Container Orchestration w/Kubernetes

Tejas Parikh (t.parikh@northeastern.edu)

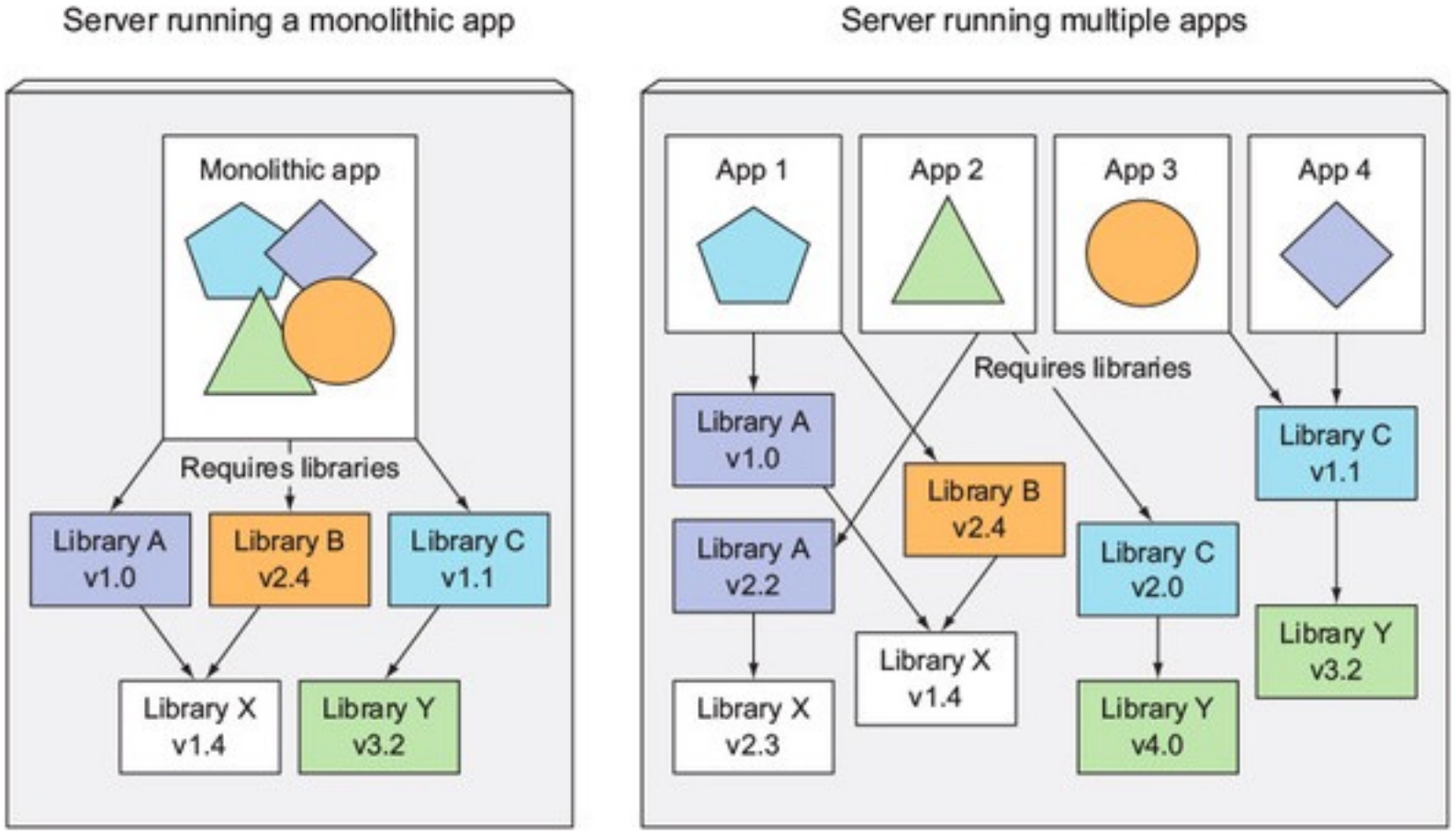
CSYE 7125 Advanced Cloud Computing

Northeastern University

Need for Orchestration

- Switch from Monoliths to Microservices applications.
- Rapid release cycles.
- Respond to system load automatically.
- Deployments are at a much larger scale.
- Need to use resources efficiently.
- DevOps or Noops
- Providing consistent environment for applications

Multiple applications running on the same host may have conflicting dependencies.



Kubernetes

Kubernetes

- Kubernetes is Greek for pilot or helmsman (the person holding the ship's steering wheel).
- Kubernetes abstracts away the hardware infrastructure and exposes your whole data center as a single enormous computational resource.

What make containers possible?

- **Linux Namespaces** make sure each process sees its own personal view of the system (files, processes, network interfaces, hostname, and so on).
- **Linux Control Groups** (cgroups), which limit the amount of resources(CPU, memory, bandwidth, etc.) the process can consume.

Linux Namespaces

- When running a process inside a namespace, the process will only see resources that are inside the same namespace.
- Process does not have to belong to one namespace. It can belong to one namespace for each of following kind:
  - Mount (mnt)
  - Process ID (pid)
  - Network (net)
  - Inter-process communication (ipc)
  - UTS (UNIX Timesharing System used for hostname and domain name)
  - User Id (user)

Linux Namespaces

- Each namespace kind is used to isolate a certain group of resources.
- For example, what network namespace a process belongs to determines which network interfaces the application running inside the process sees.
- Each network interface belongs to exactly one namespace, but can be moved from one name space to another.
- Each container uses its own Network namespace, and therefore each container sees its own set of network interfaces.

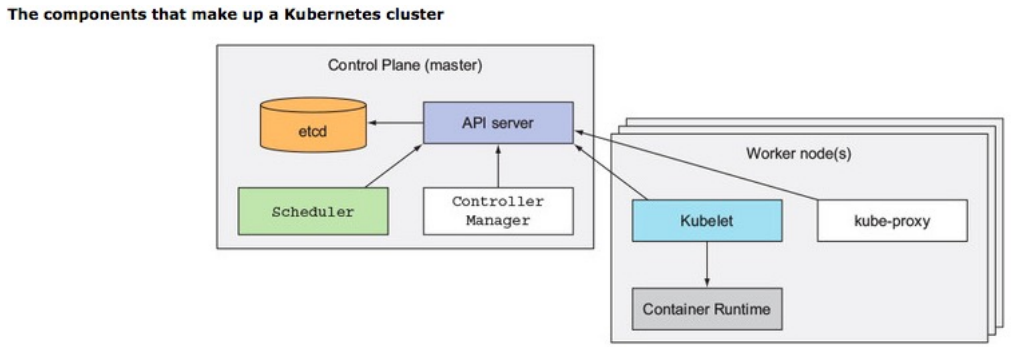
Linux Control Groups (cgroups)

- cgroups is a Linux kernel feature that limits the resource usage of a process.
- A process cannot use more than the configured amount of CPU, memory, network bandwidth, and so on.
- cgroups prevent processes from hogging resources reserved for other processes which is similar to when each process runs on a separate machine.

