstraction level so you can focus more on developing your applications and less on the infrastructure underneath.
- ○ Most popular server less frameworks are Fission, Kubeless, OpenWhisk, OpenFaas.
- ○ User needs to install these

frameworks on k8s cluster to use the Serverless functions. Administrator still needs to manage the k8s infrastructure.

- **Kubeless**: Its a k8s native server less framework that lets you deploy small bits of code/ functions without having to worry about the underlying infrastructure. It is deployed on top of k8s cluster.

Other Concepts

- Session affinity and node affinity: k8s Scheduler is responsible to deploy the pod in any available nodes depending on some rules.
- Pod Presets: It can inject informations into pods at creation time or into running pods. It can push secrets, volumes, volume mounts and

environment variables into pods, which internally pushes these values to all of its containers. User use pod label selectors to specify the Pods to which a given Pod Preset applies.

- ○ K8S watch Command: The main use of this is to call the Rest Api to the cluster and subscribe to changes. Ex: Spring Cloud Kubernetes uses this to watch for changes to ConfigMaps.

Other Tools

- ○ **Kompose**: takes docker-compose.yml file as an input and creates multiple k8s yml resource files.
- ○ **Kops**: kubernetes operations-for managing production grade k8s cluster

---

Common K8S Commands

- Get all resources: kubectl get all
- Get all pod: kubectl get pods
- Get a pod: kubectl get pod <pod_name>
- Delete a pod: kubectl delete pod <pod_name>
- Delete all pods: kubectl delete pod --all
- Deploy all resource: kubectl apply -f <yml file name>
- Details of a running pod: kubectl describe pod <pod_name>
- Details of a running service: kubectl describe service <name>
- Details of a running ReplicaSet: kubectl describe rs <name>
- Connect to a running pod: kubectl exec -it <pod_name> /bin/bash
- To rollout a deployment: kubectl

rollout status deploy <deployment_name>

- ≡ ○ Rollout history: kubectl rollout history deploy <deploymentname
- ≡ ○ Get all namespaces: kubectl get namespaces
- ≡ ○ To get all resources under kube-system namespace: kubectl get all -n kube-system
- ≡ ○ To get all persistence volume or persistence volume claim : kubectl get pv/pvc
- ≡ ○ To get all running nodes: kubectl get nodes
- ≡ ○ To get configmaps: kubectl get configmaps
- ≡ ○ To describe cm: kubectl describe cm <config_map_name>
- ≡ ○ To get secret: kubectl get secret <secret_name>
- ≡ ○ To describe a secret: kubectl describe secret <secret_name>

- To output any k8s resources to yaml format: kubectl get secret <secret_name> -o yaml
- To auto scale a deployment through HPA object: kubectl autoscale deployment api-gateway --cpu-percent 200 --min 1 --max 4
- To get all hpa: kubectl get hpa
- To describe a hpa: kubectl describe hba <hpa_name>
- To watch pods: kubectl get po --watch
- To get all roles: kubectl get roles
- kubectl describe role <role_name
- To get a rolebinding: kubectl get rolebinding <role_bibding_name>
- kubectl describe rolebinding nam

========== END==========