Container Orchestration Using Kubernetes

Workloads

A Day in the Life of an DevOps Engineer

You are working as a DevOps Engineer in an organization, and you have been asked to design a solution to create a highly available MySQL and WordPress deployment using autoscaling. All the users should be added using ConfigMaps and the sensitive data should be added using Secrets.

You also need to ensure that the service is running on the NodePort and the WordPress pod should not deploy if the MySQL service is not deployed.

To achieve all the above, along with some additional concepts, we would be learning a few concepts in this lesson that will help you find a solution for the above scenario.

Learning Objectives

By the end of this lesson, you will be able to:

- Construct Kubernetes pod to comprehend its lifecycle.
- Construct a namespace with a resource quota for managing resources within a Kubernetes container.
- Deploy a multitier application to understand deployment process in Kubernetes.
- Design and implement a horizontal pod autoscaler to achieve scalability.
- Construct a Kubernetes ReplicaSet to achieve high avaibility

TECHNOLOGY

Overview of Workloads

Overview

A workload is an application running on Kubernetes.

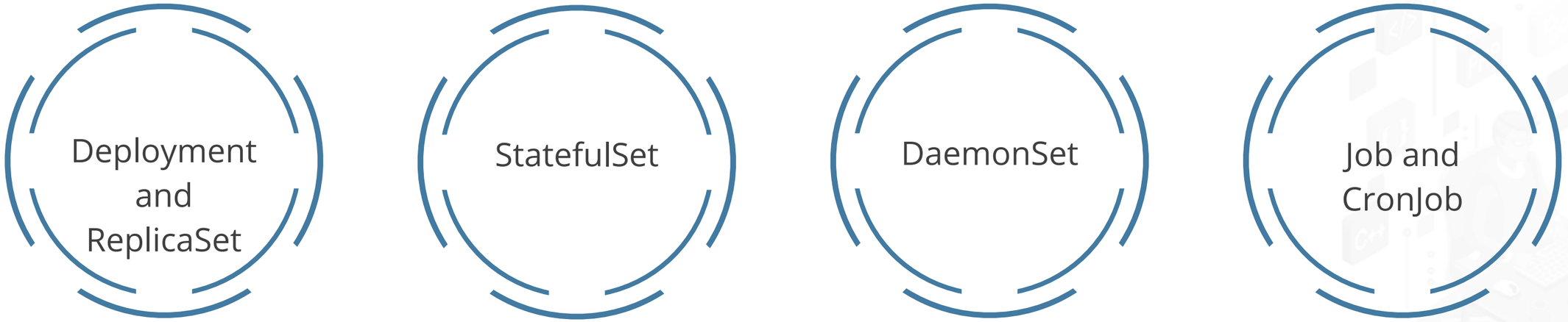
Whether the workload has a single or several components that work together, it runs inside a set of pods.

When a pod is running in a cluster, a critical fault on the node fails all pods on that node.

Kubernetes treats that level of failure as final. It creates a new pod to recover, even if the node becomes healthy later.

Overview

Kubernetes provides several built-in workload resources:



Users can add in a third-party workload resource if they want a specific behavior that is not part of Kubernetes' core.

Understanding Pods

Introduction

Pods are the smallest deployable units of computing that users can create and manage in Kubernetes.

A pod is a group of one or more containers with shared storage and network resources and a specification for how to run the Containers.

Its contents are always co-located, co-scheduled, and run in a shared context.

It models an application-specific **logical host**. It contains one or more application containers that are relatively tightly coupled.

Using Pods

Use Deployment or Job for workloads instead of making pods directly, and choose **StatefulSet** for stateful pods.

Users use pods in a Kubernetes cluster in two ways:

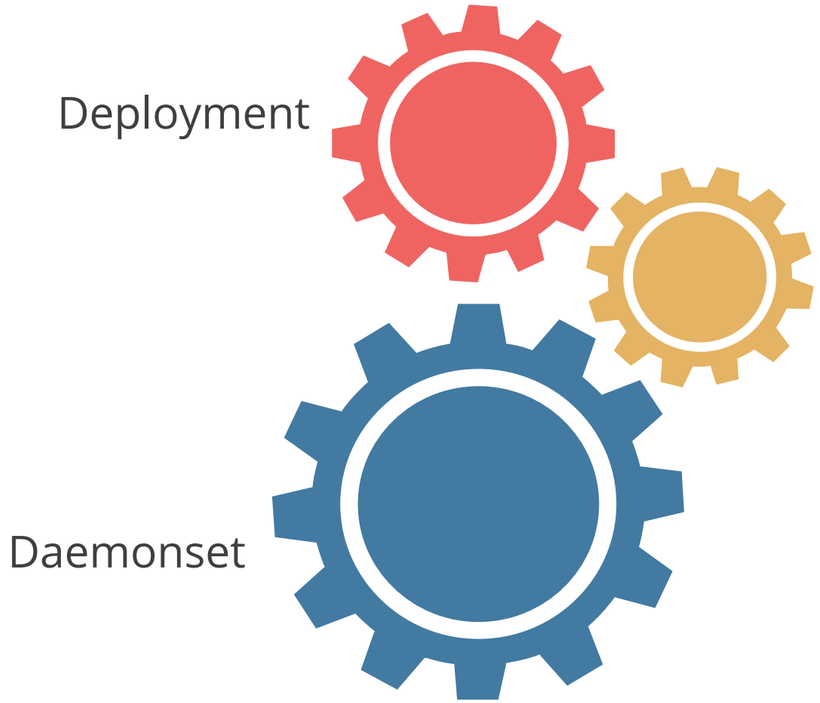
pods that run a single container

pods that run multiple containers that need to work together

Pods and Controllers

New pods are assigned to a cluster node, with a controller managing their replication, updates, and recovery from failures.

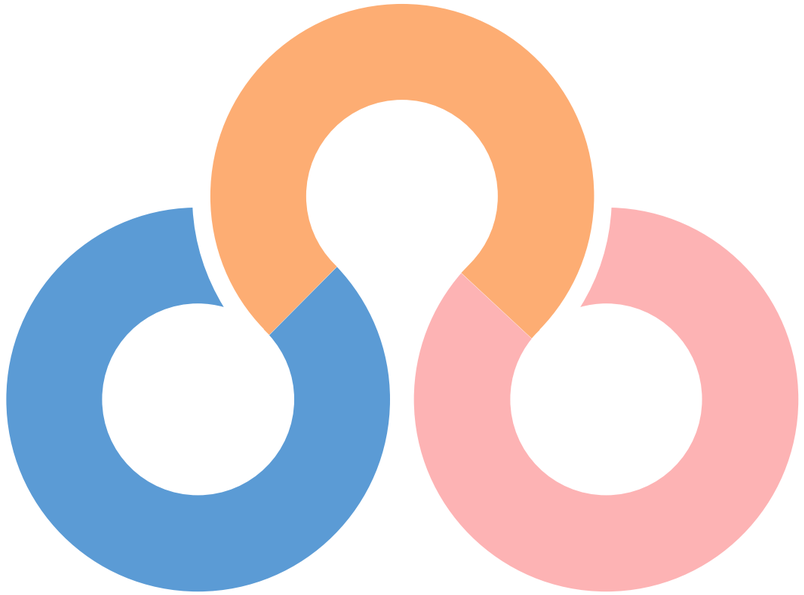
Some examples of workload resources that manage one or more pods:



Pods and Controllers

Each workload resource implements its own rules for handling changes to the pod template.

Modifying the pod template or switching to a new pod template has no direct effect on the pods that already exist.



The **Kubelet** does not directly observe or manage any of the details around pod templates and updates.

Pod Templates

Pod templates are specifications that help users to create pods. They are included in workload resources such as **Deployments**, **Jobs**, and **DaemonSets.**

The sample below is a basic pod manifest with a single-container template.

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
# Name of the pod
spec:
  containers:
# List of Containers
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Pod Update and Replacement

Pod update operations like **patch** and **replace** have some limitations:

- The metadata about a pod is immutable.
- If the **metadata.deletionTimestamp** is set, no new entry can be added to the metadata.finalizers list.
- pod updates may not change fields other than **spec.containers[*].image, spec.initContainers[*].image, spec.activeDeadlineSeconds,** or **spec.tolerations**.
- Only two types of updates are allowed while updating the **spec.activeDeadlineSeconds** field.

Resource Sharing and Communication

Pods enable data sharing and communication among their constituent containers.

Storage in pods

All containers in the pod can access the shared volumes, allowing those containers to share data.

pod networking

Within a pod, containers share an IP address and port space and can find each other via localhost.

Privileged Mode for Containers

Any container in a pod can enable Privileged Mode using the privileged flag on the security context of the container spec.

This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

Pod Lifecycle

Pod Lifecycle

Pods progress from a Pending to Running state, and end in Succeeded or Failed based on container success.

Kubernetes tracks different container statuses within a pod and decides what action to take to restore the pod's health.

In the Kubernetes API, pods have both a specification and an actual status.

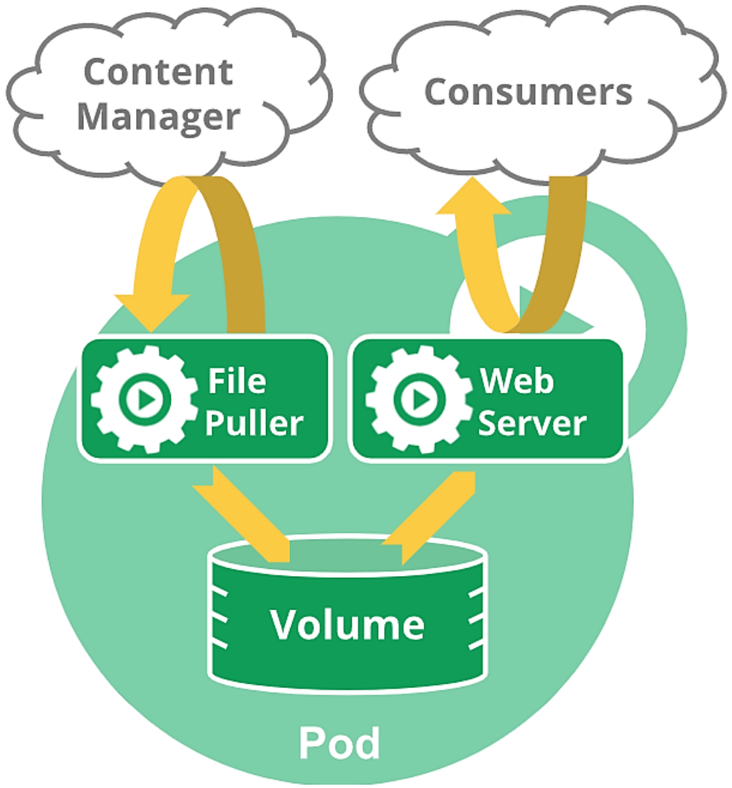
Pods are Ephemeral

Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion.

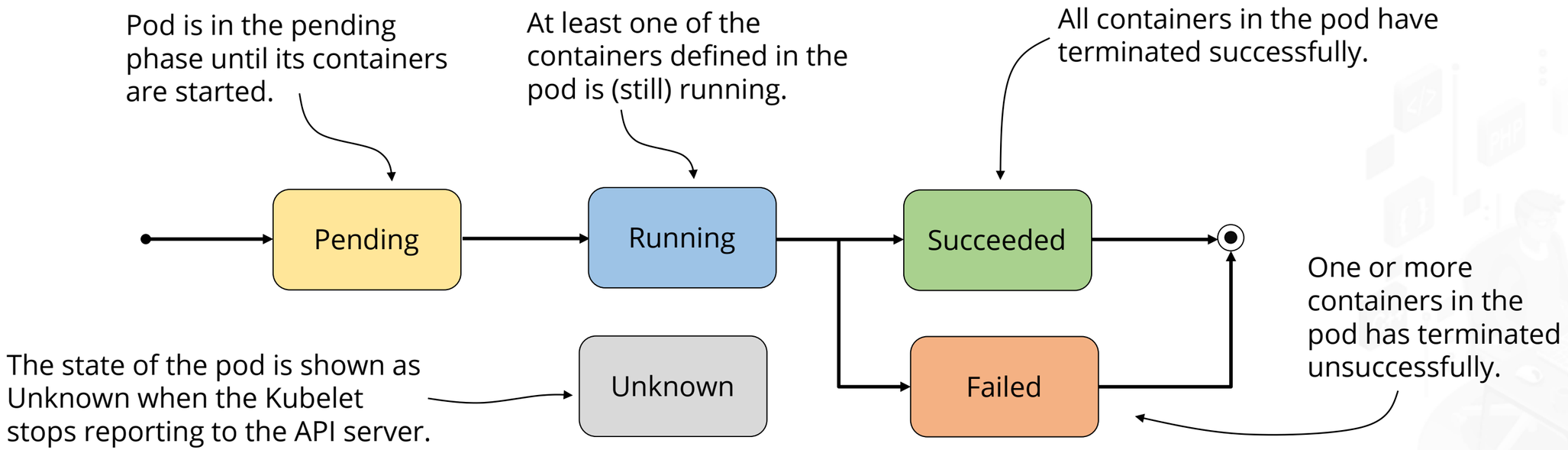
- If a node dies, the pods scheduled to that node are scheduled for deletion after a timeout period.
- Kubernetes uses a higher-level abstraction called controller, which handles the work of managing the relatively disposable pod instances.
- A given pod (as defined by a UID) is never **rescheduled** to a different Node.
- A pod can be replaced by a new, near-identical pod with even the same name, if desired but with a different UID.

Pods and Volumes

A multi-container pod contains a file puller and a web server that use a persistent volume for shared storage between the containers.



Phase of a Pod



Phase of a Pod

A pod phase represents its stage in the lifecycle, indicated by the 'phase' field in its podStatus object. Possible phase values include:

Value	Description
Pending	The pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run.
Running	The pod has been bound to a node, and all the containers have been created.
Succeeded	All containers in the pod have terminated successfully and will not be restarted.

Pod Phase

When a node in a Kubernetes cluster fails or loses connection, Kubernetes marks all pods on that node as failed. Possible pod phases include:

Value	Description
Failed	All containers in the pod have terminated, and at least one container has terminated in failure.
Unknown	This phase typically occurs due to an error in communicating with the node where the pod should be running.

Container States

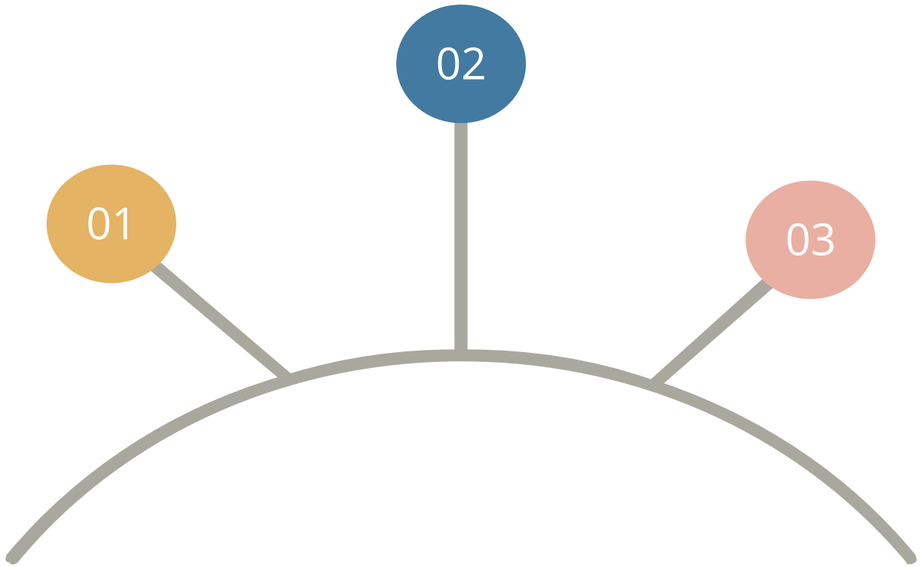
Kubernetes tracks the state of each container inside a pod. Each state has a specific meaning:

Running

The Running status indicates that a container is executing without issues.

Waiting

If a container is not in either the Running or Terminated state, it is Waiting.



Terminated

A container in the terminated state began execution and then either ran to completion or failed for some reason.