Architecture of a Kubernetes cluster

Kubernetes cluster is composed of many nodes, which can be split into two types:

- The master node, which hosts the Kubernetes Control Plane that controls and manages the whole Kubernetes system
- Worker nodes that run the actual applications you deploy



Master Node or Control Plane

- The Control Plane is what controls the cluster and makes it function. It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability. These components are
  - The Kubernetes API Server, which you and the other Control Plane components communicate with
  - The Scheduler, which schedules your apps (assigns a worker node to each deployable component of your application)
  - The Controller Manager, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures, and so on
  - etcd, a reliable distributed data store that persistently stores the cluster configuration.
- The components of the Control Plane hold and control the state of the cluster, but they don't run your applications. This is done by the (worker) nodes.

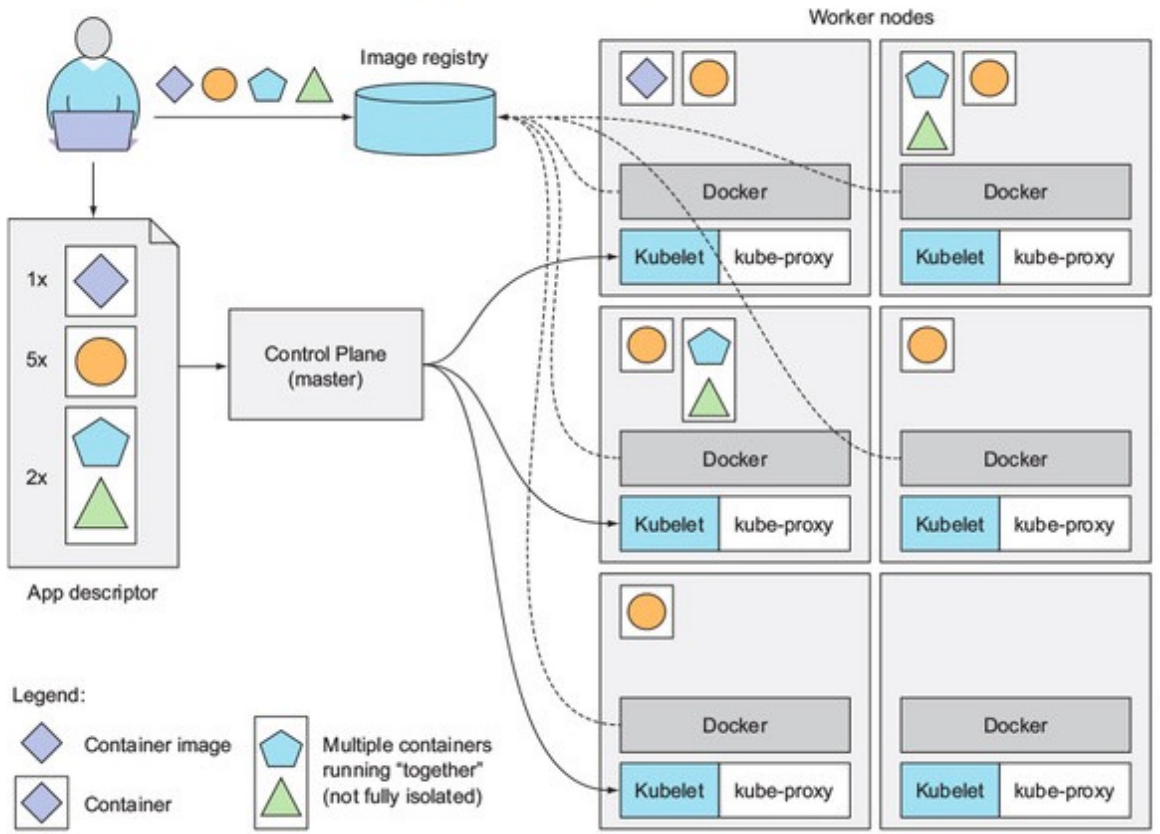
Worker Nodes

The worker nodes are the machines that run your containerized applications. The task of running, monitoring, and providing services to your applications is done by the following components:

- Docker, rkt, or another container runtime, which runs your containers
- The Kubelet, which talks to the API server and manages containers on its node
- The Kubernetes Service Proxy (kube-proxy), which load-balances network traffic between application components

Kubernetes & Applications

A basic overview of the Kubernetes architecture and an application running on top of it



