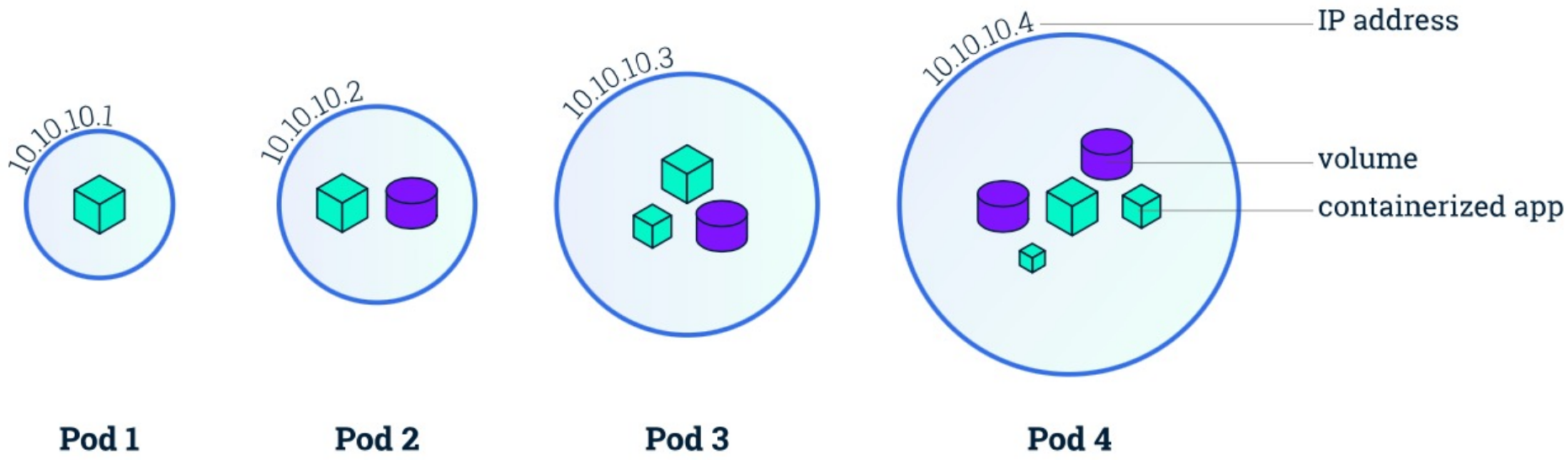
Kubernetes Avançado

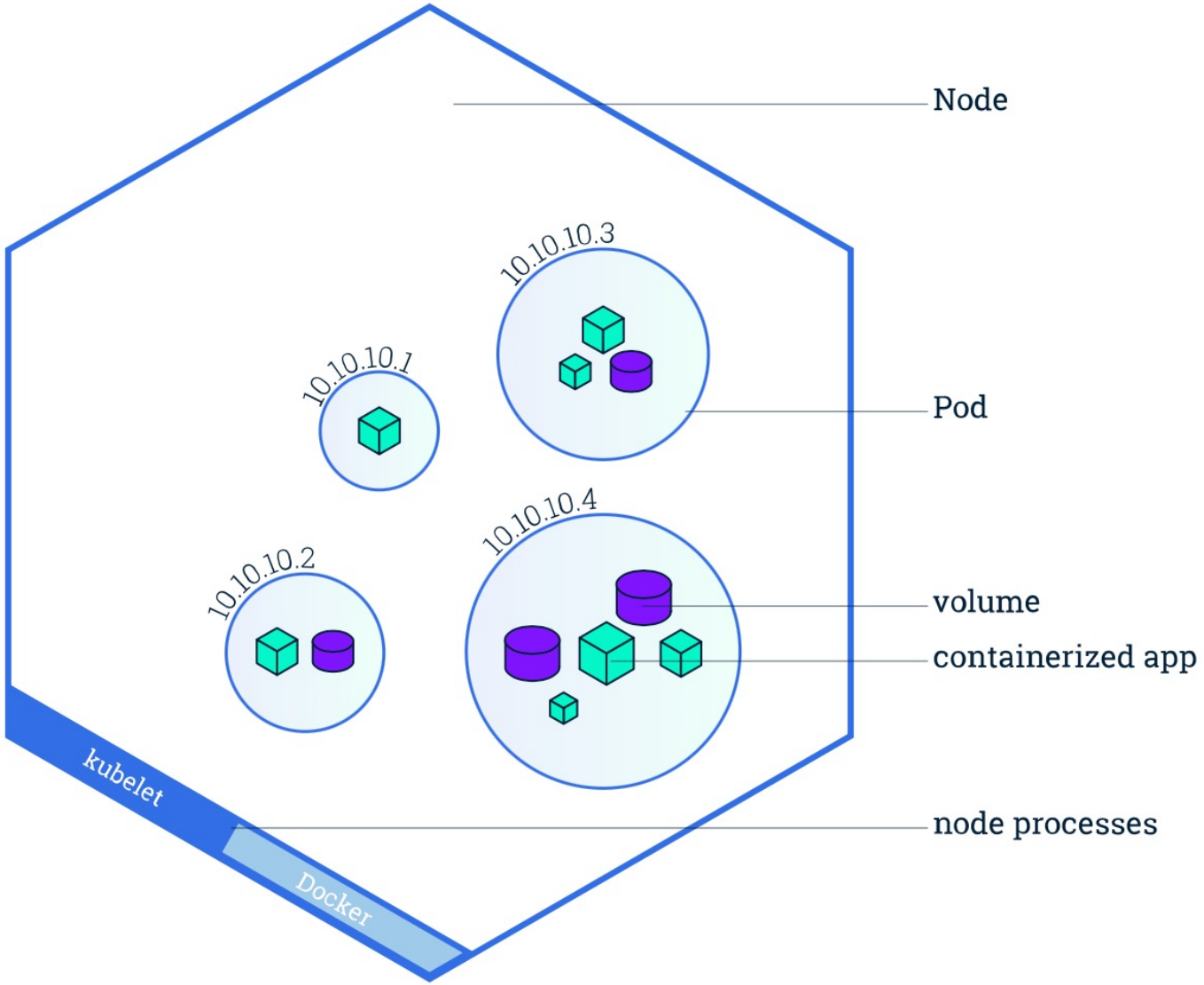
Aula Teórica nº11

2020/2021

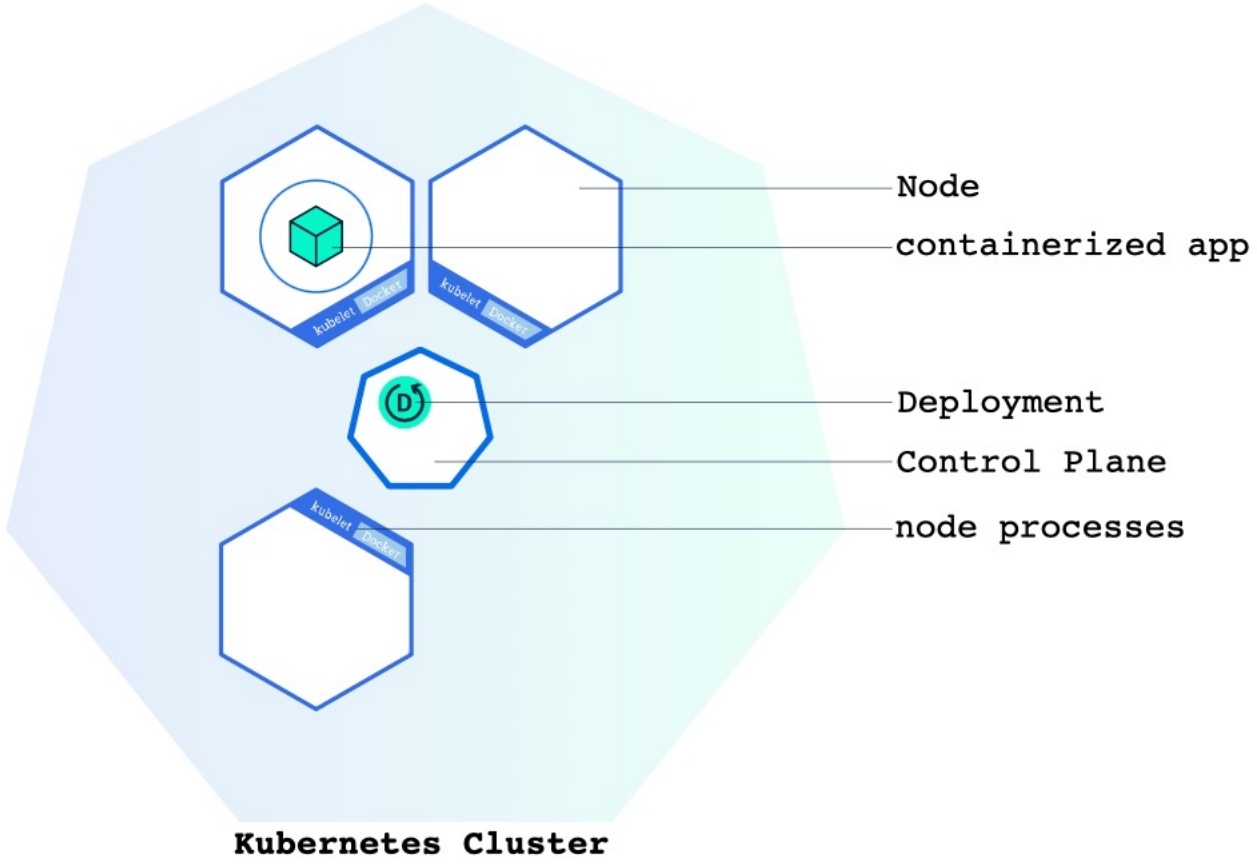
PODs Overview