Container Restart Policy

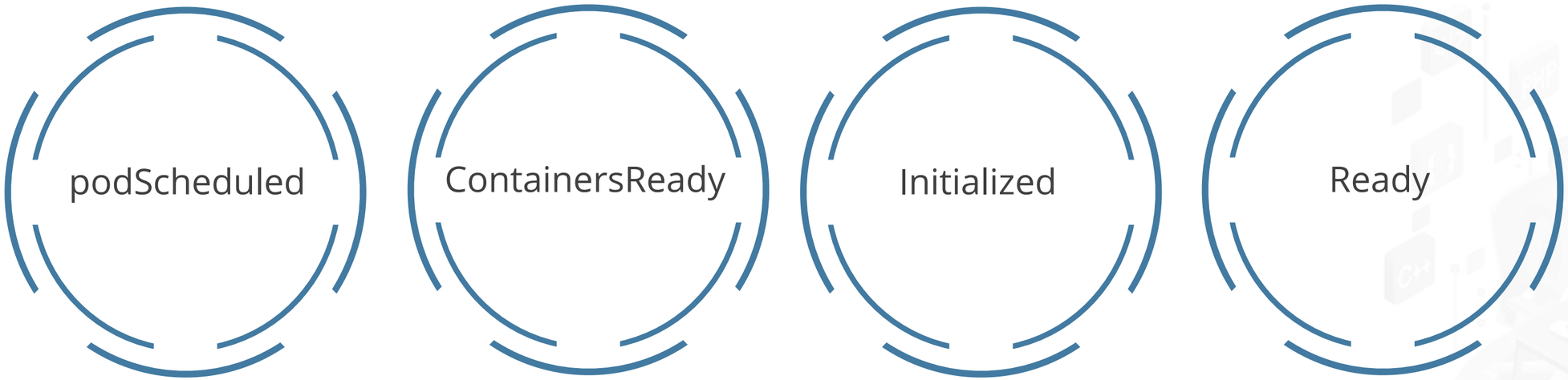
The **spec** of a pod has a **restartPolicy** field with possible values **Always, OnFailure,** and **Never**.

1. The **restartPolicy** applies to all containers in the pod.

2. **restartPolicy** only refers to restarts of the containers by the **kubelet** on the same node.

Pod Conditions

A pod has a **podStatus**, which has an array of **podConditions** through which the pod has passed or not. Following are the podConditions:



Pod Conditions

Field Name	Description
type	Represents the **name** of the pod condition
status	Indicates whether that condition is applicable, with possible values, such as **True**, **False**, or **Unknown**
lastProbeTime	Represents the **timestamp** of when the pod condition was **last probed**
lastTransitionTime	Represents the **timestamp** for when the pod **last transitioned** from one status to another
reason	Represents **machine-readable**, **UpperCamelCase** text indicating the reason for the condition's last transition
message	Represents **human-readable** message indicating details about the last status transition

Termination of Pods

The design aim is to be able to request deletion, know when processes terminate, and ensure that deletes are eventually complete.

If the kubelet or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start, including the full original grace period.

Forced Pod Termination

Forced deletions can be potentially disruptive for some workloads and their pods.

2. The control plane cleans up terminated pods when the number of pods exceeds the configured threshold.

1. The **kubectl delete** command supports the **--grace-period=<seconds>** option, which allows overriding the default values.

3. When a forced deletion is performed, the API server does not wait for confirmation from the kubelet.

Forced pod termination is useful when Kubernetes pods are stuck in a terminating state.

Understanding the Pod Lifecycle

Duration: 10 mins

Problem Statement:

You've been asked to create and describe a Kubernetes pod to comprehend its lifecycle

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create and describe a Kubernetes pod

Init Containers

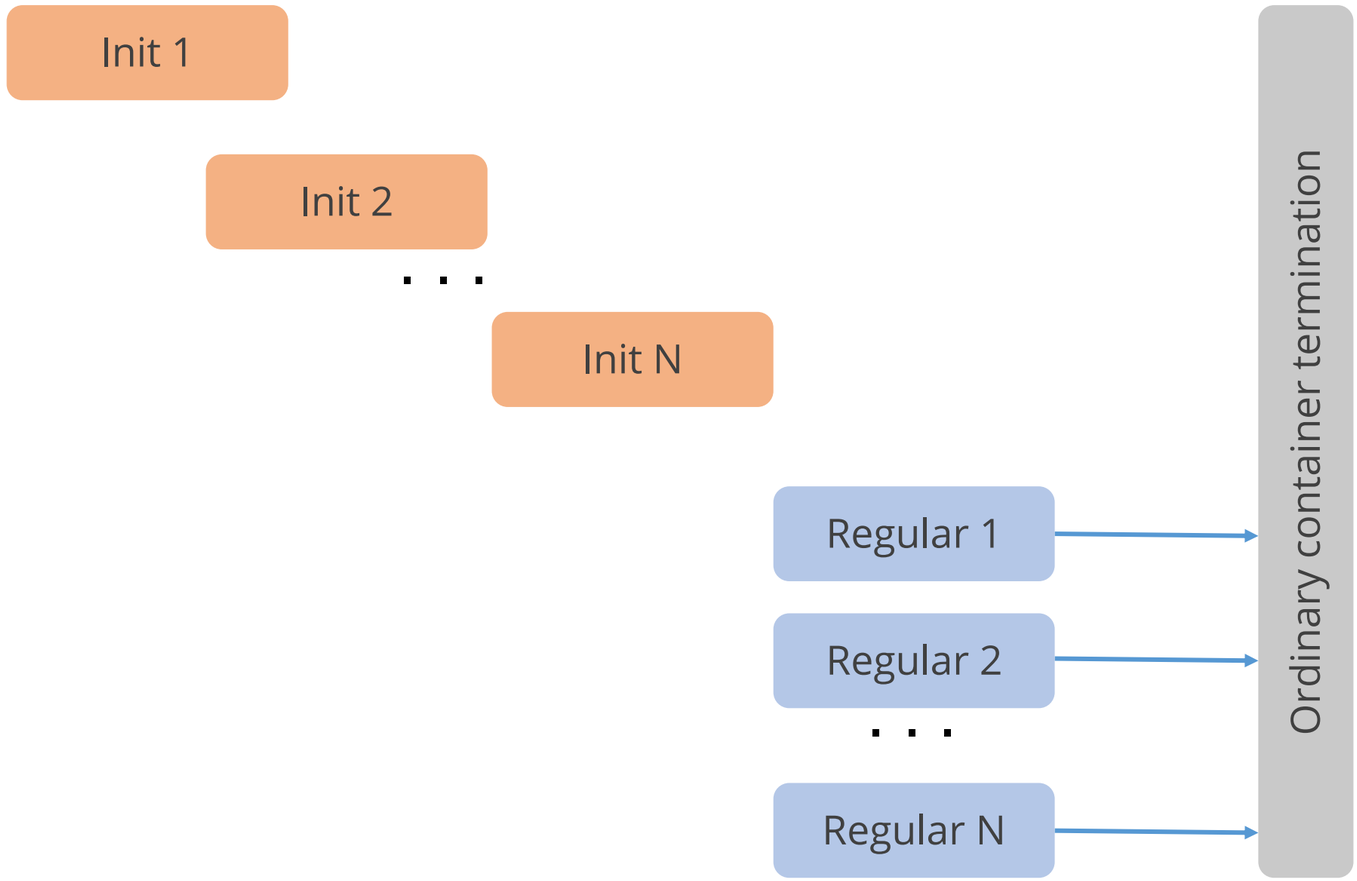
Introduction

Init containers are specialized containers that run before app containers in a pod.

Init containers can contain utilities or setup scripts that are not present in an app image.

Init Containers

A pod can have multiple containers running apps within it. It can also have one or more **init** containers that are run before the app containers are started.



Init Containers

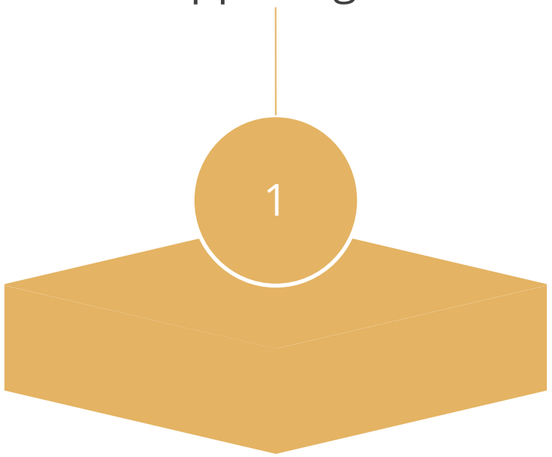
The differences between **regular** and **init** containers are as follows:

- Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings.
- Init containers do not support **lifecycle, livenessProbe, readinessProbe, or startupProbe** because they must run to completion before the pod can be ready.

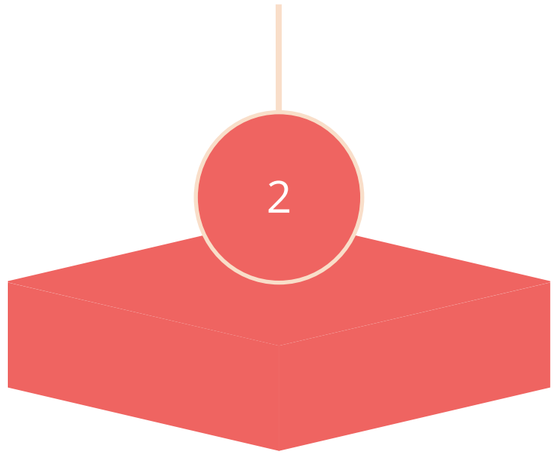
Init Containers

The advantages of **init** containers for start-up related code are:

Init containers can contain utilities or custom code for a setup that is not present in an app image.



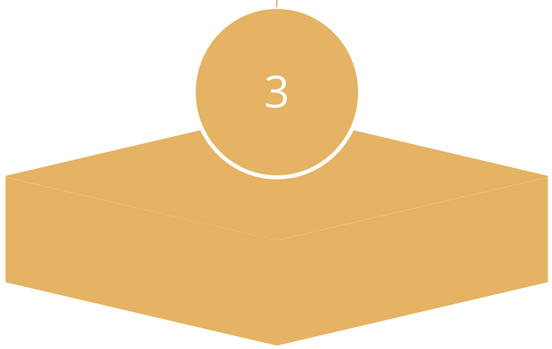
The application image builder and deployer roles can work independently without the need to jointly build a single app image.



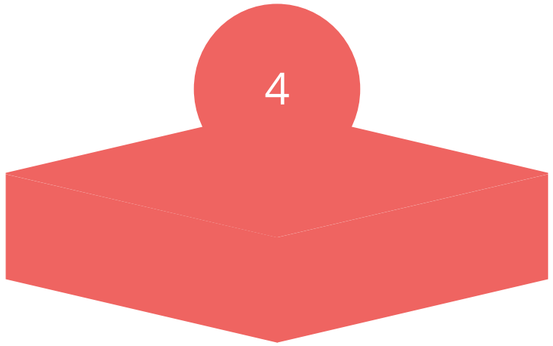
Init Containers

The advantages of **init** containers for start-up related code are:

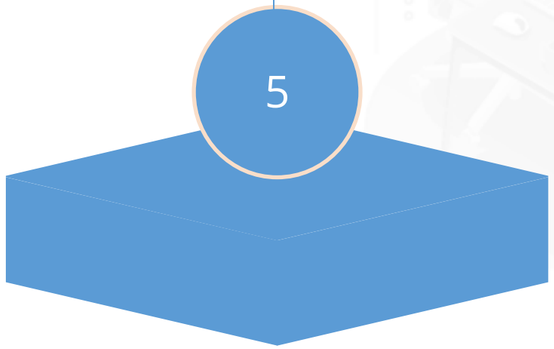
Init containers offer a mechanism to block or delay app container startup until a set of preconditions are met.



Init containers can run with a different view of the filesystem than app containers in the same pod.



Init containers can securely run utilities or custom code that would otherwise make an app container image less secure.



Init Containers in Use

This example defines a simple pod that has two **init** containers:

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  myapp-pod
  labels:
    app: myapp
  spec:
    containers:
    - name: myapp-container
      image: busybox:1.28
      command: ['sh','-c','echo The app os running! 66 sleep3600
   initContainers:
      - name: init01
      image: busybox:1.28
      command: ['sh', '-c' , "sleep 60; done"]
    - name: init02
      image: busybox:1.28
      command: ['sh', '-c', "sleep 120; done"]
```

Init Containers in Use

Start the pod by running:

```
kubectl apply -f myapp.yaml
```

The output is similar to this:

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY    STATUS     RESTARTS   AGE
myapp-pod   0/1      Init:0/2   0          6s
```

or for more details:

```
kubectl describe -f myapp.yaml
```

Configuring a pod Using Init Container

Duration: 15 mins

Problem Statement:

You've been asked to create and configure a pod using the init container to design more complex and flexible workflows for Kubernetes applications.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pod
2. Create the services
3. Verify the pod’s state

Managing Container Resources

Introduction

The most common resources to specify in a pod are CPU and memory (RAM).

When the resource **request** is specified for containers in a pod, the Kube-scheduler uses this information to decide on which node to place the pod.

When the resource **limit** is specified for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit set.

Requests and Limits

If the node where a pod is running has enough resources available, it is possible (and allowed) for a container to use more resources than requested.

01

If the memory limit is **4GiB** for a container, the kubelet (and container runtime) enforce the limit.

02

Limits can be implemented either reactively or by enforcement.

Resource Types

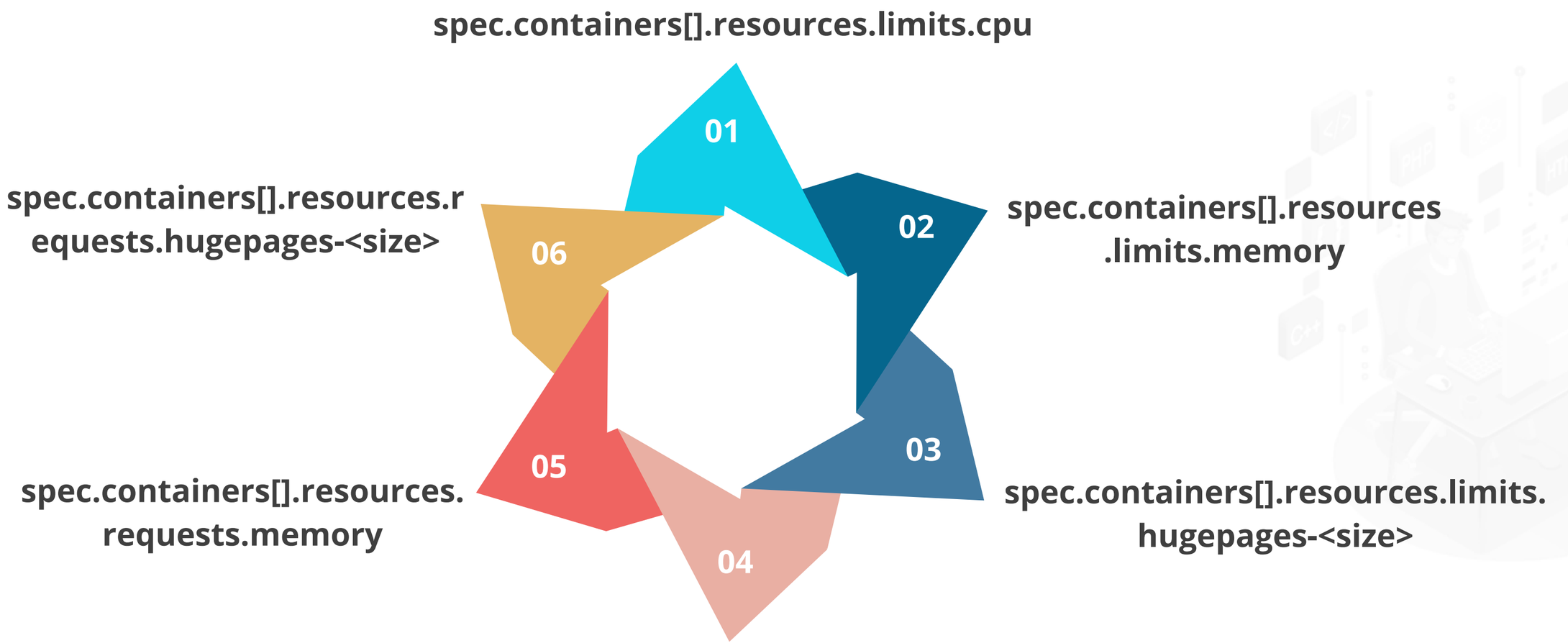
CPU and **memory** are both **resource types**.