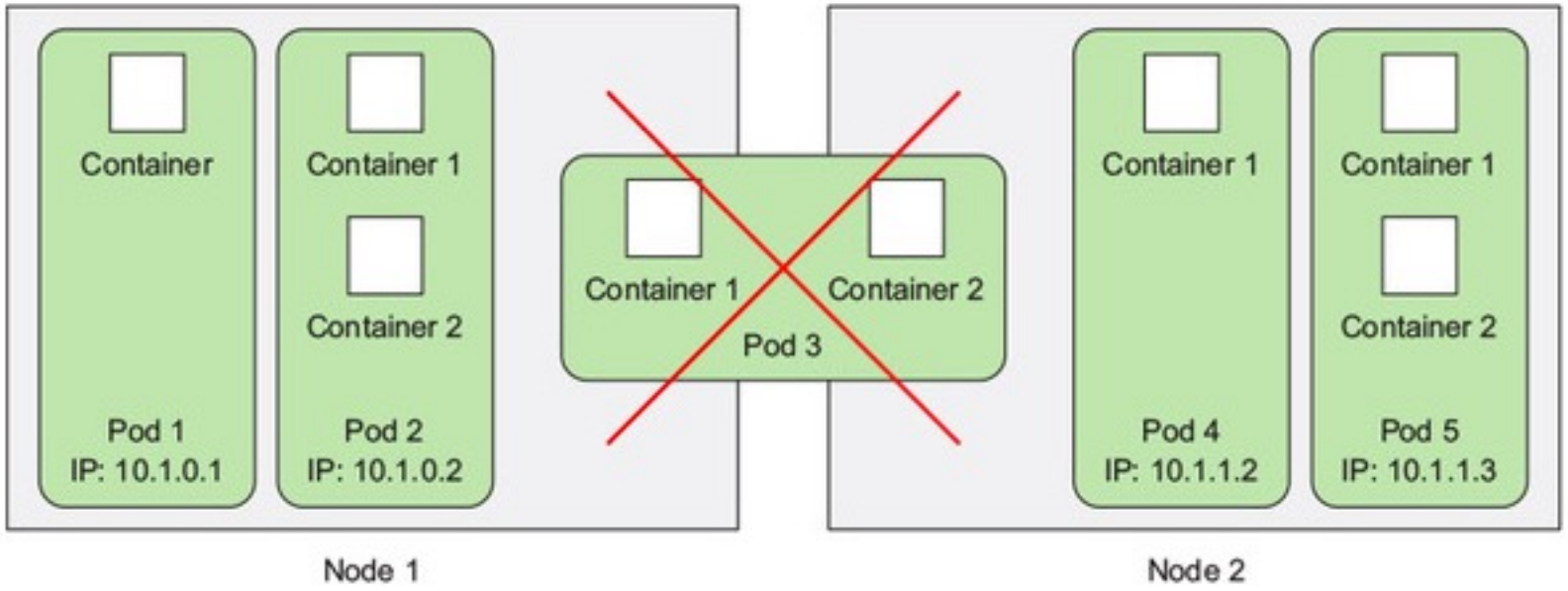
Kubernetes Core Concepts

Pods

- A pod is a co-located group of containers and represents the basic building block in Kubernetes.
- Instead of deploying containers individually, you always deploy and operate on a pod of containers.
- A pod does NOT always includes more than one container—it’s common for pods to contain only a single container.
- The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes.

Pods

All containers of a pod run on the same node. A pod never spans two nodes.

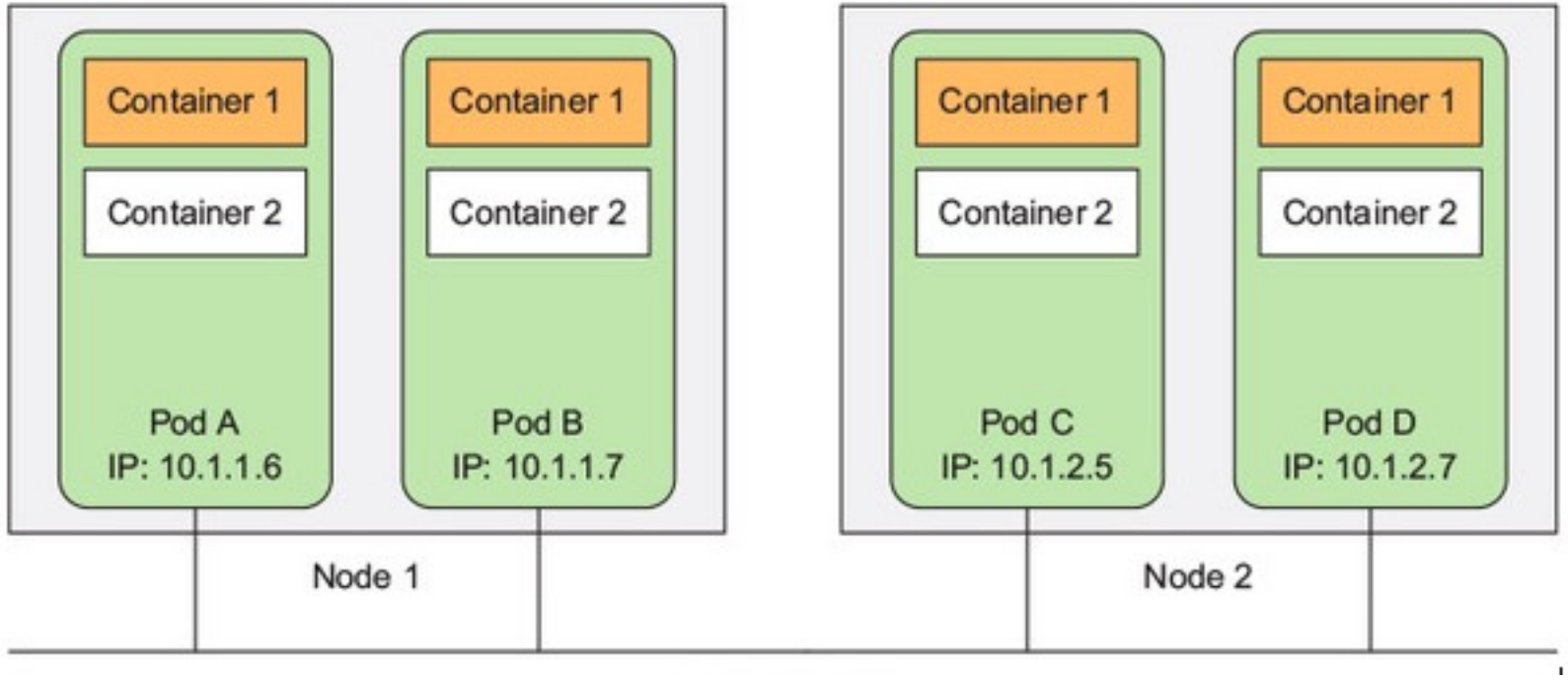


Why do we need pods?

- A pod of containers allows you to run closely related processes together and provide them with (almost) the same environment as if they were all running in a single container, while keeping them somewhat isolated.
- You want containers inside each group to share certain resources, although not all, so that they’re not fully isolated.
- Kubernetes achieves this by configuring Docker to have all containers of a pod share the same set of Linux namespaces instead of each container having its own set.