Nodes Overview



Deploying an app on Kubernetes

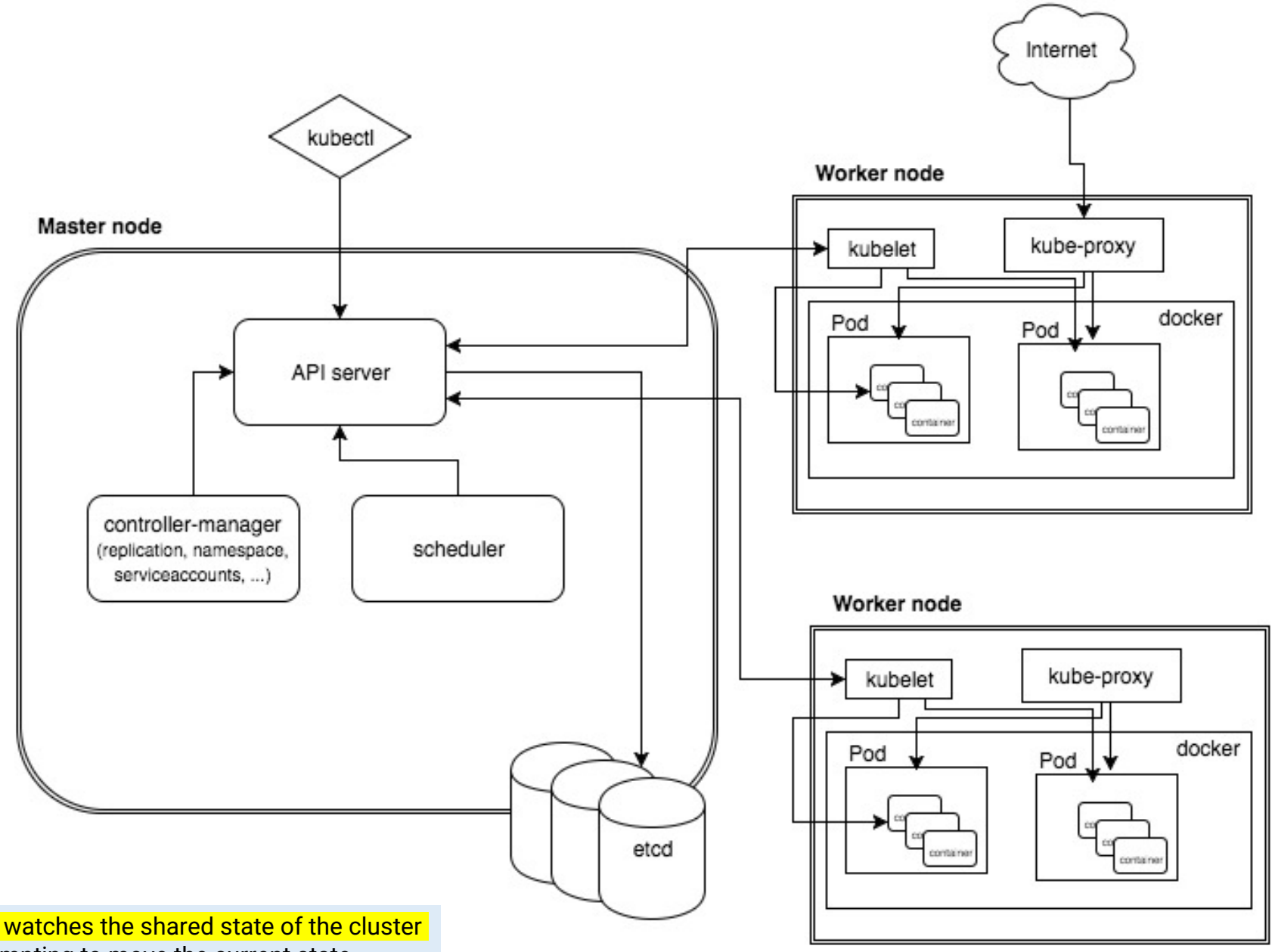


Architecture

The Kubernetes **API server** validates and configures data for the API objects which include pods, services, replication controllers, and others. The API Server services REST operations and provides the frontend to the cluster's shared state through which all other components interact.