CPU represents compute processing and is specified in units of Kubernetes CPUs.

CPU and memory are collectively referred to as **compute resources** or **resources**.

Resource Requests and Limits of Pod and Container

Each container of a pod can specify one or more of the following:



Resource Units in Kubernetes

CPU resource units

In Kubernetes, one CPU is equivalent to **one physical CPU core or one virtual core,** depending on whether the node is a physical host or a virtual machine running inside a physical machine.

Memory resource units

Memory can be expressed as a plain integer or as a fixed-point number, using one of these suffixes:
Ei, Pi, Ti, Gi, Mi, Ki.

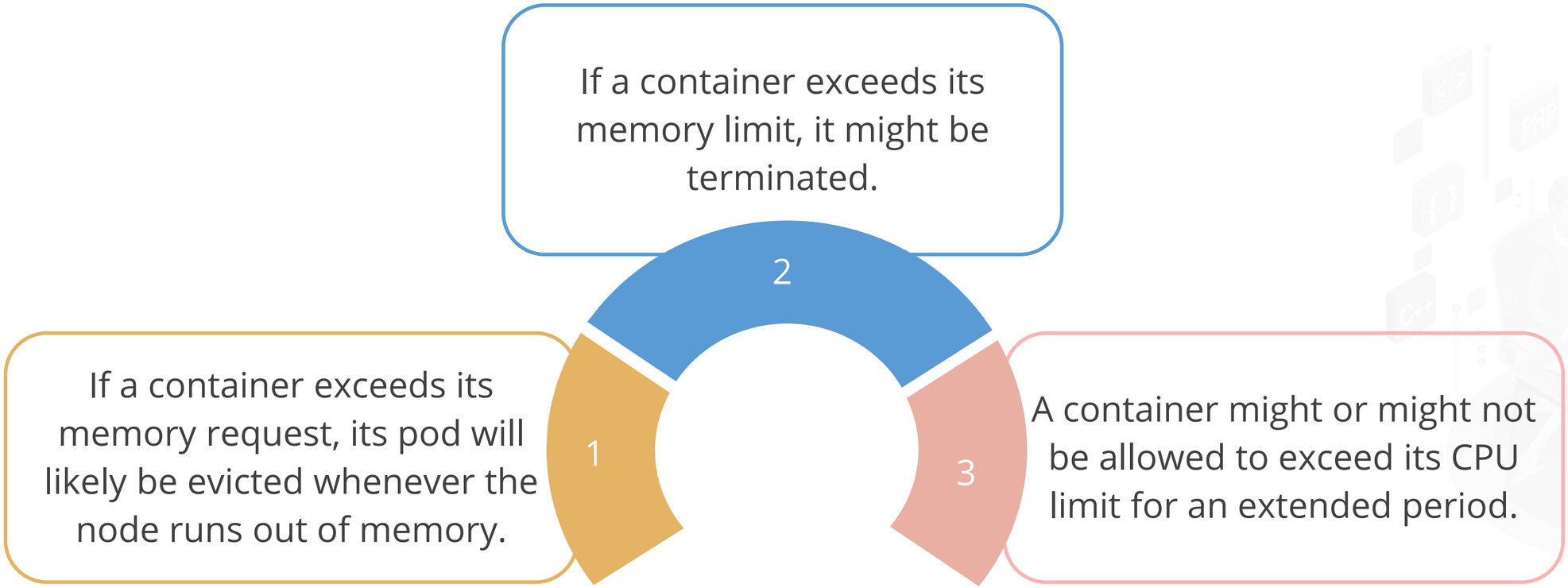
Resource Units in Kubernetes

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  Frontend
  spec:
    containers:
     -  name: app
        image: images.my company.example/app;V4
      resources:
            requests:
              memory: “64Mi”
              cpu: ‘250m”
            limits:
              memory: “128Mi”
              cpu: “500m”
     -  name: log-aggregator
        image: images.my company.example/log-aggregator;v6
        resources:
            requests:
              memory: “64Mi”
              cpu: ‘250m”
            limits:
              memory: “128Mi”
              cpu: “500m”
```

- In this example, the pod has two containers.
- Each container has a request of 0.25 CPU and 64MiB (2^{26} bytes) of memory.
- Each container has a limit of 0.5 to 1 CPU and 128MiB to 256 MiB of memory.

How Pods with Resource Limits Are Run

When the kubelet starts a container of a pod, it passes the CPU and memory limits to the container runtime.



Local Ephemeral Storage

Nodes have **local ephemeral storage**, backed by locally-attached writeable devices or, sometimes, by RAM.

Kubernetes supports two ways to configure local ephemeral storage on a node:

Single Filesystem

In this configuration, place the different kinds of ephemeral local data (emptyDir volumes, writeable layers, container images, logs) in one filesystem.

Two Filesystems

Use this filesystem for other data (for example, system logs that are not related to Kubernetes). It can even be the root filesystem.

Local Ephemeral Storage

Ephemeral storage can be used for managing local ephemeral storage. Each container of a pod can specify one or more of the following:

spec.containers[ ].resources.limits.ephemeral-storage

spec.containers[ ].resources.requests.ephemeral-storage

Local Ephemeral Storage

In the following example, the pod has two containers:

```
apiVersion: v1
 Kind:  pod
 Metadata:
    name:  Frontend
   spec:
     containers:
      -  name: app
         image: images.my company.example/app;V4
       resources:
            requests:
              ephemeral-storage: “2Gi”
            limits:
              ephemeral-storage: “4Gi”
              cpu: “500m”
      -  name: log-aggregator
         image: images.my company.example/log-aggregator;v6
         resources:
            requests:
              ephemeral-storage: “2Gi”
            limits:
              ephemeral-storage: “4Gi”
```

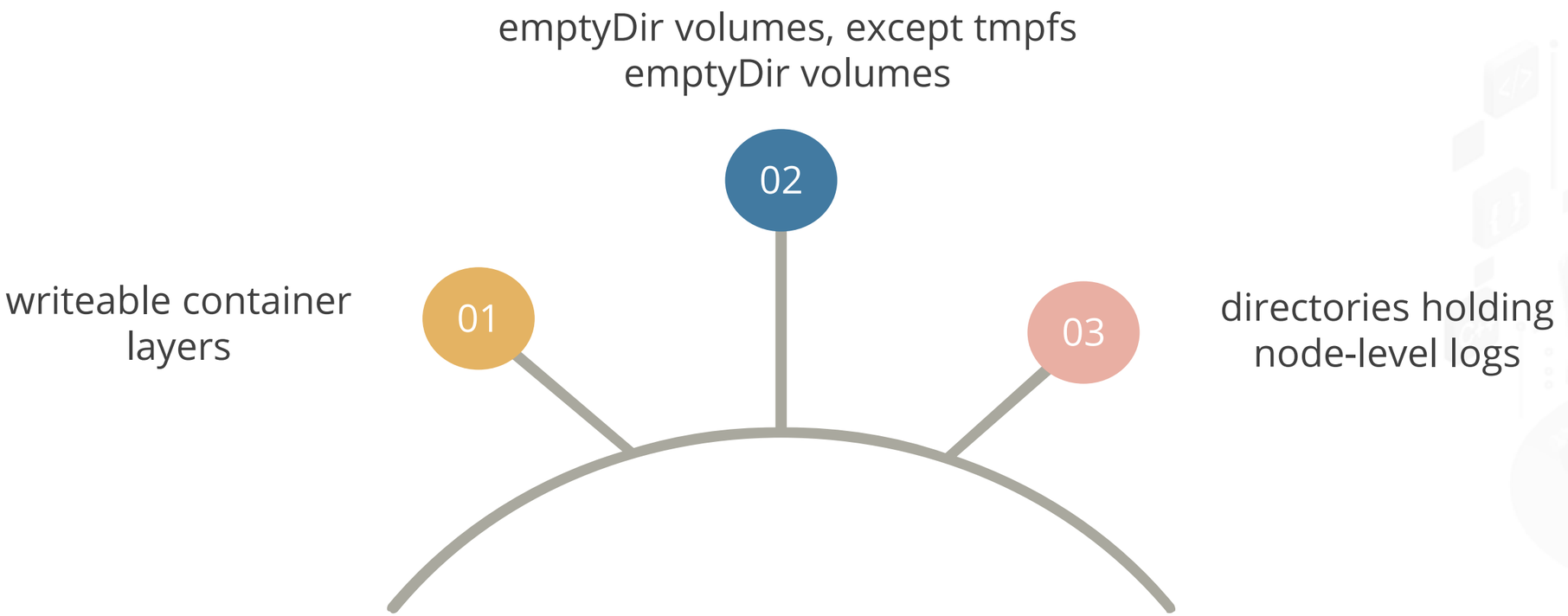
Local Ephemeral Storage

When a pod is created, the Kubernetes scheduler selects a node on which the pod will run.

The scheduler ensures that the sum of the resource requests of the scheduled containers is less than the capacity of the node.

Ephemeral Storage Consumption Management

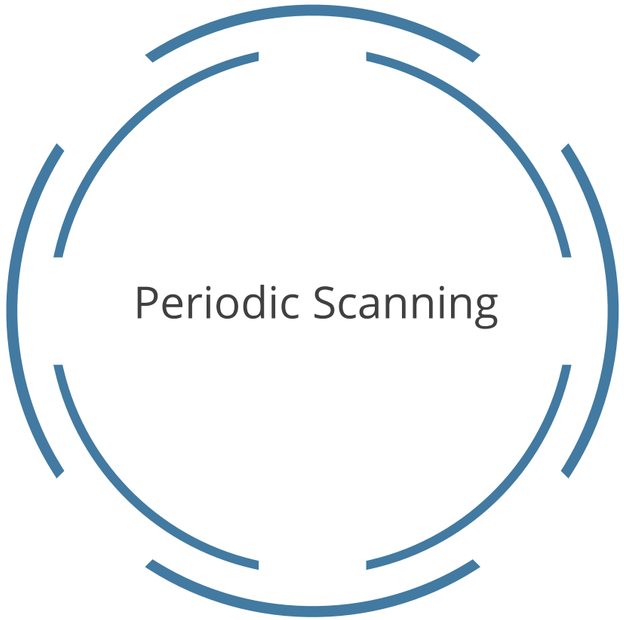
If the kubelet is managing local ephemeral storage as a resource, it measures the use of storage in:

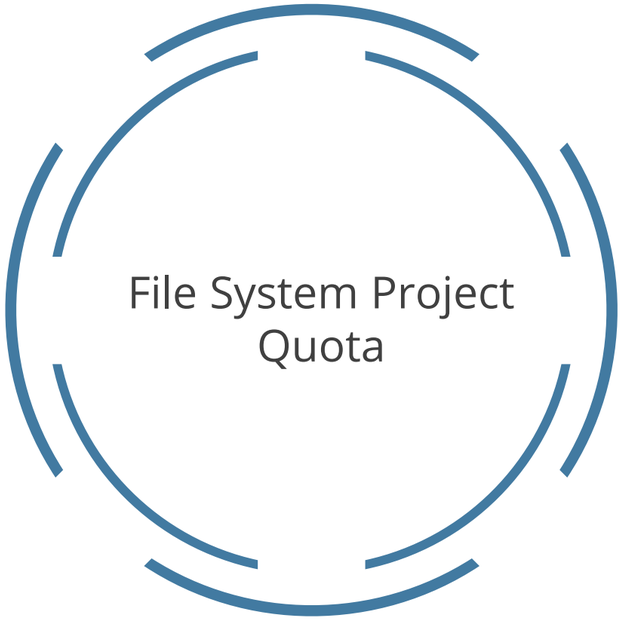


If a pod uses more ephemeral storage than allowed, it receives an eviction signal from kubelet.

Periodic Scanning and File System Project Quota

The kubelet supports different ways to measure pod storage use:





Resource Quota

Resource Quota

Problem Statement

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Solution

Resource quotas are a tool for administrators to address this concern.

Explanation

A **ResourceQuota** object sets limits on the number and amount of resources used per namespace, including object creation and compute resource usage.

Resource Quota

Resource Quota works as follows:

- Teams use separate namespaces, managed via RBAC.
- Each namespace gets a ResourceQuota from the admin.
- Users create resources within namespaces, monitored by the quota system to stay within ResourceQuota limits.
- Resource creation exceeding a quota results in a **403 FORBIDDEN** error with details of the breach.
- For CPU and memory, users must define requests or limits if quotas are active; non-compliance may prevent pod creation.

Resource Quota

Resource Name	Description
Limits.cpu	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
Limits.memory	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value
Requests.cpu	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value
Requests.memory	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
Hugepages-<size>	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
CPU	Same as requests.cpu
Memory	Same as requests.cpu

Compute Resource Quota

In the following example, setting resource quota for compute resources:

```
apiVersion: v1
kind: ResourceQuota
metadata:
     name: compute-resources
     namespace: test
spec:
     hard:
          requests.cpu: "1"
          requests.memory: 1Gi
          limits.cpu: "2"
          limits.memory: 2Gi
          requests.nvidia.com/gpu: 4
```

Storage Resource Quota

Resource Name	Description
requests.storage	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
persistentvolumeclaims	The total number of PersistentVolumeClaims that can exist in the namespace.
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	Across all persistent volume claims associated with the <storage-class-name>, the sum of storage requests cannot exceed this value.
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	Across all persistent volume claims associated with the <storage-class-name>, the total number of persistent volume claims that can exist in the namespace.

Object Count Quota

Set quota for the total number of certain Kubernetes resources

1. **count/<resource>.<group>** for resources from non-core groups

2. **count/<resource>** for resources from the core group

Object Count Quota

The examples demonstrate referencing various resource types for quota settings.

count/pods: Set a quota on the number of pods.

count/deployments.apps: Set a quota on the number of deployments in the apps API group.

count/services: Set a quota on the number of services**.**

count/secrets: Set a quota on the number of secrets.

Object Count Quota

The examples demonstrate referencing various resource types for quota settings.

count/configmaps: Set a quota on the number of ConfigMaps.

count/jobs.batch: Set a quota on the number of jobs in the batch API group.

count/cronjobs.batch: Set a quota on the number of cron jobs in the batch API group.

Object Count Quota

In the following example, setting resource quota for Kubernetes resources:

```
apiVersion: v1
kind: ResourceQuota
metadata:
     name: compute-resources
     namespace: test
spec:
     hard:
          configmaps: "10"
          persistentvolumeclaims: "4"
          pods: "4"
          replicationcontrollers: "20"
          secrets: "10"
          services: “4"
```

Managing Container Resources with Resource Quota

Duration: 10 mins

Problem Statement:

You've been asked to create a namespace with a resource quota for managing resources within a Kubernetes container

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a namespace with a resource quota
2. Validate the resource quota by creating pods

Multi-Container Pods

Using Pods in Different Situations

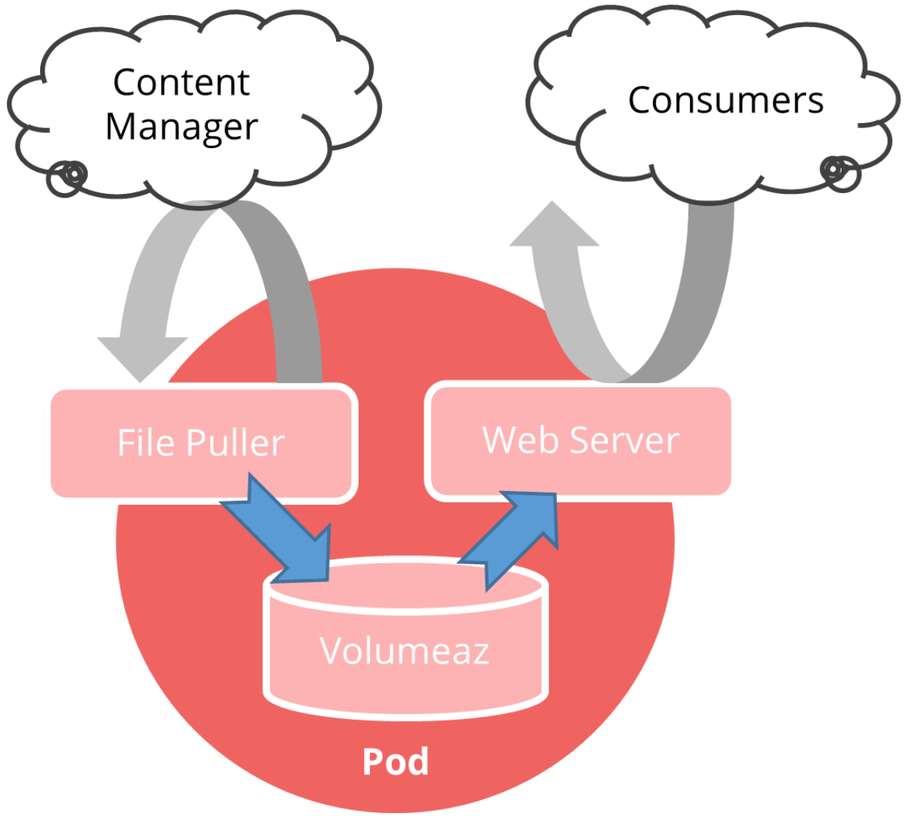
Pods are created using workload resources such as Deployment or Job. pods in a Kubernetes cluster are used in two main ways:

Pods that run a single container

Pods that run multiple containers that need to work together

Using Pods in Different Situations

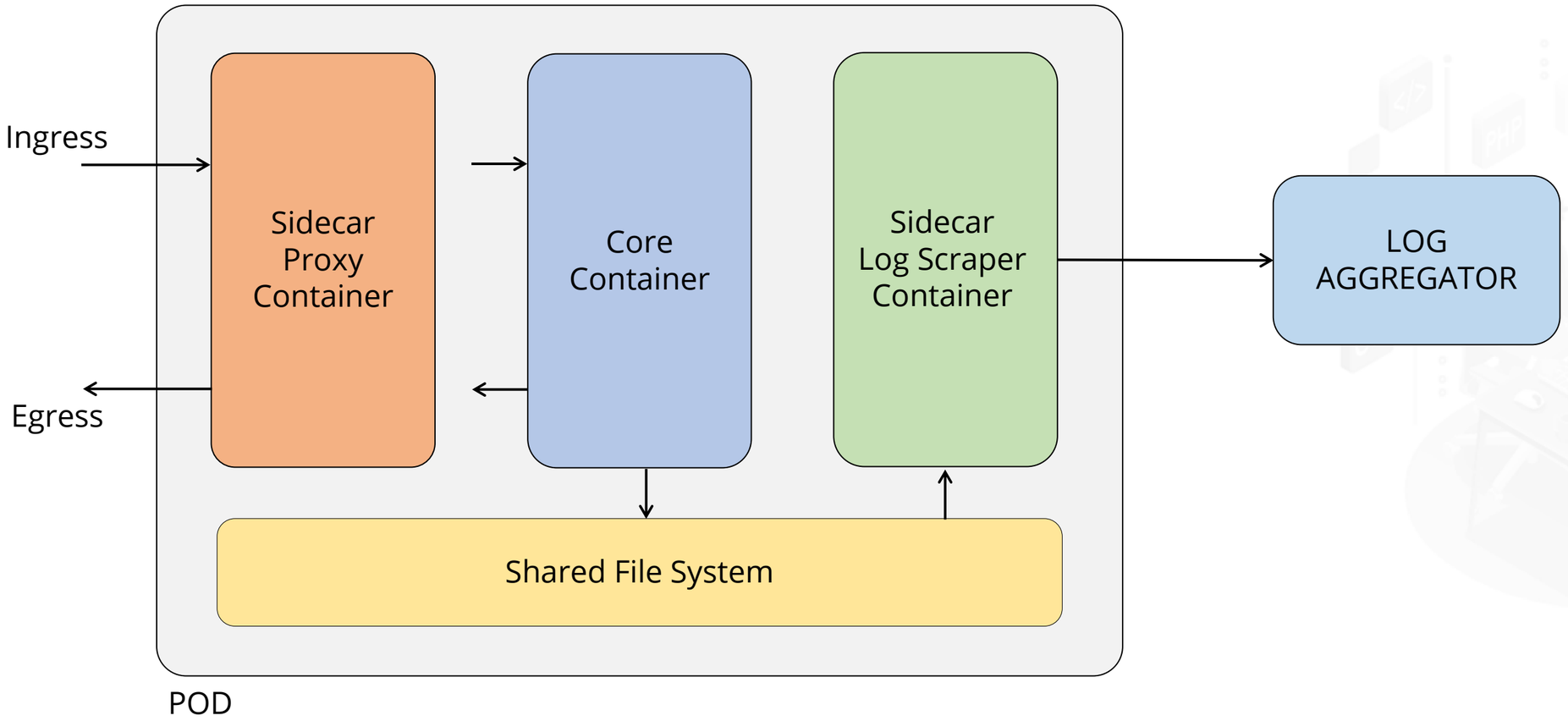
Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service.



There can be a container that acts as a web server for files in a shared volume and a separate **sidecar** container that updates those files from a remote source.

Different Sidecar containers

Sidecar containers of a pod supports functionalities, such as logging, monitoring, service discovery, security, and more.



Creating a Multi-Container Pod

Duration: 10 mins

Problem Statement:

You've been asked to create a multi-container pod in a Kubernetes cluster, allowing you to run and interact with multiple containers within a single pod

ASSISTED PRACTICE

Assisted Practice: Guidelines

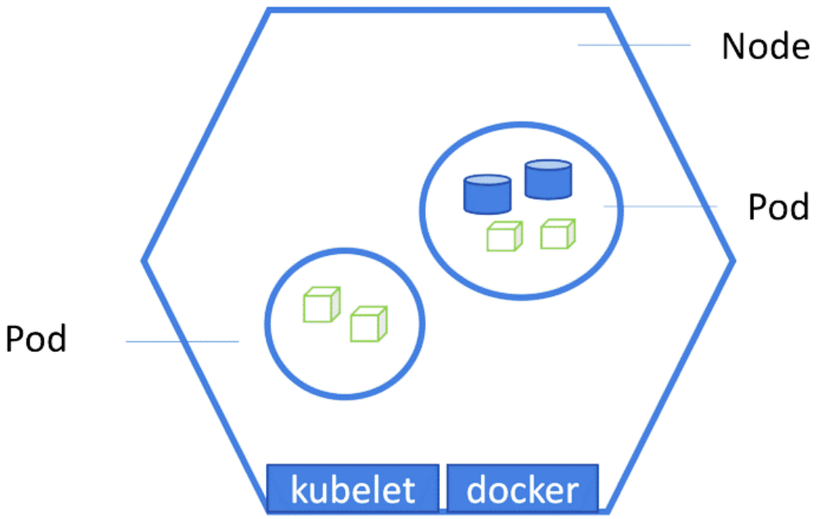
Steps to be followed:

1. Create and access a multi-container pod

Static Pods

Overview

Static pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. They are always bound to one kubelet on a specific node.



- The main use of Static pods is to run a self-hosted control plane.
- The kubelet automatically tries to create a mirror pod on the Kubernetes API server for each static pod.
- The pods running on a node are visible on the API server but cannot be controlled from there.

Control-plane Pods

Pods that form the control plane, like the Kube-apiserver, Kube-scheduler, Kube-controller-manager, and etcd, operate as static pods overseen by the Kubelet service.

In the control-plane node, The control-plane pods manifest can be found in **/etc/kubernetes/manifests**

kubelet on control-plane node uses this directory by running it with --pod-manifest-path=/etc/kubernetes/manifests/

Understanding Static Pods

Duration: 5 mins

Problem Statement:

You've been asked to create a static pod.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

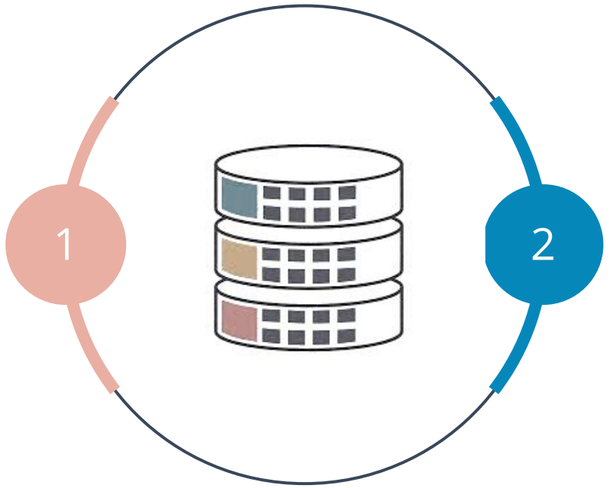
1. Create a static pod in the worker node
2. Try to delete the pod from control-plane
3. Delete static pod from worker node

Deployment

What Is Deployment?

A Deployment provides declarative updates for pods and ReplicaSets.

Users set a target configuration for a Deployment, and the controller adjusts the current state to match it gradually.



Users can configure Deployments to generate new ReplicaSets or replace old ones, inheriting their resources.

Creating a Deployment

To create a Deployment from a YAML manifest

```
kind: Deployment
metadata:
  name: nginx-deployment                                    # Name
of Deployment
  labels:
    app: nginx
spec:
  replicas: 3
# Number of pods
  selector:
    matchLabels:
      app: nginx
  template:
# pod Specifications
    << POD Template >>
```

Use Cases

Typical use cases for Deployments are:

1. Create a Deployment to rollout a ReplicaSet
2. Declare the new state of pods
3. Rollback to an earlier Deployment revision
4. Scale up the Deployment to facilitate more load

Use Cases

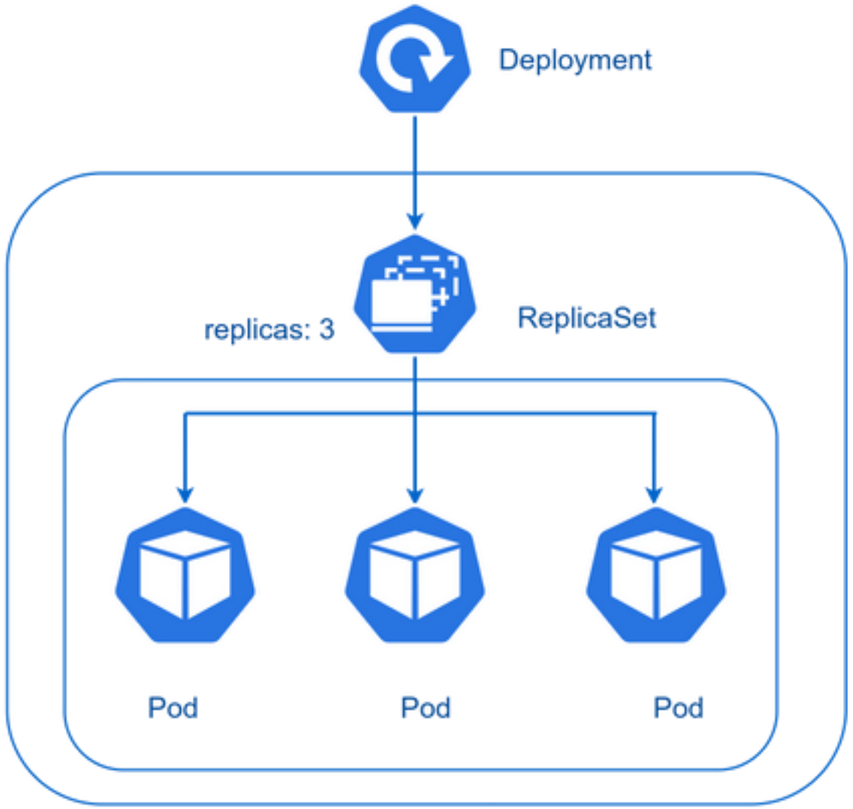
Typical use cases for Deployments are:

5. Pause the rollout of a Deployment

6. Use the status of the Deployment

7. Clean up older ReplicaSets

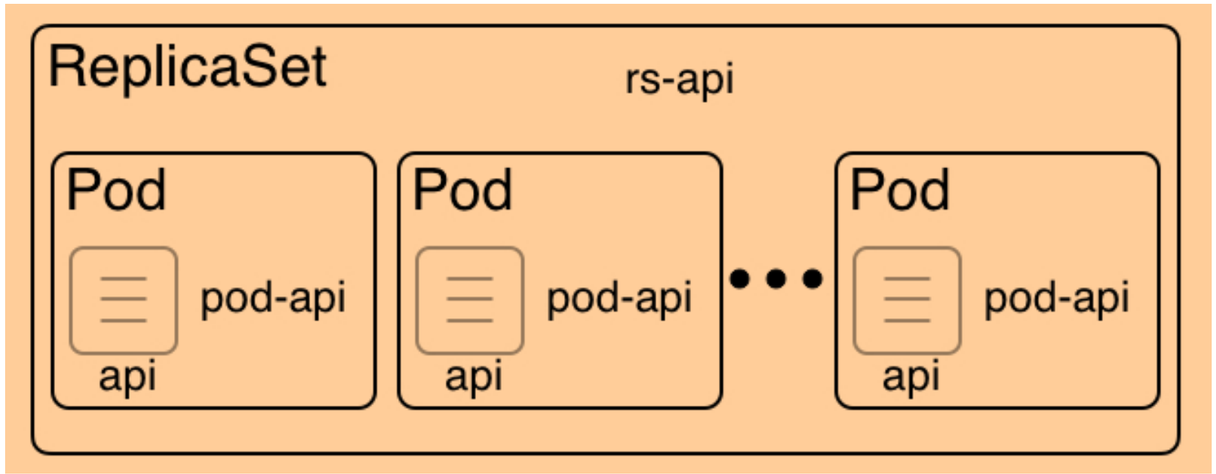
Deployment and Replicasets



- Deployment creates a ReplicaSet that creates three replicated pods.
- Deployment updated the pods by creating a new ReplicaSet and scaling it up to 3 replicas
- Deployment maintains the changes as Revision and for each revision, a ReplicaSet will be maintained.

Purpose of ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical pods.



How a ReplicaSet Works

A ReplicaSet is defined with fields, including a selector that specifies how to identify pods.

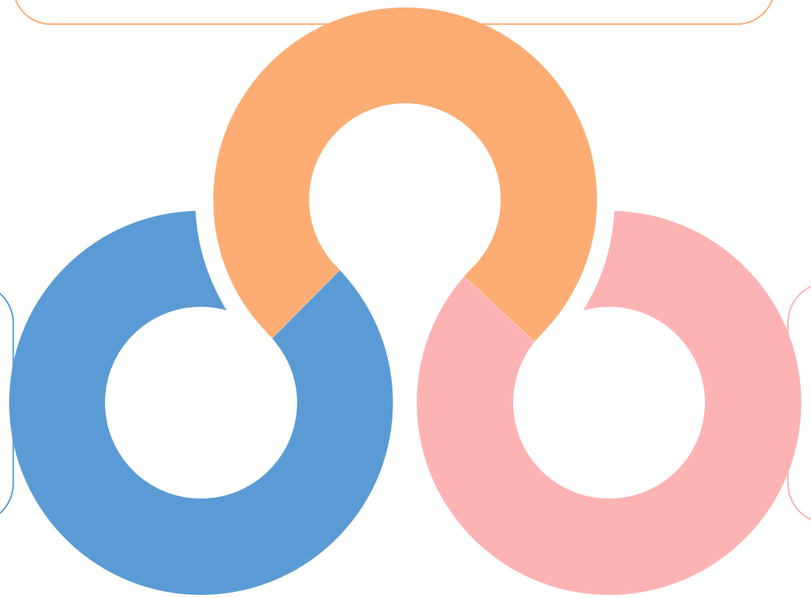
A ReplicaSet identifies new pods to acquire by using its selector.

A ReplicaSet is linked to its pods via the pods' **metadata.ownerReferences** field, which specifies by what resource the current object is owned.

ReplicaSet in Use

A ReplicaSet ensures that a specified number of pod replicas are running at any given time.

Deployment provides declarative updates to pods along with a lot of other useful features.



Deployment is a higher-level concept that manages ReplicaSets.

Deployment is recommended over directly using ReplicaSets.

Using a Deployment

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-
frontened:v3
```

When to Use a ReplicaSet

To check for the pods brought up, use:

```
kubectl get pods
```

The pod information will be shown like:

Name	Ready	Status	Restarts	Age
frontend-b2zdv	1/1	Running	0	6m36s
frontend-vcmts	1/1	Running	0	6m36s
frontend-wtsmm	1/1	Running	0	6m36s

When to Use a ReplicaSet

To confirm that the frontend ReplicaSet is the owner of the pods, retrieve the yaml for a running pod.