Flat inter-pod network

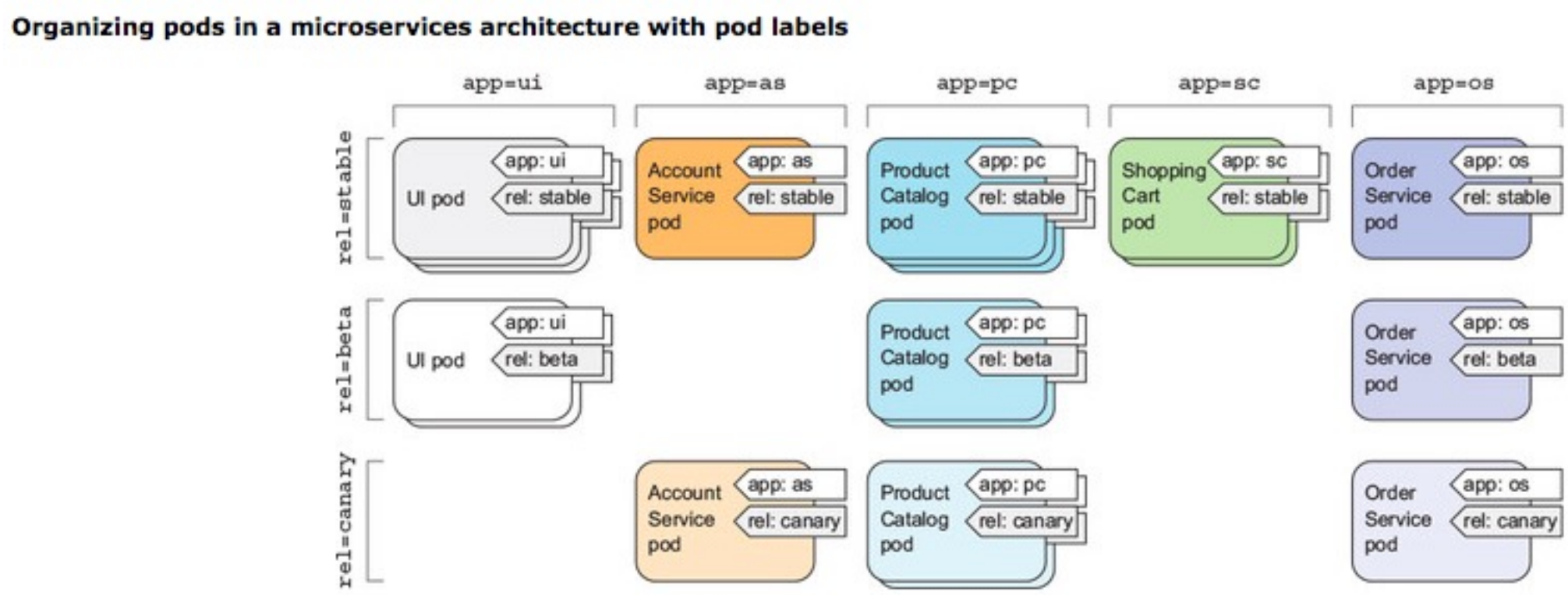
Each pod gets a routable IP address and all other pods see the pod under that IP address.



Organizing pods with labels

- Labels are a simple, yet incredibly powerful, Kubernetes feature for organizing not only pods, but all other Kubernetes resources.
- A label is an arbitrary key-value pair you attach to a resource, which is then utilized when selecting resources using label selectors.
- A resource can have more than one label, as long as the keys of those labels are unique within that resource.

Organizing pods with labels



Using namespaces to group resources

- Kubernetes namespaces provide a scope for objects names.
- Instead of having all your resources in one single namespace, you can split them into multiple name-spaces, which also allows you to use the same resource names multiple times (across different namespaces).
- Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups.
- Namespaces can also be used for separating resources in a multi-tenant environment, splitting up resources into production, development, and QA environments, or in any other way you may need.
- Resource names only need to be unique within a namespace.
- Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don’t provide any kind of isolation of running objects.

Keeping Pods Healthy

- Pods represent the basic deployable unit in Kubernetes.
- In real-world use cases, you want your deployments to stay up and running automatically and remain healthy without any manual intervention. To do this, you almost never create pods directly.
- Instead, you create other types of resources, such as ReplicationControllers or Deployments, which then create and manage the actual pods.

Liveness Probes

- Kubernetes can check if a container is still alive through liveness probes.
- You can specify a liveness probe for each container in the pod's specification.
- Kubernetes will periodically execute the probe and restart the container if the probe fails.

Liveness Probes

- Kubernetes can probe a container using one of the three mechanisms:
  - An **HTTP GET** probe performs an HTTP GET request on the container's IP address, a port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is 2xx or 3xx), the probe is considered successful. If the server returns an error response code or if it doesn't respond at all, the probe is considered a failure and the container will be restarted as a result.
  - A **TCP Socket** probe tries to open a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the container is restarted.
  - An **Exec** probe executes an arbitrary command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. All other codes are considered failures.