The Kubernetes **scheduler** is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity. The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/ policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on. Workload-specific requirements will be exposed through the API as necessary.

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.



In Kubernetes, a **controller** is a control loop that watches the shared state of the cluster through the API Server and makes changes attempting to move the current state towards the desired state.

Pods and Services in the Kube System

```
ubuntu@kmaster:~$ kubectl get pods --namespace=kube-system
NAME                                READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-54rqb            1/1     Running   0          12d kmaster
coredns-66bff467f8-kxmg2            1/1     Running   0          12d kmaster
etcd-kmaster                        1/1     Running   0          12d kmaster
kube-apiserver-kmaster              1/1     Running   0          12d kmaster
kube-controller-manager-kmaster     1/1     Running   0          12d kmaster
kube-flannel-ds-amd64-4cz52         1/1     Running   0          12d kmaster
kube-flannel-ds-amd64-9vqqw         1/1     Running   0          12d kworker1
kube-proxy-h89qz                    1/1     Running   0          12d kmaster
kube-proxy-vgqm9                    1/1     Running   0          12d kworker1
kube-scheduler-kmaster              1/1     Running   0          12d kmaster
```

```
ubuntu@kmaster:~$ kubectl get svc  --namespace=kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   12d
```

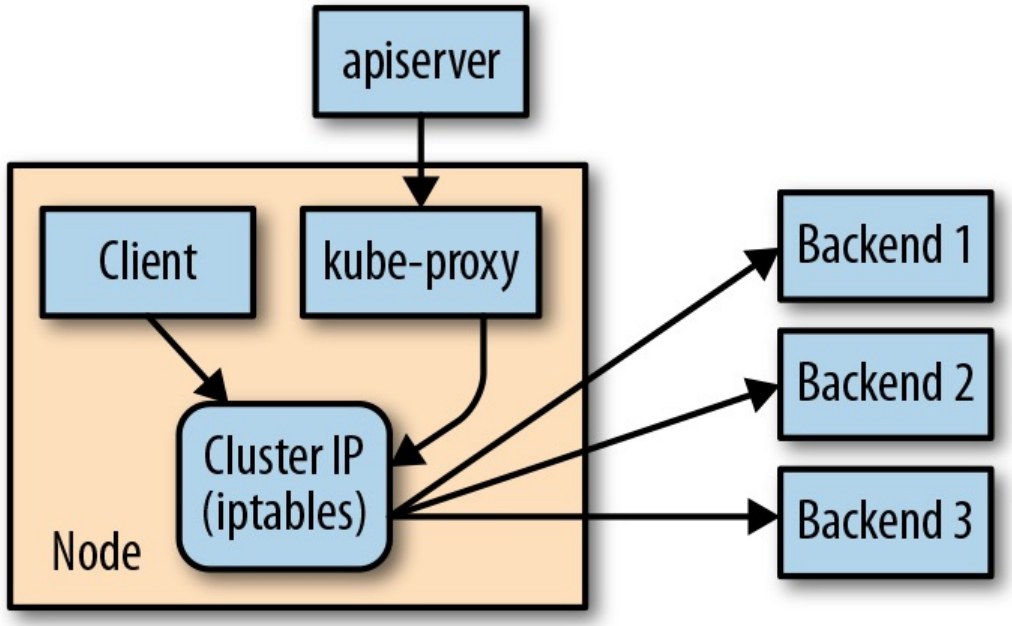
Kubelet

- Responsible for maintaining a set of pods on a local system.
- Within a Kubernetes cluster functions as a local agent that watches for pod specs via the Kubernetes API server.
- Responsible for registering a node with a Kubernetes cluster, sending events and pod status, and reporting resource utilization.

```
ubuntu@kworker1(or kmaster):~$ sudo systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Fri 2020-04-24 22:40:08 UTC; 1 weeks 5 days ago
     Docs: https://kubernetes.io/docs/home/
 Main PID: 6002 (kubelet)
    Tasks: 15 (limit: 2318)
   CGroup: /system.slice/kubelet.service
           └─6002 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf ...
```

kube-proxy and Cluster IPs

- Cluster IPs are stable virtual IPs that load-balance traffic across all of the endpoints in a service. This magic is performed by a component running on every node in the cluster called the `kube-proxy`



- The cluster IP itself is usually assigned by the API server as the service is created. However, when creating the service, the user can specify a specific cluster IP.