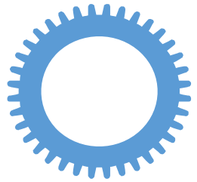
kubectl get pods frontend-b2zdv -o yaml

```
apiVersion: app/v1
Kind: pod
Metadata:
 creationTimestamp: “2020-02-12T07:06:162”
  generateName: frontend
  labels:
    tier: frontend
name: frontend-b2zdv
Namespace: default
 ownerreferences:
 apiVersion: apps/v1
 blockOwenersDeletion: true
 controller: true
 kind: replicaSet
 name: frontend
 uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
```

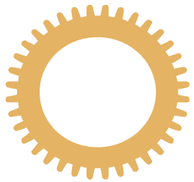
The output will be similar to the given image, with the frontend ReplicaSet's info set in the metadata's ownerReferences field.

Pod Template

Pod templates are specifications for creating pods and are included in workload resources such as Deployments, Jobs, and DaemonSets.



The **.spec.template** is a pod template that is also required to have labels in place.



For the template's Restart Policy field, **.spec.template.spec.restartPolicy**, the only allowed value, is **Always**, which is the default.

Pod Labels and Selector

The label and selectors helps the users to identify a set of objects. It is the core grouping primitive in Kubernetes.

1. The **.spec.selector** field is a label selector. These are the labels used to identify potential pods to acquire.

2. In the ReplicaSet, **.spec.template.metadata.labels** must match **spec.selector**, or it will be rejected by the API.

Replicas

Specify the number of pods that should run concurrently by setting **.spec.replicas**. The ReplicaSet will create or delete its pods to match this number.

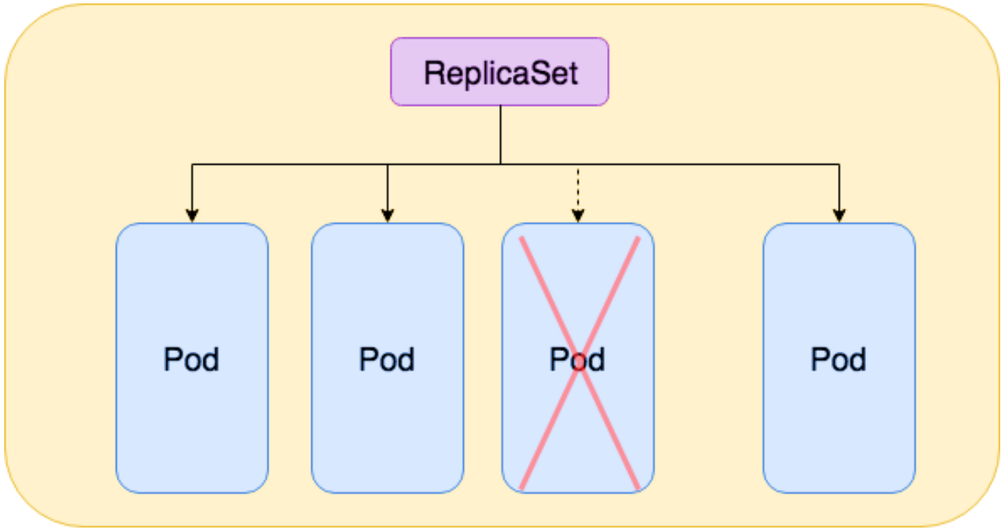
If **.spec.replicas** is not specified, it sets to 1 by default.

Working with Deployments

To delete a Deployment and all its pods, use **kubectl delete**. The Garbage Collector will automatically delete all the dependent pods by default.

When using the REST API or the **client-go** library, ensure that **propagationPolicy** is set to **Background** or **Foreground** in the -d option.

Working with Deployments



Isolating pods from a ReplicaSet

- pods can be removed from a ReplicaSet by changing their labels.
- This technique may be used to remove pods from Service for debugging, data recovery, and so on.
- pods that are removed in this way will be replaced automatically.

Scaling Deployments

Scale a Deployment by updating the .spec.replicas value. To scale down, the ReplicaSet controller uses an algorithm to determine which pods to remove first

1. Pending (and unschedulable) pods are scaled down first.

2. If the **controller.kubernetes.io/pod-deletion-cost** annotation is set, the pod with the lower value will come first.

3. Pods on nodes with more Replicas come before pods on nodes with fewer replicas.

4. If the pods' creation times differ, the pod that was created more recently comes before the older pod.

Scaling Deployments

An application must be scaled when the load on the application grows or shrinks.

An example of this is shown below:

Example

```
// To decrease the number of NGINX replicas from 3 to 1 perform:
kubectl scale deployment/my-nginx --replicas=1

deployment.apps/my-nginx scaled

// Now, you only have one pod managed by the deployment.

kubectl get pod -l app=nginx

NAME                          READY     STATUS    RESTARTS   AGE
my-nginx-2035384211-j5fhi     1/1       Running   0          30m

// To have the system automatically choose the number of NGINX replicas as needed, ranging
from 1 to 3:

kubectl autoscale deployment/my-nginx --min=1 --max=3

horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Updating Labels

Update labels on existing pods and other resources as needed before creating new ones with kubectl label. To label all NGINX pods, use the following command:

```
kubectl label pod -l app=nginx tier=fe

pod/my-nginx-2035384211-j5fhi labeled
pod/my-nginx-2035384211-u2c7e labeled
pod/my-nginx-2035384211-u3t6x labeled

// This filters all pods with the label "app=nginx" and then labels them
with "tier=fe". To see the pod you labeled, run:

kubectl get pod -l app=nginx -L tier

NAME                        READY     STATUS    RESTARTS   AGE       TIER
my-nginx-2035384211-j5fhi   1/1       Running   0          23m       fe
my-nginx-2035384211-u2c7e   1/1       Running   0          23m       fe
my-nginx-2035384211-u3t6x   1/1       Running   0          23m       fe
```

Updating Annotations

Annotations provide non-essential metadata accessible to API clients like tools and libraries, and can be added using kubectl annotate, as illustrated below:

```
kubectl annotate pod my-nginx-v4-9gw19 description='my frontend running
nginx'
kubectl get pod my-nginx-v4-9gw19 -o yaml

apiVersion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
```

kubectl Apply

- kubectl apply pushes configuration changes to the cluster.
- It compares the version of the configuration that is being pushed with the previous version.
- It applies changes made without overwriting any automated changes to properties that have not been specified.

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-
deployment.yaml
deployment.apps/my-nginx configured
```

kubectl Edit

Resources can be updated using kubectl edit.

An example of this is shown below:

Example

```
kubectl edit deployment/my-nginx

// This is similar to first getting the resource, editing it in the text
editor, and then applying the resource with the updated version:

kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
vi /tmp/nginx.yaml
# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml
deployment.apps/my-nginx configured

rm /tmp/nginx.yaml
```

Disruptive Updates

replace --force changes resource fields that cannot be updated once initialized. It deletes and re-creates the resource.

The process to modify the original configuration file is shown below:

```
kubectl replace -f https://k8s.io/examples/application/nginx/nginx-
deployment.yaml --force

deployment.apps/my-nginx deleted
deployment.apps/my-nginx replaced
```

Horizontal Pod Autoscaler (HPA)

A ReplicaSet can also be a target for **Horizontal pod Autoscalers (HPA)**. Thus, a Deployment can be auto-scaled by an HPA. Here is an example of HPA targeting the Deployment:

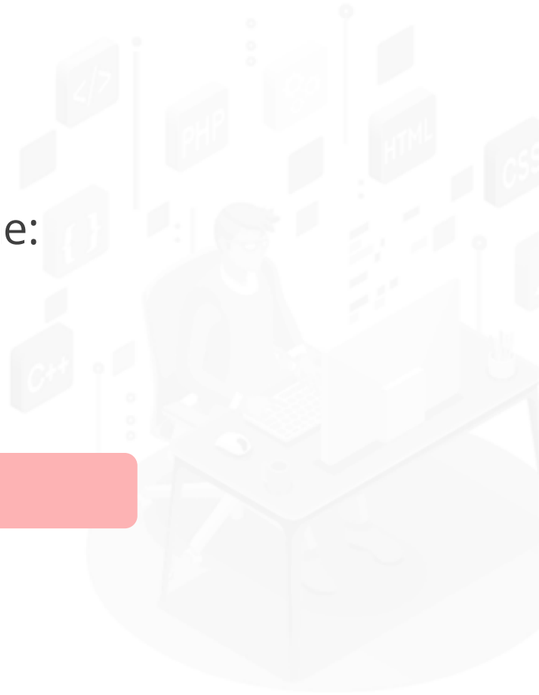
```
Apiversion: autoscaling/v1
Kind: horizontalpodAutoscaler
Metadata:
        name: fronted-scalar
Spec:
        scaleTargetRef:
        kind: Deployment
        name: frontend
        minReplicas: 3
        maxReplicas: 10
        targetCPUUtilizationpercentage: 50
```

Horizontal Pod Autoscaler (HPA)

Save this manifest into **hpa.yaml** and submit it to a Kubernetes cluster to create the defined HPA that autoscales the target Deployment, depending on the CPU usage of the replicated pods:

Alternatively, use the **kubectl autoscale** command to accomplish the same:

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50
```



Alternatives to ReplicaSet

The following is a list of alternatives to Kubernetes ReplicaSet:

1. Deployment
2. Bare pods
3. Job
4. DaemonSet
5. ReplicationController

Perform Rolling Updates on a Deployments

Deployment Strategies

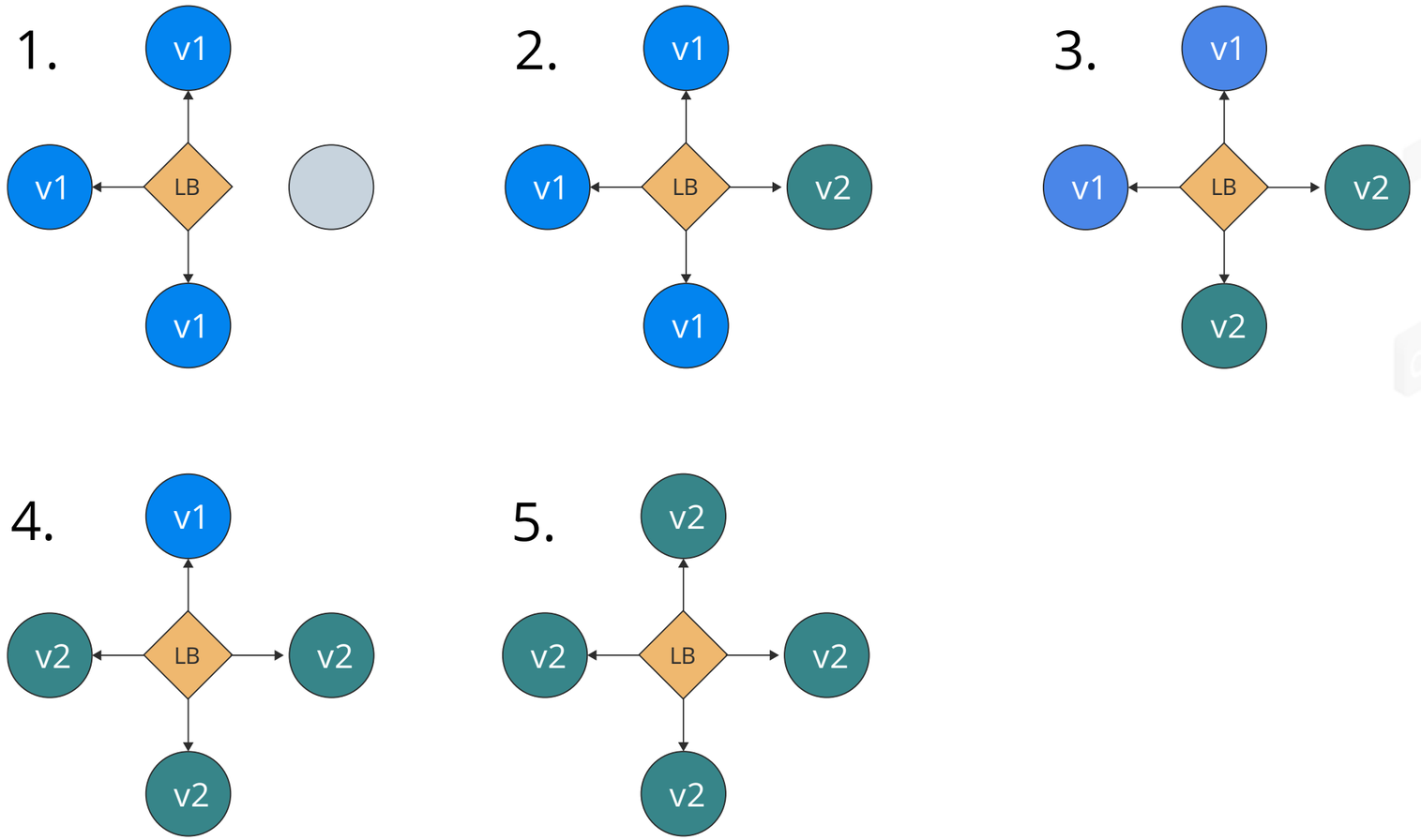
Deployment strategies determine the method of replacing active pods in Kubernetes during configuration changes. There exist two primary strategy types:

1. **Recreate strategy** kills all the pods managed by the Deployment, prior to starting a new pod.

2. **Rolling update strategy** runs old as well as new configurations alongside each other.

Deployment Strategies

1 Transition of pods in RollingUpdate Strategy



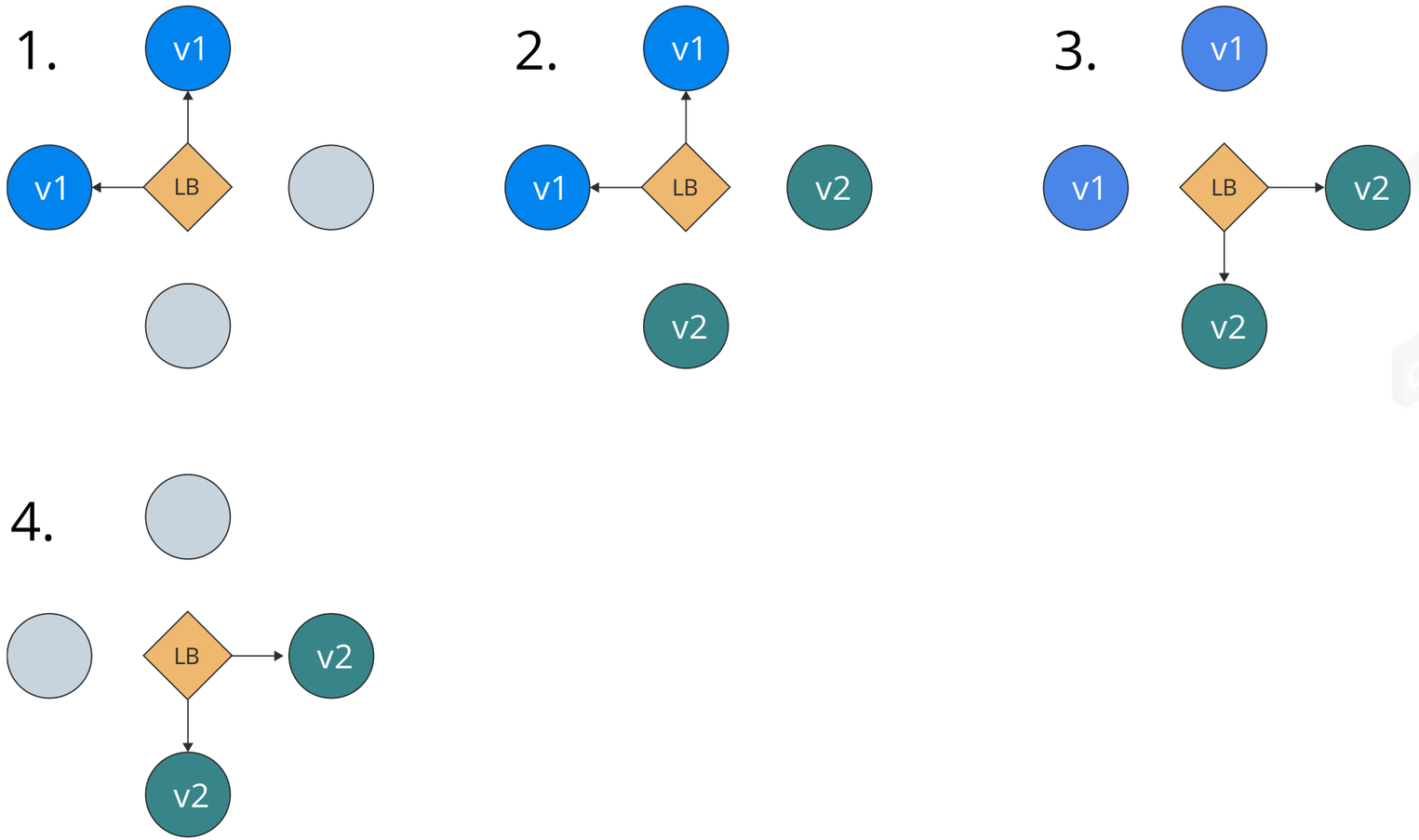
Performing a Rolling Update

To enable the rolling update feature of a Deployment, **.spec.strategy.type** must be set to RollingUpdate. This is shown in the YAML file given below:

```
sion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 strategy:
      rollingUpdate:
           maxSurge: 25%
           maxUnavailable: 1
      type: RollingUpdate
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-frontened:v3
```

Deployment Strategies

2 Transition of pods in Recreate Strategy



Performing a Recreate

To enable the rolling update feature of a Deployment, **.spec.strategy.type** must be set to Recreate. This is shown in the YAML file given below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 strategy:
     type: Recreate
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-frontened:v3
```

Rollout Commands

To get the rollout status of the latest Deployment Rolling Update, use **kubectl rollout status**.