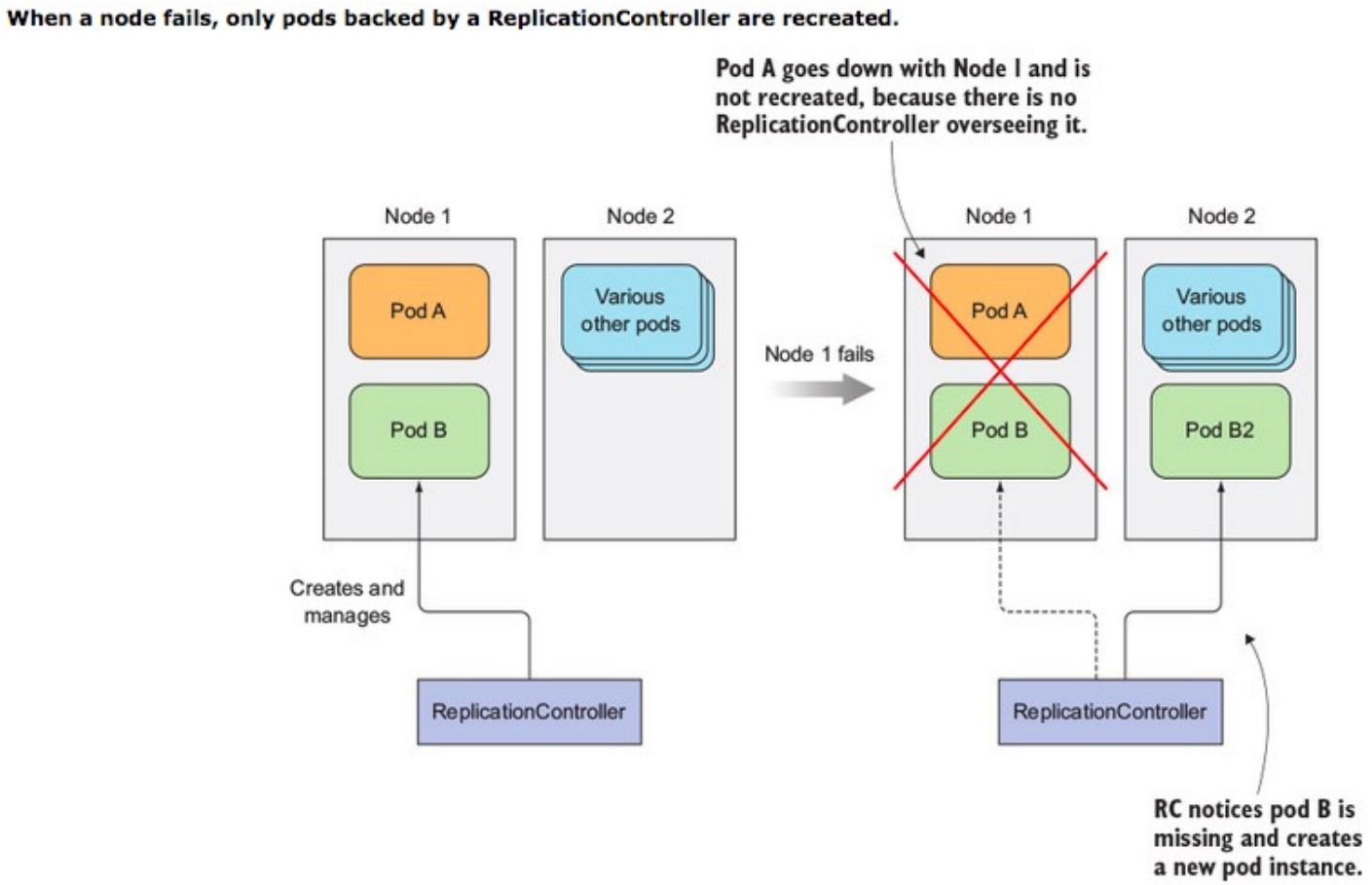
Effective Liveness Probes

- Liveness probe should checks if the server is responding.
- Liveness probes shouldn’t use too many computational resources and shouldn’t take too long to complete. By default, the probes are executed relatively often and are only allowed one second to complete.
- Kubernetes will retry the probe several times before considering it a single failed attempt so you do not need to implement retry logic in your probes.

ReplicationControllers

- A ReplicationController is a Kubernetes resource that ensures its pods are always kept running.
- If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.

ReplicationControllers



Operation of a ReplicationController

- A ReplicationController constantly monitors the list of running pods and makes sure the actual number of pods of a “type” always matches the desired number.
- If too few such pods are running, it creates new replicas from a pod template.
- If too many such pods are running, it removes the excess replicas.

ReplicationControllers

A ReplicationController has three essential parts

1. A label selector, which determines what pods are in the ReplicationController's scope
2. A replica count, which specifies the desired number of pods that should be running
3. A pod template, which is used when creating new pod replicas

ReplicaSets

- Initially, ReplicationControllers were the only Kubernetes component for replicating pods and rescheduling them when nodes failed.
- Later, a similar resource called a ReplicaSet was introduced. It's a new generation of ReplicationController and replaces it completely.
- A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors. Whereas a ReplicationController's label selector only allows matching pods that include a certain label, a ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.

DaemonSets

- Both ReplicationControllers and ReplicaSets are used for running a specific number of pods deployed anywhere in the Kubernetes cluster. But certain cases exist when you want a pod to run on each and every node in the cluster.
- Those cases include infrastructure-related pods that perform system-level operations. For example, you'll want to run a log collector and a resource monitor on every node. Another good example is Kubernetes' own kube-proxy process, which needs to run on all nodes to make services work.

Using a DaemonSet to run a pod on every node

- To run a pod on all cluster nodes, you create a DaemonSet object, which is much like a ReplicationController or a ReplicaSet, except that pods created by a DaemonSet already have a target node specified and skip the Kubernetes Scheduler.
- A DaemonSet makes sure it creates as many pods as there are nodes and deploys each one on its own node.
- Whereas a ReplicaSet or ReplicationController makes sure that a desired number of pod replicas exist in the cluster, a DaemonSet doesn’t have any notion of a desired replica count. It doesn’t need it because its job is to ensure that a pod matching its pod selector is running on each node.

Job Resource

- You'll have cases where you only want to run a task that terminates after completing its work.
- ReplicationControllers, ReplicaSets, and DaemonSets run continuous tasks that are never considered completed. Processes in such pods are restarted when they exit. But in a completable task, after its process terminates, it should not be restarted again.
- Kubernetes includes support for this through the Job resource, which is similar to the other resources we've discussed in this chapter, but it allows you to run a pod whose container isn't restarted when the process running inside finishes successfully. Once it does, the pod is considered complete.
- An example of such a job would be if you had data stored somewhere and you needed to transform and export it somewhere.

Services: Enabling clients to discover and talk to pods

Pods need a way of finding other pods if they want to consume the services they provide. Unlike in the non-Kubernetes world, where a sysadmin would configure each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because

- Pods are ephemeral—They may come and go at any time, whether it's because a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed.
- Kubernetes assigns an IP address to a pod after the pod has been scheduled to a node and before it's started—Clients thus can't know the IP address of the server pod up front.
- Horizontal scaling means multiple pods may provide the same service—Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.

Services

- A Kubernetes Service is a resource you create to make a single, constant point of entry to a group of pods providing the same service.
- Each service has an IP address and port that never change while the service exists.
- Clients can open connections to that IP and port, and those connections are then routed to one of the pods backing that service.
- This way, clients of a service don’t need to know the location of individual pods providing the service, allowing those pods to be moved around the cluster at any time.

Service Discovery

- Discovering services through environment variables
- Discovering services through DNS
- Connecting to the service through its FQDN

Readiness Probes

- Liveness probes help keep your apps healthy by ensuring unhealthy containers are restarted automatically.
- Similar to liveness probes, Kubernetes allows you to also define a readiness probe for your pod.
- The readiness probe is invoked periodically and determines whether the specific pod should receive client requests or not.

Readiness Probe Types

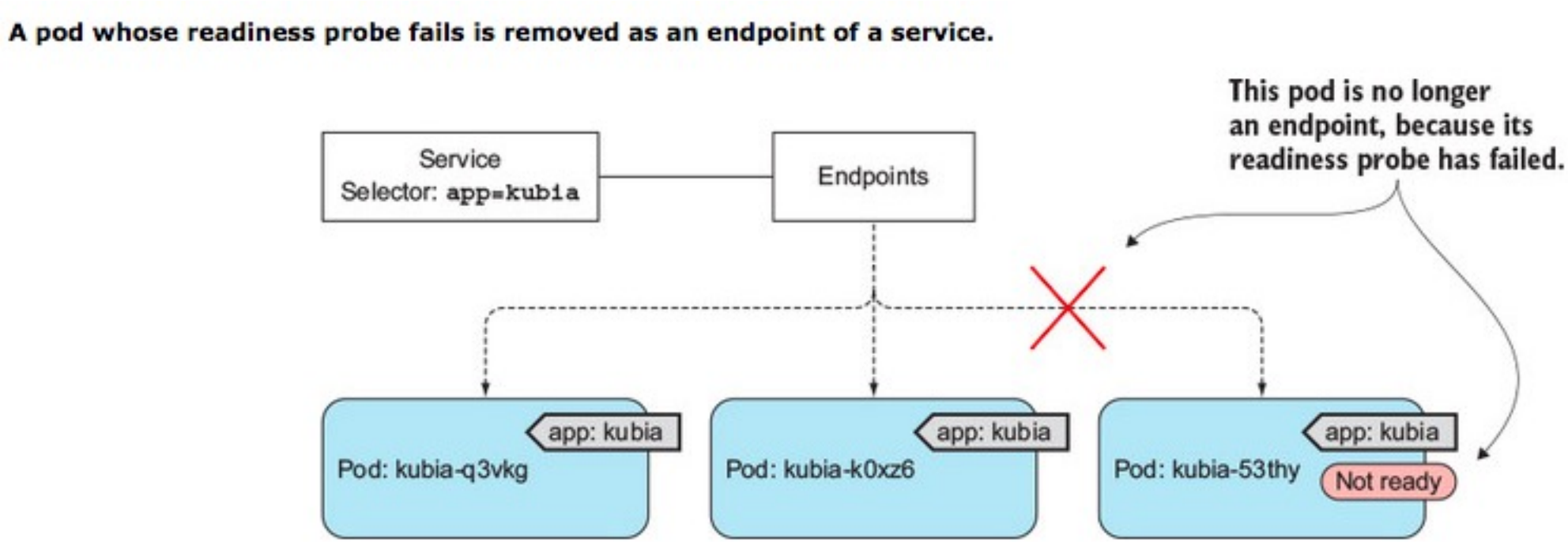
Like liveness probes, three types of readiness probes exist:

1. An **Exec** probe, where a process is executed. The container's status is determined by the process' exit status code.
2. An **HTTP** GET probe, which sends an HTTP GET request to the container and the HTTP status code of the response determines whether the container is ready or not.
3. A **TCP Socket** probe, which opens a TCP connection to a specified port of the container. If the connection is established, the container is considered ready.

Understanding the operation of readiness probes

- When a container is started, Kubernetes can be configured to wait for a configurable amount of time to pass before performing the first readiness check. After that, it invokes the probe periodically and acts based on the result of the readiness probe. If a pod reports that it's not ready, it's removed from the service. If the pod then becomes ready again, it's re-added.
- Unlike liveness probes, if a container fails the readiness check, it won't be killed or restarted. This is an important distinction between liveness and readiness probes.
- Liveness probes keep pods healthy by killing off unhealthy containers and replacing them with new, healthy ones.
- Readiness probes make sure that only pods that are ready to serve requests receive them.

Understanding the operation of readiness probes

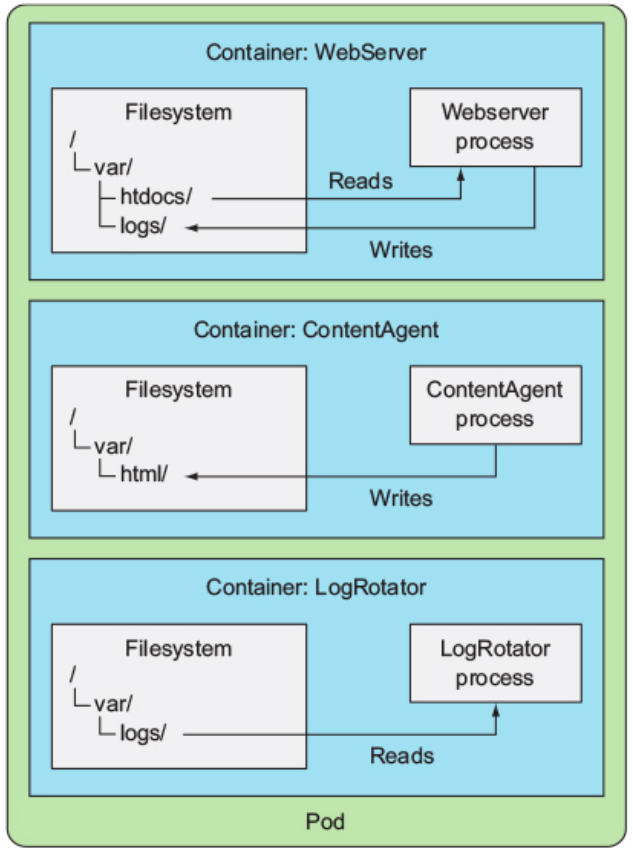


Volumes: attaching disk storage to containers

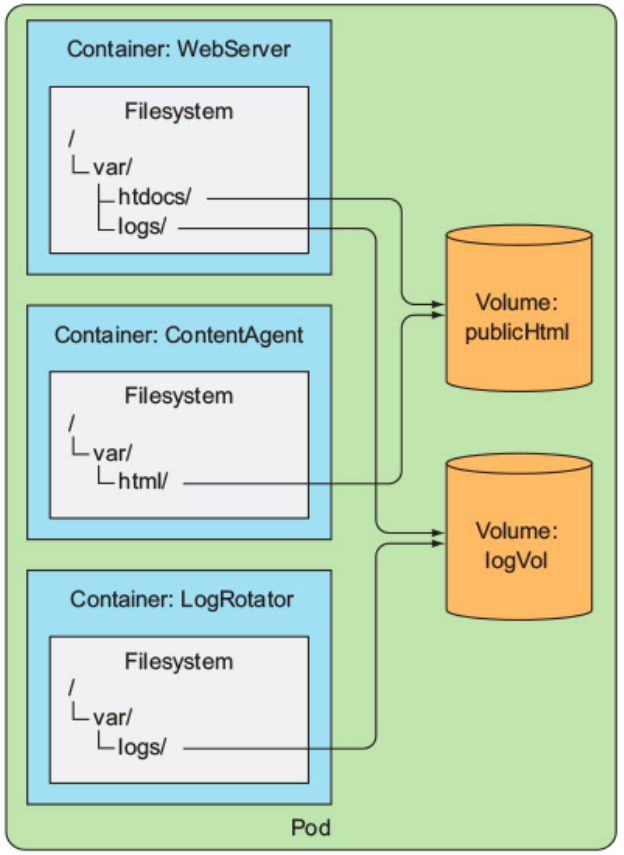
- Container in a pod has its own isolated filesystem, because the file-system comes from the container's image.
- Every new container starts off with the exact set of files that was added to the image at build time.
- In certain scenarios you want the new container to continue where the last one finished, such as when restarting a process on a physical machine. You may not need (or want) the whole filesystem to be persisted, but you do want to preserve the directories that hold actual data.
- Kubernetes provides this by defining storage **volumes**.
- Storage volumes aren't top-level resources like pods, but are instead defined as a part of a pod and share the same lifecycle as the pod.
- A volume is created when the pod is started and is destroyed when the pod is deleted. Because of this, a volume's contents will persist across container restarts. After a container is restarted, the new container can see all the files that were written to the volume by the previous container. Also, if a pod contains multiple containers, the volume can be used by all of them at once.