DNS for Services and Pods

- Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.
- Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain. This is best illustrated by example:
  - Assume a Service named **foo** in the Kubernetes namespace **bar**. A Pod running in namespace **bar** can look up this service by simply doing a DNS query for **foo**. A Pod running in namespace **quux** can look up this service by doing a DNS query for **foo.bar**.

Labels and Annotations

- *Labels* are **key/value pairs** that can be attached to Kubernetes objects such as Pods and ReplicaSets. They can be arbitrary, and are useful for attaching identifying information to Kubernetes objects. Labels provide the foundation for grouping objects. Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes
- *Annotations*, on the other hand, provide a storage mechanism that resembles labels: annotations are **key/value pairs** designed to hold nonidentifying information that can be leveraged by tools and libraries.

Reconciliation Loops

- The central concept behind a reconciliation loop is the notion of *desired* state versus *observed* or *current* state. Desired state is the state we want. With a **ReplicaSet**, it is the desired number of replicas and the definition of the Pod to replicate. For example, "the desired state is that there are three replicas of a Pod running the nginx server."
- In contrast, the current state is the currently observed state of the system. For example, "there are only two nginx Pods currently running."
- The reconciliation loop is constantly running, observing the current state of the world and taking action to try to make the observed state match the desired state.
- There are many benefits to the reconciliation loop approach to managing state. It is an inherently goal-driven, self-healing system, yet it can often be easily expressed in a few lines of code.

Pods and ReplicaSets

- Though ReplicaSets create and manage Pods, they do not own the Pods they create. ReplicaSets use label queries to identify the set of Pods they should be managing. They then use the exact same Pod API that we can use with `kubectl` commands to create the Pods that they are managing.
- In a similar decoupling, ReplicaSets that create multiple Pods and the services that load-balance to those Pods are also totally separate, decoupled API objects.
- The decoupling of Pods and ReplicaSets enables:
  - Adopting Existing Containers
  - Quarantining Containers

Autoscaling a ReplicaSet

- While there will be times when you want to have explicit control over the number of replicas in a ReplicaSet, often you simply want to have "enough" replicas. The definition varies depending on the needs of the containers in the ReplicaSet.
- With a web server like NGINX, you may want to scale due to CPU usage. For an in-memory cache, you may want to scale with memory consumption. In some cases you may want to scale in response to custom application metrics.
- Kubernetes can handle all of these scenarios via ***Horizontal Pod Autoscaling*** (HPA).

DaemonSets

- Deployments and ReplicaSets are generally about creating a service (e.g., a web server) with multiple replicas for redundancy. Another reason to replicate a set of Pods is to schedule a single Pod on every node within the cluster.
- A **DaemonSet** ensures a copy of a Pod is running across a set of nodes in a Kubernetes cluster. DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node. DaemonSets share similar functionality with ReplicaSets; both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match.
- ReplicaSets should be used when your application is completely decoupled from the node and you can run multiple copies on a given node without special consideration. DaemonSets should be used when a single copy of your application must run on all or a subset of the nodes in the cluster.

Service

- Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. If you use a Deployment to run your app, it can create and destroy Pods dynamically
- Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later