Demo

```
kubectl rollout status deploy/frontend
```

Upon completion of rollout, the output will be as shown below:

Demo

```
deployment ”frontend" successfully rolled out
```

Canary Deployments

Multiple labels may be needed to differentiate Deployments of the same component with different releases or configurations, followed by a canary rollout of the new app version alongside the old one.

A track label distinguishes releases; for instance, **stable** denotes the main stable release.

```
name: frontend
     replicas: 3
     ...
     labels:
        app: guestbook
        tier: frontend
        track: stable
     ...
     image: gb-frontend:v3
// Then, create a new release of the guestbook frontend that carries the
track label with a different value so that two sets of pods do not overlap:
    name: frontend-canary
     replicas: 1
```

Canary Deployments

A new release of the guestbook front end carrying the track label with a different value (canary) may be created to avoid an overlap. This is shown below:

Demo

```
    labels:
       app: guestbook
       tier: frontend
       track: canary
    ...
    image: gb-frontend:v4
```

The front-end service spans both sets of replicas by selecting the common subset of their labels to ensure that the traffic is redirected to both applications.

Demo

```
selector:
    app: guestbook
    tier: frontend
```

Perform Rollbacks on a Deployments

Performing a Rollback on a Deployment

Rolling back a Deployment is a three-step process:

Step 1: Find the Deployment revision, and list all the revisions of a Deployment

```
kubectl rollout history deployment <deployment-name>

This returns a list of Deployment revisions:
deployment "<deployment-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

Performing a Rollback on a Deployment

The details of a specific revision are given below:

```
Kubectl rollout history deployment <deployment-name> --revision=1

// This returns the details of that revision:

deployment "<deployment-name>" with revision #1
pod Template:
Labels:       foo=bar
Containers:
app:
 Image:         ...
 Port:          ...
 Environment:   ...
 Mounts:        ...
 Volumes:       ...
```

Performing a Rollback on a Deployment

Step 2: Roll back to a specific revision

```
// Set the revision number you get from Step 1 in --to-revision

kubectl rollout undo deployment <deployment-name> --to-revision=<revision>

// If it succeeds, the command returns the following:

deployment "<deployment-name>" rolled back
```

Performing a Rollback on a Deployment

Step 3: Watch the progress of the DaemonSet rollback

```
// kubectl rollout undo deployment tells the server to start rolling back the
Deployment. The real rollback is done asynchronously inside the cluster control
plane.

kubectl rollout status deployment/<deployment-name>
```

After completion of rollback, the output will be as shown below:

```
deployment "<deployment-name>" successfully rolled out
```

Deploying Multitier Application Using Kubernetes

Duration: 20 mins

Problem Statement:

You've been asked to deploy a multitier application with Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Deployment for MySQl
2. Creating a Deployment for WordPress
3. Creating a Service for WordPress and MySQL Deployment
4. Verifying the Deployment of the application

Deploying Multitier Application Postgres and Gogs

Duration: 20 mins

Problem Statement:

You've been asked to deploy a multitier application Postgres and Gogs using Kubernetes

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Deployment for Postgres
2. Creating a Deployment for Gogs
3. Creating a Service for Postgres and Gogs Deployment
4. Verifying the Deployment of the application

Deploying a Voting Application

Duration: 20 mins

Problem Statement:

You've been asked to deploy a voting application in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a namespace
2. Creating the application
3. Verifying the Deployment of the application

ReplicaSets and Metrics Server

Duration: 20 mins

Problem Statement:

You've been asked to create a ReplicaSet and deploy a metrics server.

ASSISTED PRACTICE

Assisted Practice: Guidelines

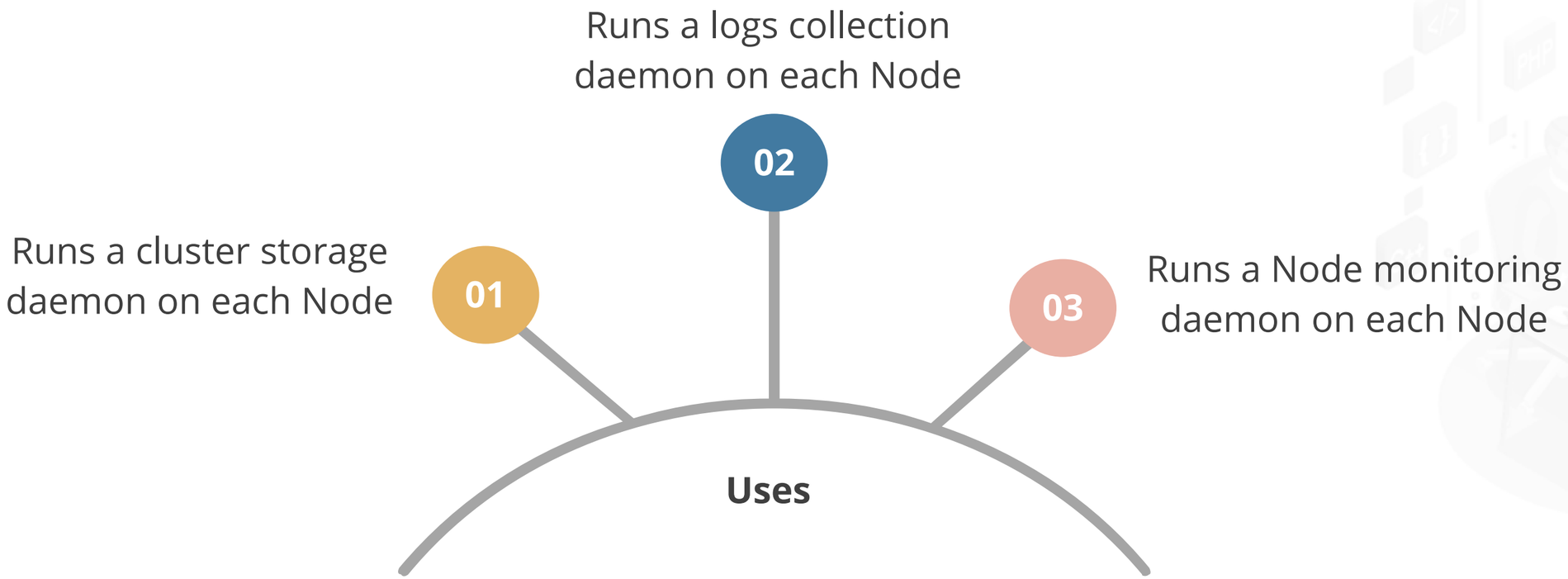
Steps to be followed:

1. Creating a ReplicaSet
2. Configuring the metrics server
3. Verifying the metrics server Deployment

DaemonSet

DaemonSet: Introduction

A DaemonSet ensures that all the Nodes run a copy of a pod.



Writing a DaemonSet Spec

The daemonset.yaml file shown below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
```

Writing a DaemonSet Spec

```
# This toleration is to have the DaemonSet
runnable on master Nodes; remove it if your
masters can't run the pod
- key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image:
quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
```

```
volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath:
/var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Required Fields

As with all other Kubernetes configurations, a DaemonSet needs apiVersion, kind, and metadata fields. A DaemonSet also needs a .spec section.

Note

The DaemonSet object's name needs to be a valid DNS subdomain name.

Pod Template

The **.spec.template** is a pod template.

This template has the same schema as a pod. It is nested and does not have an apiVersion.

A pod template in a DaemonSet must set relevant labels as well as necessary fields.

The RestartPolicy must be unspecified or set to *Always*.

Pod Selector

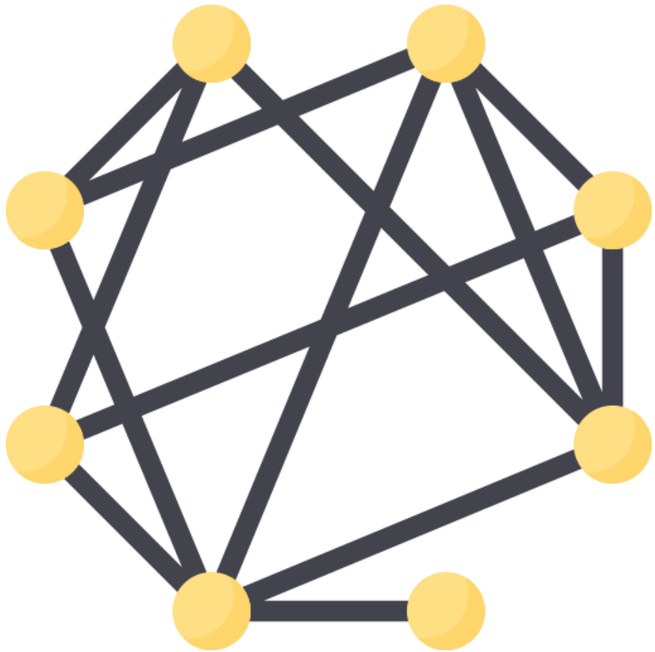
The **.spec.selector** field is a pod selector. The **.spec.selector** object consists of two fields:

1. matchLabels
2. matchExpressions

The result is ANDed when both the fields are specified.

Running Pods on Select Nodes

If **.spec.template.spec.nodeselector** is set, the DaemonSet Controller will create pods on Nodes, matching the Node selector.



How Daemon Pods Are Scheduled

The DaemonSet Controller creates and schedules a DaemonSet.

It results in:

1. Inconsistent pod behavior

2. Loss of control over decisions when preemption is enabled (w.r.t. scheduling)

Running Pods on Select Nodes

The DaemonSet Controller schedules DaemonSets and binds the pod to the target host only when creating or modifying the DaemonSet pod.

Changes are not made to the spec.template of the DaemonSet.

```
nodeAffinity:

necessaryDuringSchedulingIgnoredDuringExecu
tion:
    nodeselectorTerms:
    - matchFields:
      - key: metadata.name
        operator: In
        values:
        - target-host-name
```

Using Labels

Labels allow one to slice and dice resources along any dimension specified by a label.

Demo

```
kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
kubectl get pod -Lapp -Ltier -Lrole

NAME                           READY     STATUS    RESTARTS   AGE       APP         TIER       ROLE
guestbook-fe-4nlpb             1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-fe-ght6d             1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-fe-jpy62             1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-redis-master-5pg3b   1/1       Running   0          1m        guestbook   backend    master
guestbook-redis-slave-2q2yf    1/1       Running   0          1m        guestbook   backend    slave
guestbook-redis-slave-qgazl    1/1       Running   0          1m        guestbook   backend    slave
my-nginx-divi2                 1/1       Running   0          29m       nginx       <none>     <none>
my-nginx-o0ef1                 1/1       Running   0          29m       nginx       <none>     <none>

kubectl get pod -lapp=guestbook,role=slave

NAME                           READY     STATUS    RESTARTS   AGE
guestbook-redis-slave-2q2yf    1/1       Running   0          3m
guestbook-redis-slave-qgazl    1/1       Running   0          3m
```

Deployment of Flask Application with Redis

Duration: 15 mins

Problem Statement:

You have been assigned a task to deploy a Flask application with Redis.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a new directory and adding the required files
2. Creating and tagging the Flask image
3. Logging into Docker and pushing the Flask image
4. Creating the Redis and Flask Deployments
5. Creating the Redis and Flask Services
6. Verifying the Flask application

Perform Rolling Updates on a DaemonSet

DaemonSet Update Strategy

A DaemonSet has two types of update strategies:

OnDelete

- Users update a DaemonSet template.
- A new DaemonSet pod is created only after the old DaemonSet pod is deleted.

RollingUpdate

- Users update a DaemonSet template.
- The old DaemonSet pod is killed and a new DaemonSet pod is created automatically.

Performing a Rolling Update

To enable the rolling update feature of a DaemonSet, **.spec.updateStrategy.type** must be set to RollingUpdate. This is shown in the YAML file given below:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
 spec:
      tolerations:
```

Performing a Rolling Update

```
# This toleration is to have the Daemondet runnable on master nodes. Remove it if your masters can't
run the pod.
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Performing a Rolling Update

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet.

Demo

```
kubectl create -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml

// Alternatively, use kubectl apply to create the same DaemonSet if you plan to update the
DaemonSet with kubectl apply.

kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml
```

To check the update strategy, use the following command:

Demo

```
// Check the update strategy of your DaemonSet, and make sure it's set to RollingUpdate.

kubectl get ds/fluentd-elasticsearch -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}' -n
kube-system
```

Performing a Rolling Update

Use the following command to check the DaemonSet manifest if the DaemonSet has not been created in the system.

Demo

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml --dry-
run=client -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
```

The output will be as shown below:

Demo

```
RollingUpdate

// If the output isn't RollingUpdate, go back and modify the DaemonSet object or
manifest.
```

Updating a DaemonSet Template

DaemonSet can be updated by applying a new YAML file as shown below:

Demo

```
    apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      # This toleration is to have the DaemonSet runnable
on master Nodes. Remove it if your master Nodes can't run the
pod.
- key: node-role.kubernetes.io/master
```

Demo

```
effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Commands

To update DaemonSets using configuration files, use **kubectl apply**.

Demo

```
//Declarative command
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset- update.yaml
```

To update DaemonSets using imperative commands, use **kubectl edit**.

Demo

```
//Imperative command
kubectl edit ds/fluentd-elasticsearch -n kube-system
```

To update a container image in the DaemonSet template, use **kubectl set image**.

Demo

```
kubectl set image ds/fluentd-elasticsearch fluentd-
elasticsearch=quay.io/fluentd_elasticsearch/fluentd:v2.6.0 -n kube-system
```

Commands

To get the rollout status of the latest DaemonSet Rolling Update, use **kubectl rollout status**.

Demo

```
kubectl rollout status ds/fluentd-elasticsearch -n kube-system
```

Upon completion of rollout, the output will be as shown below:

Demo

```
daemonset "fluentd-elasticsearch" successfully rolled out
```

Troubleshooting

When the rolling update is not completed due to nonavailability of resources, find the nodes that do not have DaemonSet pods scheduled.

This can be done by comparing the output of kubectl get Nodes, where user will get the following output:

Demo

```
kubectl get pod -l name=fluentd-elasticsearch -o wide -n kube-system
```

Troubleshooting

Broken rollout

- If a recent DaemonSet template update is broken, DaemonSet rollout will not progress.
- Example: If the container is crash looping or the container image does not exist (often due to a typo)

Clock skew

If .spec.minReadySeconds is specified in the DaemonSet, clock skew between the master and the Nodes will make the DaemonSet unable to detect the right rollout progress.

Clean up

To delete DaemonSet from a namespace, use the command shown below.