Volumes

Three containers of the same pod without shared storage



Three containers sharing two volumes mounted at various mount paths



Volume Types

- **emptyDir**—A simple empty directory used for storing transient data.
- **hostPath**—Used for mounting directories from the worker node's filesystem into the pod.
- **nfs**—An NFS share mounted into the pod.
- **gcePersistentDisk** (Google Compute Engine Persistent Disk)
- **awsElasticBlockStore** (Amazon Web Services Elastic Block Store Volume)
- **azureDisk** (Microsoft Azure Disk Volume)—Used for mounting cloud provider-specific storage.
- cinder, cephfs, iscsi, flocker, glusterfs, quobyte, rbd, flexVolume, vsphere-Volume, photonPersistentDisk, scaleIO—Used for mounting other types of network storage.
- **configMap**, **secret**, **downwardAPI**—Special types of volumes used to expose certain Kubernetes resources and cluster information to the pod.
- **persistentVolumeClaim**—A way to use a pre or dynamically provisioned persistent storage.

ConfigMaps

- The whole point of an app's configuration is to keep the config options that vary between environments, or change frequently, separate from the application's source code.
- Kubernetes allows separating configuration options into a separate object called a ConfigMap, which is a map containing key/value pairs with the values ranging from short literals to full config files.
- The contents of the map are instead passed to containers as either environment variables or as files in a volume.
- Environment variables can also be referenced in command-line arguments using the $(ENV_VAR) syntax, you can also pass ConfigMap entries to processes as command-line arguments.

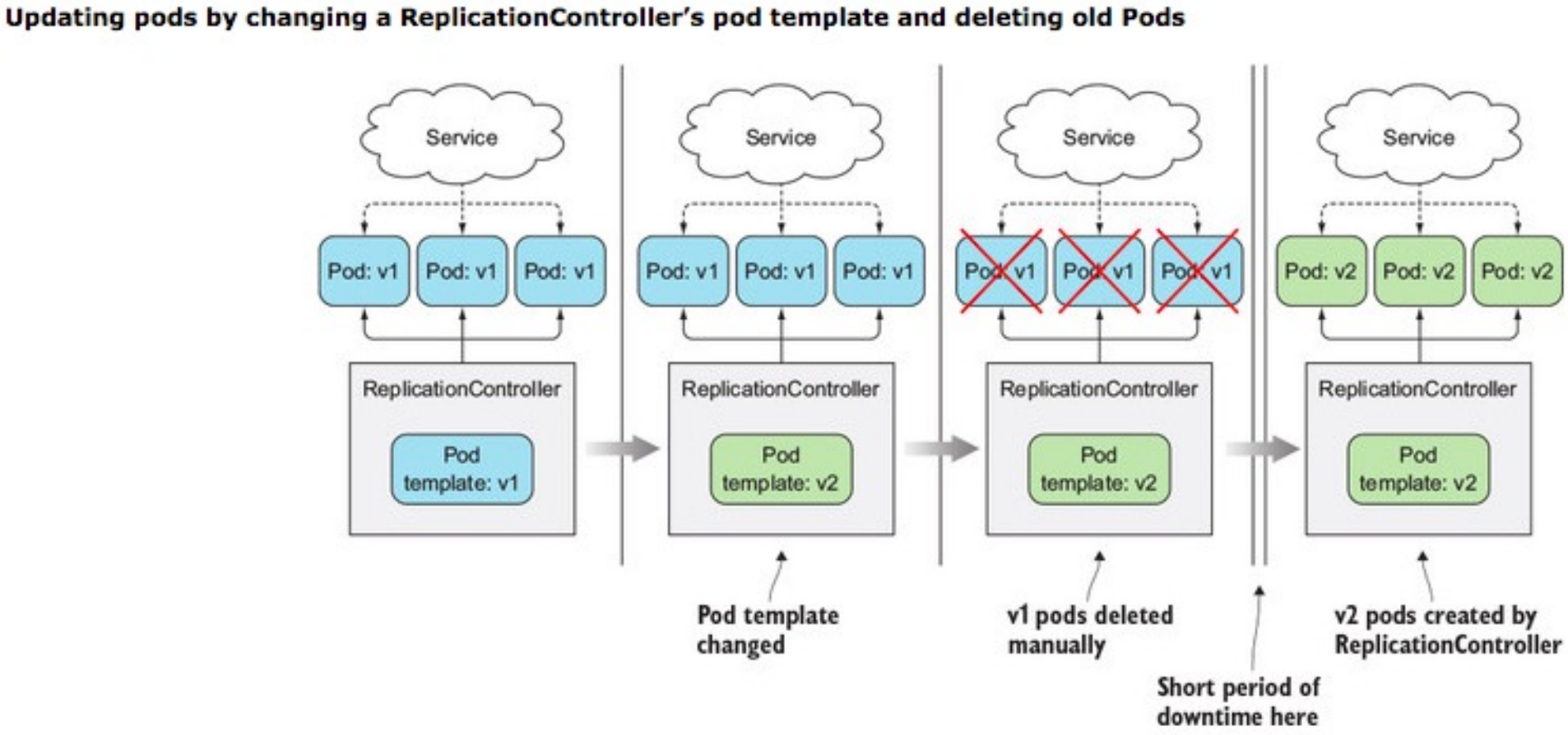
Secrets

- Config data usually also includes sensitive information, such as credentials and private encryption keys, which need to be kept secure.
- To store and distribute such information, Kubernetes provides a separate object called a Secret. Secrets are much like ConfigMaps—they're also maps that hold key-value pairs. They can be used the same way as a ConfigMap.
- Kubernetes helps keep your Secrets safe by making sure each Secret is only distributed to the nodes that run the pods that need access to the Secret. Also, on the nodes themselves, Secrets are always stored in memory and never written to physical storage, which would require wiping the disks after deleting the Secrets from them.
- Secrets are stored in encrypted on the master node, more specifically in etcd.