if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Service

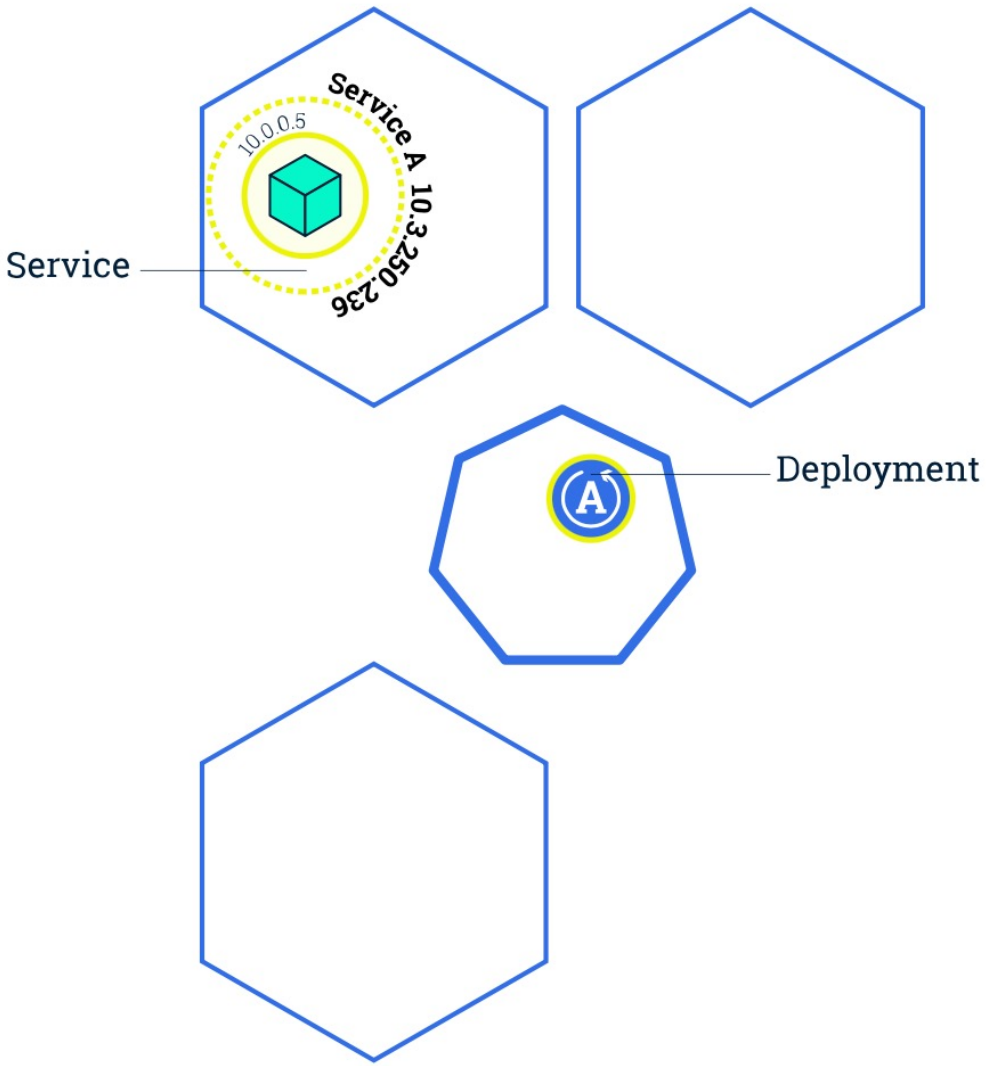
- A Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service).
- The set of Pods targeted by a Service is usually determined by a selector
- For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.
- The Service abstraction enables this decoupling

Service

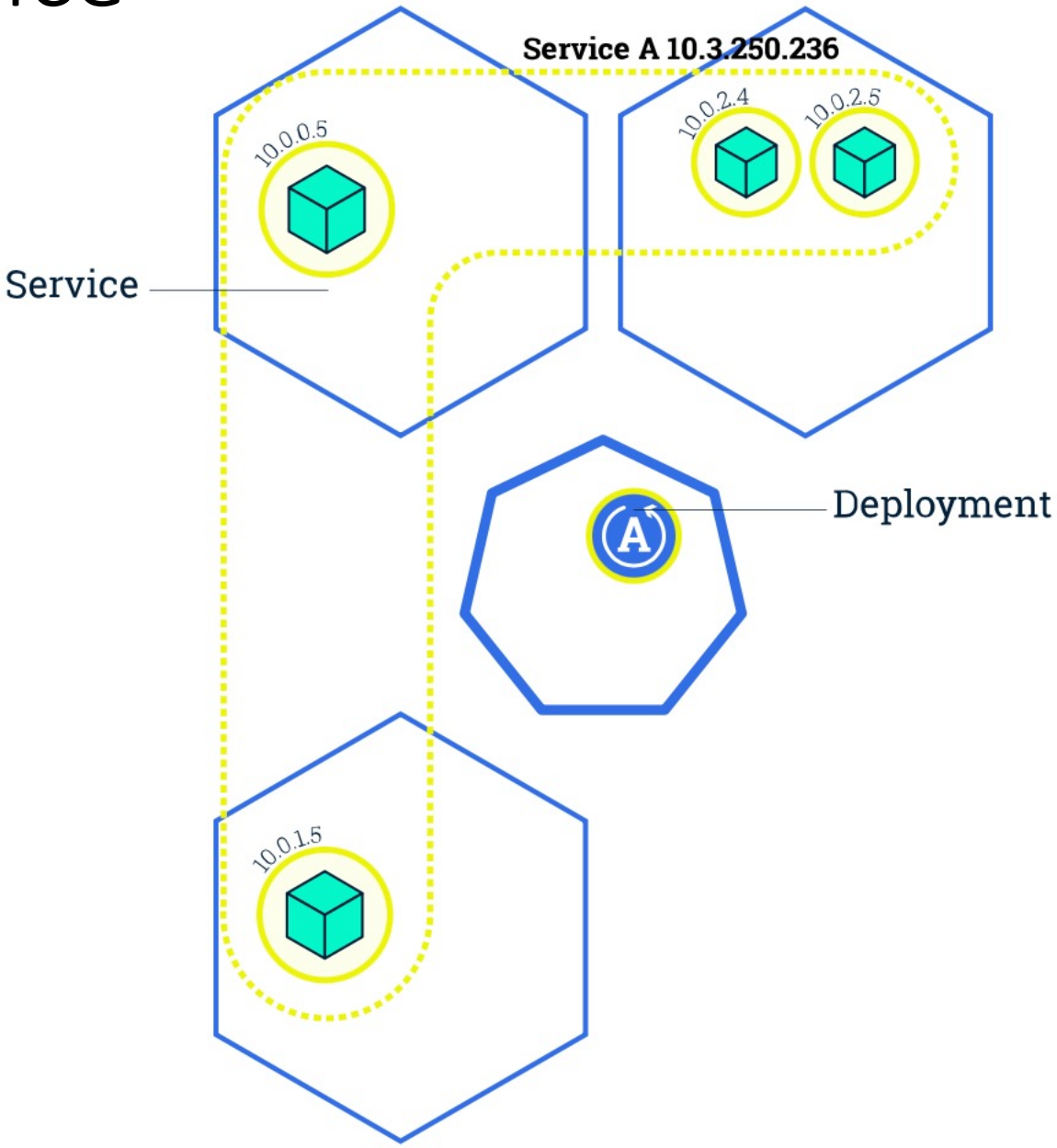
- Suppose you have a set of Pods that each listen on TCP port 9376 and carry a label app=MyApp
- This specification creates a new Service object named “my-service”, which targets TCP port 9376 on any Pod with the app=MyApp label
- Kubernetes assigns this Service an IP address (sometimes called the “cluster IP”)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Deploying a Service