```
kubectl delete ds fluentd-elasticsearch -n kube-system
```

Rollbacks

Performing a Rollback on a DaemonSet

Rolling back a DaemonSet is a three-step process:

Step 1: Find the DaemonSet revision, and list all the revisions of a DaemonSet

Demo

```
kubectl rollout history daemonset <daemonset-name>

This returns a list of DaemonSet revisions:
daemonsets "<daemonset-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

Performing a Rollback on a DaemonSet

The details of a specific revision are given below:

Demo

```
Kubectl rollout history daemonset <daemonset-name> --revision=1

// This returns the details of that revision:

daemonsets "<daemonset-name>" with revision #1
pod Template:
Labels:       foo=bar
Containers:
app:
 Image:         ...
 Port:          ...
 Environment:   ...
 Mounts:        ...
 Volumes:       ...
```

Performing a Rollback on a DaemonSet

Step 2: Roll back to a specific revision

Demo

```
// Set the revision number you get from Step 1 in --to-revision

kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>

// If it succeeds, the command returns the following:

daemonset "<daemonset-name>" rolled back
```

Performing a Rollback on a DaemonSet

Step 3: Watch the progress of the DaemonSet rollback

Demo

```
// kubectl rollout undo daemonset tells the server to start rolling back the
DaemonSet. The real rollback is done asynchronously inside the cluster control plane.
// To watch the progress of the rollback, input the command given below:

kubectl rollout status ds/<daemonset-name>
```

After completion of rollback, the output will be as shown below:

Demo

```
daemonset "<daemonset-name>" successfully rolled out
```

DaemonSet Revisions

To see what is stored in each revision, find the DaemonSet revision raw resources.

Demo

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-
selector-value>

// This returns a list of controller revisions:

NAME                                  CONTROLLER                     REVISION
AGE
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-
name>     1          1h
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-
name>     2          1h
```

Every controller revision stores annotations and a template of a DaemonSet revision.

Rollout

Duration: 10 mins

Problem Statement:

You've been assigned a task to switch the Deployment image versions via rollout.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

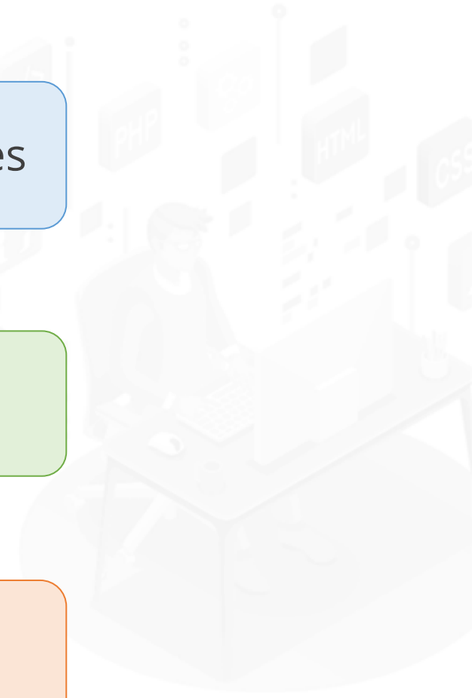
1. Creating a Deployment
2. Upgrading the image version
3. Switching back to the old version

Application Configuration

Overview

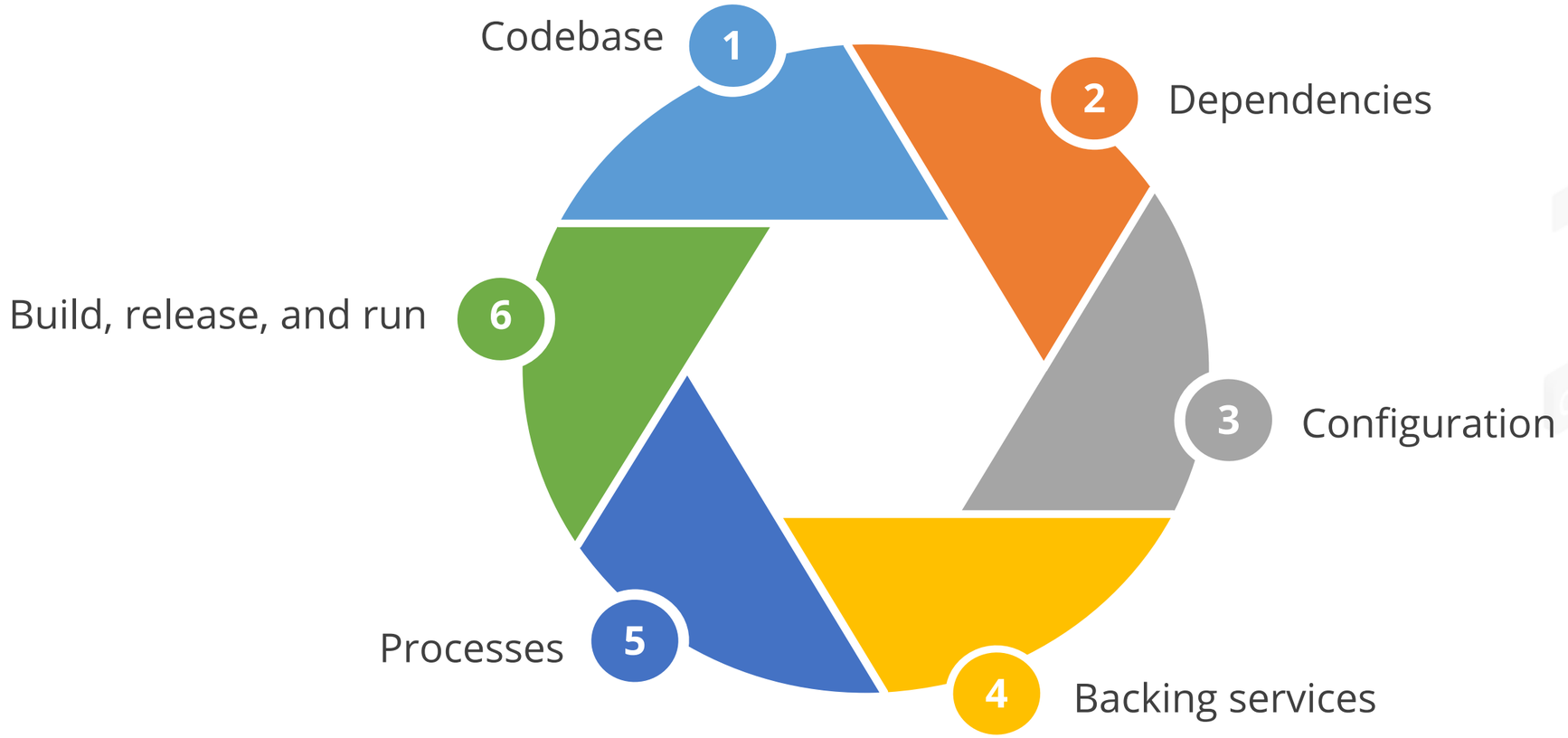
An application’s configuration is everything that is likely to vary between deploys (staging, production, developer environments, and so on). This includes:

1. Resource handles to the database, Memcached, and other backing services
2. Credentials to external services such as Amazon S3 or Twitter
3. Per-deploy values such as canonical hostname for the deploy



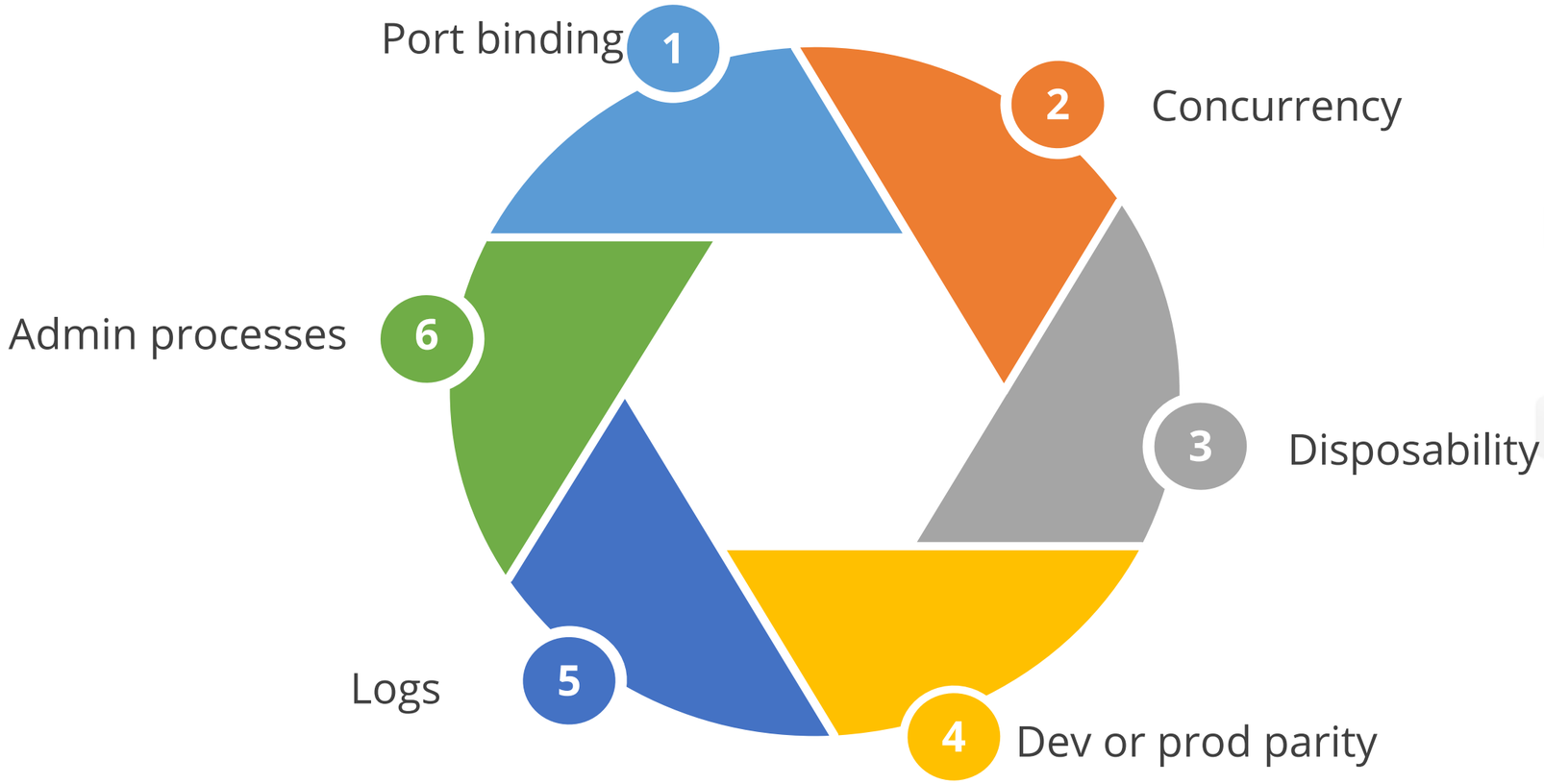
Twelve-Factor Principles

Kubernetes has evolved and is heavily influenced by twelve-factor principles:



Twelve-Factor Principles

Kubernetes has evolved and is heavily influenced by twelve-factor principles:



A Typical Way for Configuration in Kubernetes

To configure the apps in Kubernetes, use:

Environment variables for configuration

ConfigMap

Secret

Using Environment Variables for Configuration

Here is an example of how to use environment variables by specifying them directly within the pod specification:

```
apiVersion: v1
Kind: pod
Metadata:
        name: pod1
Spec:
        containers:
                name: nginx
                image: nginx
                env:
                        name: ENVVAR1
                        value: value 1
                        name: ENVVAR2
                        value: value2
```

Using Environment Variables for Configuration

Define two variables **ENVVAR1** and **ENVVAR2** in **spec.containers.env** with the values **value1** and **value2** respectively.

Use the **kubectl apply** command to submit the pod Information to Kubernetes:

```
$ kubectl apply -f
https://raw.githubusercontent.com/abhirockzz/kubernetes-in-a-nutshell/master/configuration/kin-config-envvar-in-pod.yaml
pod/pod1 created
```

Use grep to filter the variable(s):

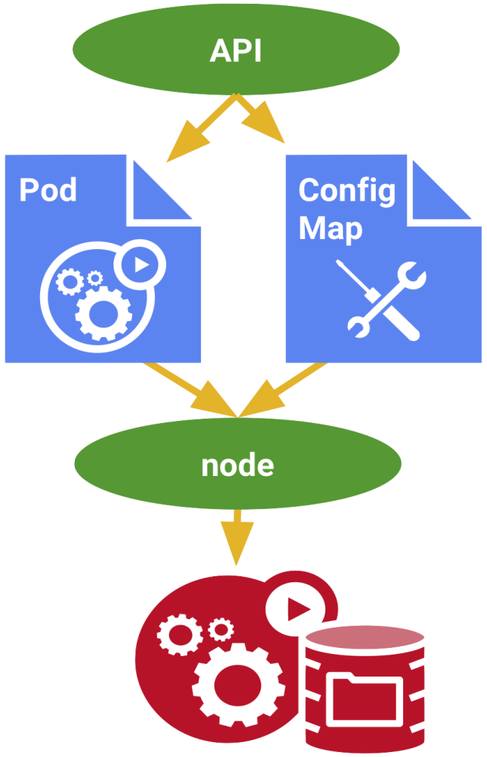
```
$ kubectl exec pod1 -it -- env | grep
ENVVAR
ENVVAR1=value1
ENVVAR2=value2
```

TECHNOLOGY

ConfigMap and Secrets

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.



- It allows decoupling of environment-specific configuration from the container images so that the applications are easily portable.
- It is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB.

ConfigMap Object

A ConfigMap is an API object that lets users store configurations for other objects to use. Unlike most Kubernetes objects that have a **spec**, a ConfigMap has **data** and **binaryData** fields.

The **data** field is designed to contain UTF-8 strings, while the **binaryData** field is designed to contain binary data as base64-encoded strings.

Each key under the **data** or the **binaryData** field must consist of **alphanumeric characters**, **hyphen** (**-**), **underscore** (**_**), or **period** (**.**).

The keys that are stored in the **data** must not overlap with the keys in the **binaryData** field.

ConfigMaps and Pods

```
apiVersion: v1
Kind: configmap
Metadata:
        name: game-demo
data:
        # property-like keys; each key map to a
simple value
        player_initial_lives: 3
        ui_properties_file_name: “user-
interface.properties”

        #file-like keys
        game.properties:
        enemy.types=aliens, monster
        players.maximum-lives=5
        user-interface.properties::
        color.good=purple1
        color.bad=yellow
        allow.textmode=true
```

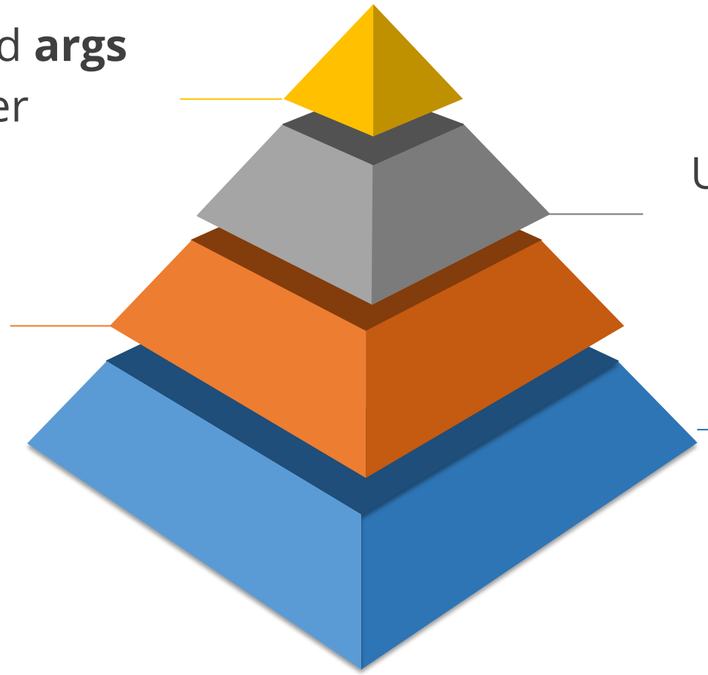
- The users can create a pod specification that refers to a ConfigMap.
- The spec can configure the container(s) in that pod based on the ConfigMap data.
- The pod and the ConfigMap must be in the same namespace.
- The example shows a ConfigMap that has some keys with single values and other keys where the value looks like a fragment of a configuration format.

ConfigMaps and Pods

There are four different ways that a **ConfigMap** is used to configure a container inside a pod:

Define **command** and **args** inside a container

Add a file in **read-only** volume for the application to read



Use **Environment variables** for a container

Write **code** to run inside the pod that uses the Kubernetes API to read a ConfigMap

Using ConfigMaps

ConfigMaps can be mounted as data volumes. They may also be used by other parts of the system, without having a direct exposure to the pod.

The most common way to use ConfigMaps is to configure settings for containers running in a pod in the same namespace. A ConfigMap can also be used separately.

Using ConfigMaps

To use ConfigMaps as files from a pod to consume a ConfigMap in a volume in a pod:

Create a ConfigMap or use an existing one

Modify the pod definition to add a volume under **.spec.volumes[]**

Add a **.spec.containers[].volumeMounts[]** to each container that needs the ConfigMap

Modify the image or command line so that the program looks for files in that directory

Using ConfigMaps

Each ConfigMap used needs to be referred to in **.spec.volumes**.

If there are multiple containers in the pod, each container needs its own **volumeMounts** block, but only one **.spec.volumes** is needed per ConfigMap.

Mounted ConfigMaps are updated automatically. When a ConfigMap currently consumed in a volume is updated, projected keys are updated as well.

The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap.

Immutable ConfigMaps

This feature is controlled by the **ImmutableEphemeralVolumes** feature gate. An immutable ConfigMap can be created by setting the **immutable** field to **true**.