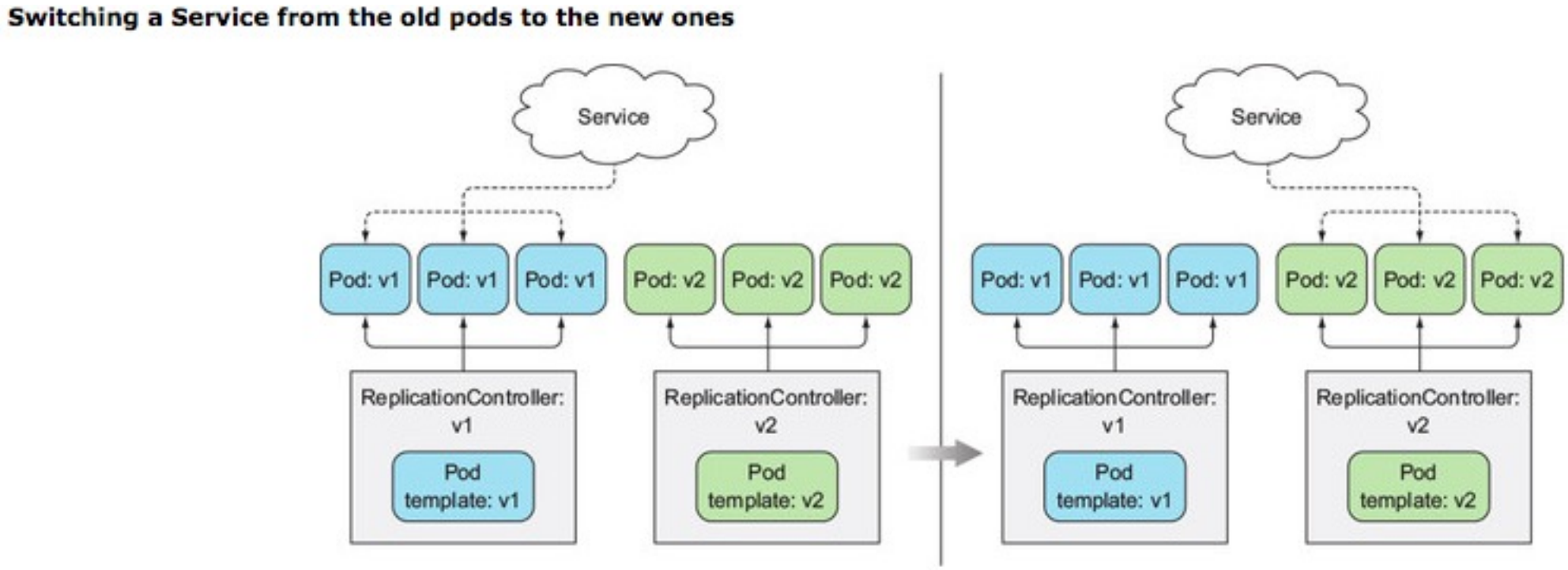
Deployments: updating applications declaratively

- Replacing pods with newer versions
- Updating managed pods
- Updating pods declaratively using Deployment resources
- Performing rolling updates
- Automatically blocking rollouts of bad versions
- Controlling the rate of the rollout
- Reverting pods to a previous version

Deleting old pods and replacing them with new ones

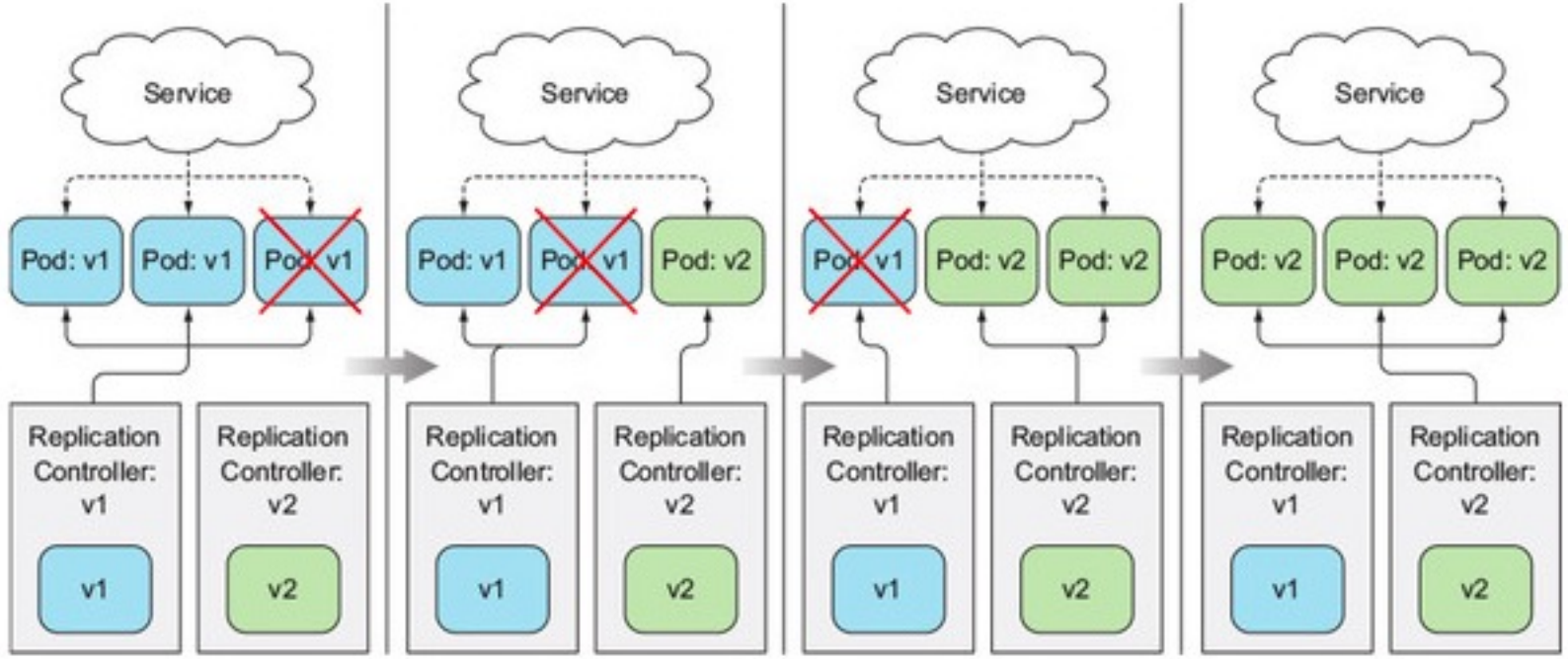


Spinning up new pods and then deleting the old ones



Rolling Update

A rolling update of pods using two ReplicationControllers



StatefulSets

StatefulSets are intended to be used with stateful applications and distributed systems.

Pets vs. Cattle

- We tend to treat our app instances as pets, where we give each instance a name and take care of each instance individually. But it's usually better to treat instances as cattle and not pay special attention to each individual instance. This makes it easy to replace unhealthy instances without giving it a second thought, similar to the way a farmer replaces unhealthy cattle.
- Instances of a stateless app, for example, behave much like heads of cattle. It doesn't matter if an instance dies—you can create a new instance and people won't notice the difference.
- On the other hand, with stateful apps, an app instance is more like a pet. When a pet dies, you can't go buy a new one and expect people not to notice. To replace a lost pet, you need to find a new one that looks and behaves exactly like the old one. In the case of apps, this means the new instance needs to have the same state and identity as the old one.

StatefulSets vs. ReplicaSet

- Pod replicas managed by a ReplicaSet or ReplicationController are much like cattle. Because they're mostly stateless, they can be replaced with a completely new pod replica at any time.
- When a stateful pod instance dies (or the node it's running on fails), the pod instance needs to be resurrected on another node, but the new instance needs to get the same name, network identity, and state as the one it's replacing. This is what happens when the pods are managed through a StatefulSet.

Additional Resources

See Lecture Page