Scaling a Service



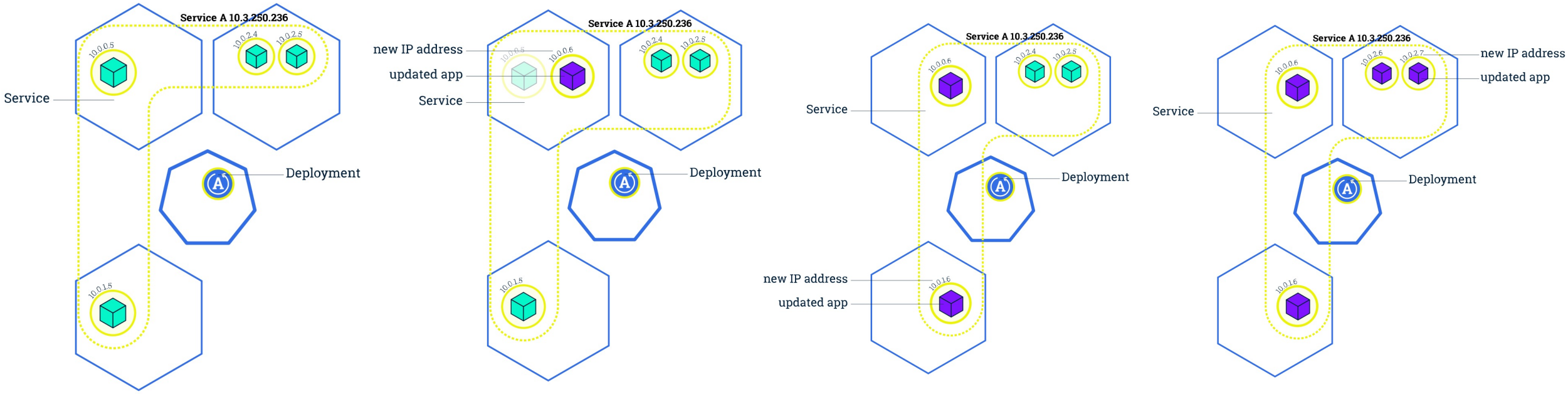
Multi-port Services

- For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object.
- When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

Rolling Updates

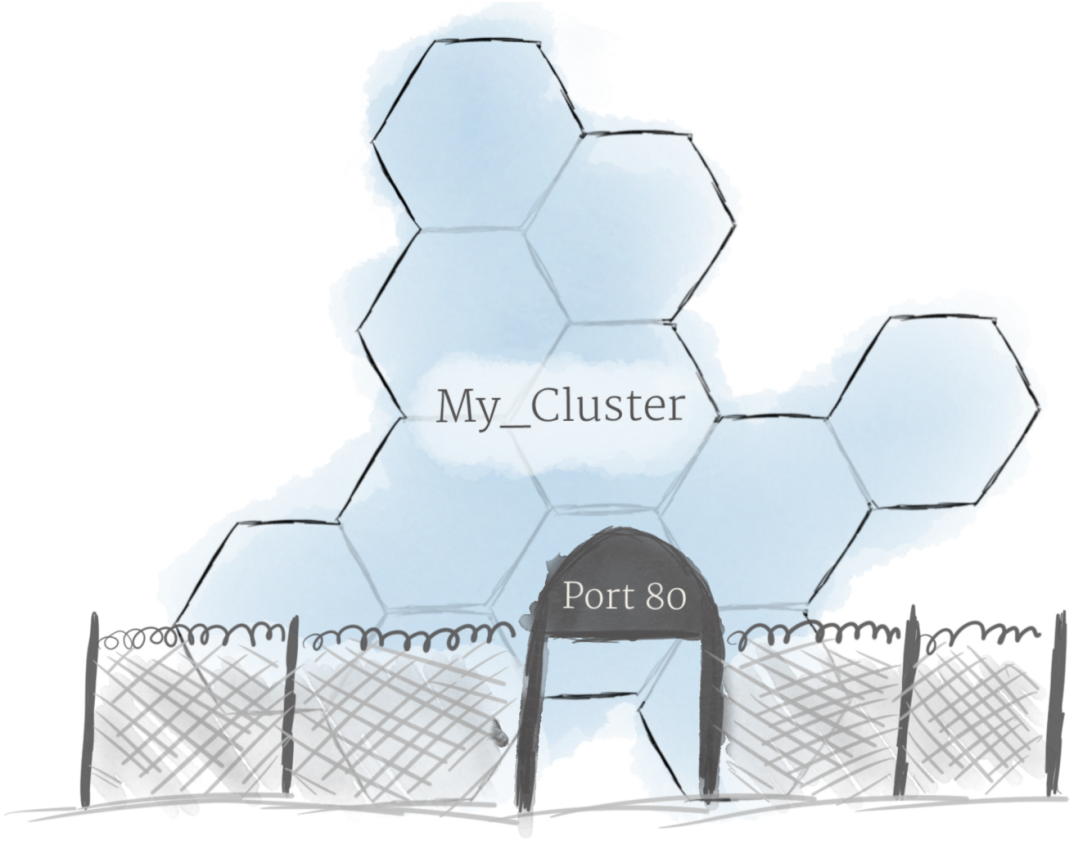
- Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.



kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
kubectl rollout status deployments/kubernetes-bootcamp
kubectl rollout undo deployments/kubernetes-bootcamp

Ingress

- By default, Kubernetes provides isolation between pods and the outside world
- If you want to communicate with a service running in a pod, you have to open up a channel for communication. This is referred to as ingress.
- There are multiple ways to add ingress to your cluster. The most common ways are by adding either an Ingress controller, or a LoadBalancer.



- When creating a **service**, you have the option of automatically creating a cloud network load balancer. This provides an externally-accessible IP address that sends traffic to the correct port on your cluster nodes

Namespaces

```
ubuntu@kmaster:~$ kubectl get namespace
NAME                 STATUS   AGE
default              Active   12d
kube-node-lease      Active   12d
kube-public          Active   12d
kube-system          Active   12d
```

- Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called **namespaces**.
- Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, we should not need to create or think about namespaces at all.
- Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace.
- Namespaces are a way to divide cluster resources between multiple users
- It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

Using Kubernetes

Example: Create POD

pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: myapp
  name: myapp
spec:
  containers:
  - name: myapp
    image: trxuk/clus-1999-app1:latest
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl create -f pod.yml
pod "myapp" created
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl get pod
NAME      READY     STATUS    RESTARTS   AGE
myapp     1/1       Running   0          16s
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl logs myapp
 * Serving Flask app "myApp" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production envir
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl describe pod myapp
Name:           myapp
Namespace:      default
Node:           docker-for-desktop/192.168.65.3
Start Time:     Wed, 05 Dec 2018 14:00:09 -0500
Labels:         <none>
Annotations:    <none>
Status:         Running
IP:             10.1.0.25
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ cat service.yaml
kind: Service
apiVersion: v1
metadata:
  name: myapp-service
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 5000
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl create -f service.yaml
service "myapp-service" created


MATJOHN2-M-J0PL:2-Slide38 matjohn2$ kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP
myapp-service   NodePort    10.99.181.192   <none>        5000:30470/TCP
```

```
MATJOHN2-M-J0PL:2-Slide38 matjohn2$ curl http://localhost:30470/
Hello Cisco LIVE! Cancun!
```