```
apiVersion: v1
Kind: ConfigMap
Metadata:
                ....
Data:
                ....
Immutable: true
```

Once a ConfigMap is marked as immutable, it is not possible to revert the change or mutate the contents of the **data** or the **binaryData** field.

Secrets

Kubernetes Secrets store and manage credentials like passwords, tokens, and SSH keys. pods use Secrets by referencing them, and there are three ways to do this.

As files in a volume mounted on one or more of its containers

As container environment variables

When images are pulled for the pod by the kubelet

Types of Secrets

Kubernetes does not impose any constraints on the type name. However, if a built-in type is being used, all the requirements defined for that type must be met. The types are as follows:

- Opaque secrets
- SSH authentication secrets
- Service account token secrets
- TLS secrets
- Docker config secrets
- Bootstrap token secrets
- Basic authentication secret

Types of Secrets

Specify a Secret's type via the **type** field in the Secret resource or through kubectl flags, which affects its validations and Kubernetes-imposed constraints.

Built-in Type	Usage
Opaque	arbitrary user-defined data
kubernetes.io/service-account-token	service account token
kubernetes.io/dockercfg	serialized **~/.dockercfg** file
kubernetes.io/dockerconfigjson	serialized **~/.docker/config.json** file
kubernetes.io/basic-auth	credentials for basic authentication
kubernetes.io/ssh-auth	credentials for SSH authentication
kubernetes.io/tls	data for a TLS client or server
bootstrap.kubernetes.io/token	bootstrap token data

Creating a Secret

There are several options to create a Secret:

1. Use **kubectl** command
2. Use config file
3. Use kustomize

Editing a Secret

Example

```
# please edit the object below. Lines beginning with a # will be ignored,and
an empty file will abort the edit. If an error occurs while saving this file will
be
#reopened with the relevant failure

apiVersion: v1
data:
        username: YWRtaW4=
        password: MWYyZDFlMmU2NRm
Kind: Secret
Metadata:
        annotations: game-demo
        kubectl.kubernetes.io/last-applied-configuration: {..}
        creationTimestamp: 2016-01-22T18:41:56Z
name: mysecret
        namespace: default
        resourceVersion: “164619”
        uid: cfee02d6-c137-11e5-8d73-42010af00002
Type: Opaque
```

An existing Secret may be edited with the command:
kubectl edit secrets mysecret

This opens the default editor and allows to update the base64 encoded Secret values in the **data** field.

How to Use Secrets

Use Secrets as data volumes or env vars for pod containers; systems can also access Secrets indirectly.
Access Secret data within a pod:

Use a pre-existing Secret or create a new one

Change the pod definition to add a volume under **.spec.volumes[]**

Specify a **.spec.containers[].volumeMounts[]** to each container that needs the Secret

Alter the command line or image so that the program looks for files in the specific directory

Usage of Secrets as Environment Variables

To use a Secret in a pod's environment variable:

Create a Secret or use an existing one. Multiple pods can reference the same Secret

Modify the pod definition in each container. The environment variable that consumes the Secret key should populate the Secret's name and key in **env[].valueFrom.secretKeyRef**

Modify the image or command line so that the program looks for values in the specified environment variables

Using Secrets

An example of a pod that uses Secrets from environment variables:

```
apiVersion: v1
Kind: pod
Metadata:
 name: secret-env-pod
Spec:
 containers:
  name: mycontainer
  image: redis
  env:
   name: SECRET-USERNAME
   valueFrom:
    secretKeyRef:
     name: mysecret
     key: username
    name: SECRET_PASSWORD
     valueFrom:
      secretKeyRef:
       name: mysecret
       key: password
Restartpolicy: Never
```

Using Secrets

The Secret keys appear as normal environment variables inside a container that consumes a Secret in the environment variables. The following is the result of commands executed inside the container:

echo $SECRET_USERNAME

echo $SECRET_PASSWORD

The output is like:

Admin

1f2d1e2e67df

Environment variables are not updated after a Secret update.

Immutable Secrets and ConfigMaps

The Kubernetes feature **Immutable Secrets and ConfigMaps** provides an option to set individual Secrets and ConfigMaps as immutable:

1. Protects from accidental (or unwanted) updates that could cause applications outages

2. Improves performance of the cluster by significantly reducing load on kube-apiserver

Understanding Config Maps and Secrets

Duration: 10 mins

Problem Statement:

You've been asked to create secrets using kubectl by adding a config map.

ASSISTED PRACTICE

Assisted Practice: Guidelines

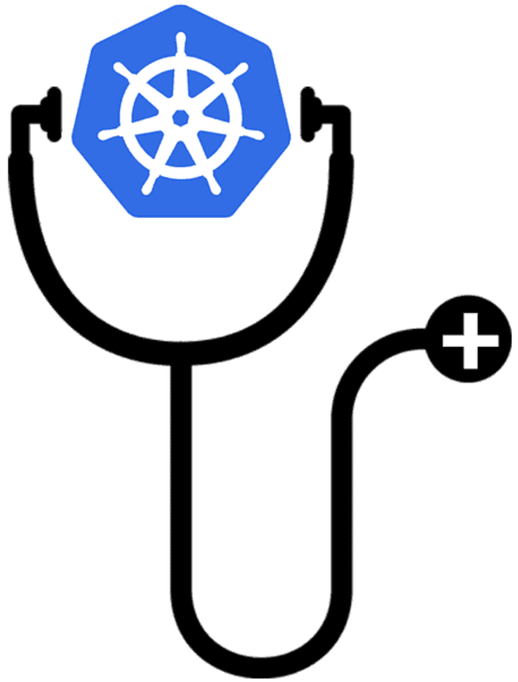
Steps to be followed:

1. Add a config map entry to the pod
2. Create secrets using kubectl

Health Monitoring

Overview

Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as **NodeCondition** and **Event**.



Recommendations and Restrictions

When running the Node Problem Detector, expect an extra resource overhead on each node.

01 A resource limit is set for the Node Problem Detector.

02 The kernel log grows relatively slowly.

03 Even under high load, the resource usage is acceptable.

Achieving Scalability

Overview

When deploying a horizontally scalable application to Kubernetes, ensure the following configurations:

1. Resource limits and requests
2. Node and pod, affinities and anti-affinities
3. Health checks
4. Deployment strategies
5. pod disruption budgets
6. Horizontal pod autoscaling

Resource Requests and Limits

Resource requests and limits dictate how much memory and CPU the application may use.

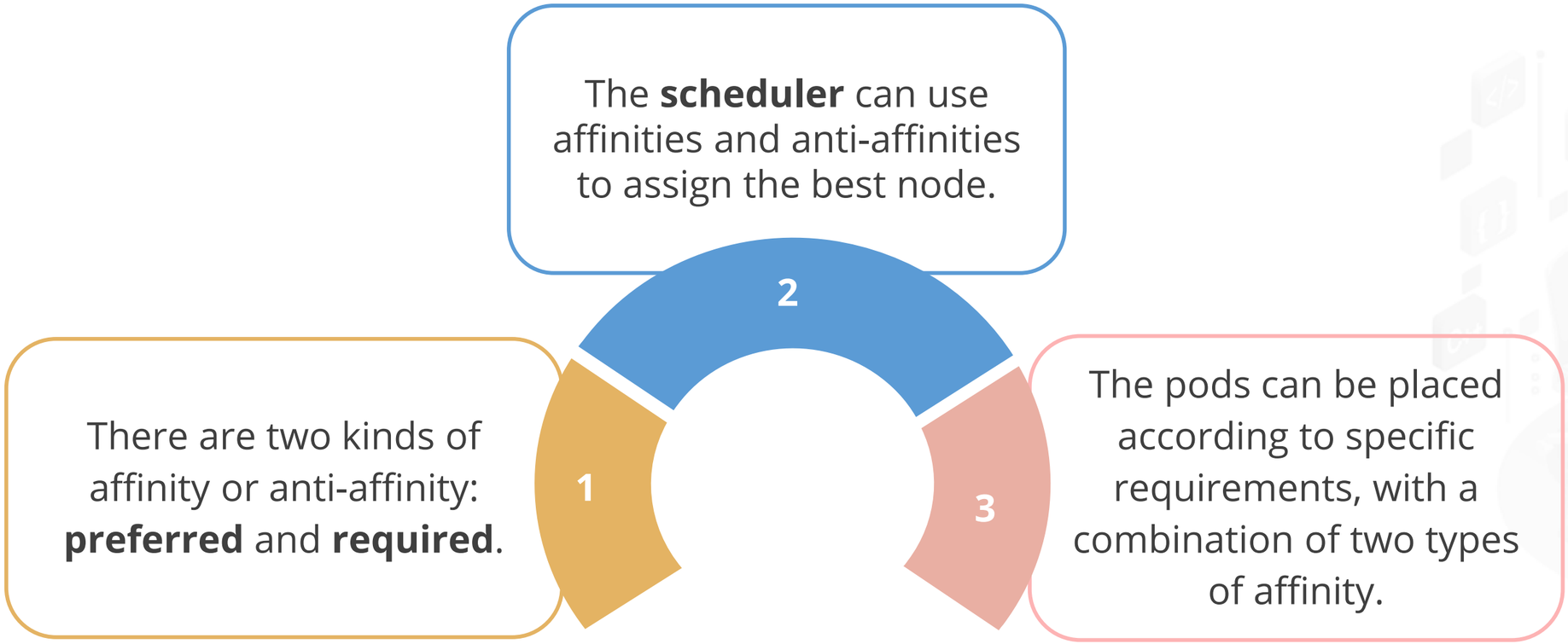
It allows **autoscalers** to estimate the capacity and ensure that the cluster expands (and contracts) as the demand changes.

It is vital to set limits and requests on a **Deployment**.

If there are no limits, the Kubernetes **scheduler** cannot ensure that workloads are spread evenly across the nodes.

Node and Pod, Affinities and Anti-affinities

Affinities and anti-affinities enable users to distribute or restrict workload based on a set of rules or constraints.



Scaling Applications

Kubernetes offers scalability by auto-adjusting workloads based on past resource use, with autoscaling spanning three dimensions.

Horizontal pod Autoscaler (HPA)	Adjusts the number of replicas of an application
Cluster Autoscaler	Adjusts the number of nodes of a cluster
Vertical pod Autoscaler (VPA)	Adjusts the resource requests and limits of a container

Scaling Applications

The autoscalers operate on one of two Kubernetes layers:

Pod level

The **HPA** and **VPA** methods take place at the pod level. Both HPA and VPA will scale the available resources or instances of the container.

Cluster level

The **cluster autoscaler** falls under the cluster level, where it scales up or down the number of nodes inside the cluster.

Health Checks

Kubernetes has two types of health checks:

Readiness Probe	Notifies Kubernetes that the pod is ready to start receiving requests
Liveness Probe	Notifies Kubernetes that the pod is still running as expected

Configuring Liveness Probes

Duration: 5 mins

Problem Statement:

You've been asked to create and configure pod using liveness probes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pod using liveness probes
2. Describe the pod

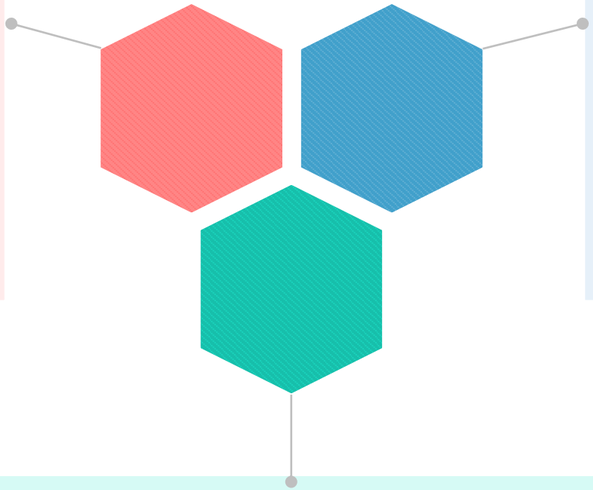
Pod Disruption Budgets

Disruptions in Kubernetes clusters are the norm, not the exception.

Reasons for disruptions:

A spot instance on AWS must be taken out of service.

.



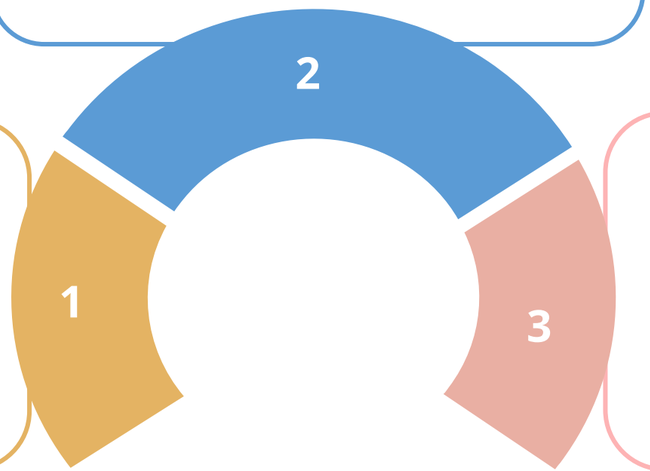
An autoscaler removes any underutilized node.

The infrastructure team must apply a new config so that they can start replacing nodes within a cluster.

Pod Disruption Budgets

The job of the pod disruption budget is to ensure that there is always a minimum number of pods ready for Deployment.

With pod disruption budget, it is possible to specify **minAvailable** or **maxUnavailable**.



The values may be a percentage of the desired **replica count** or an **absolute number**.

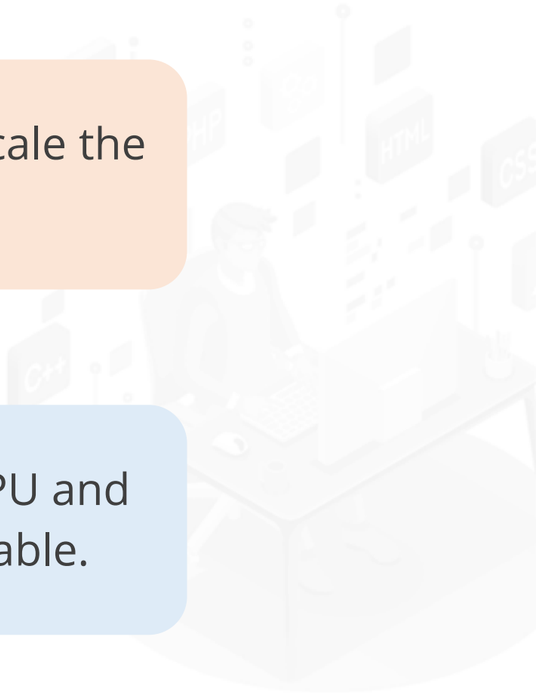
With pod disruption budget, Kubernetes can reject **voluntary disruptions**.

Horizontal Pod Autoscalers

Kubernetes control plane can scale applications based on their resource utilization. As pods become busier, the control plane brings up new replicas to share the load automatically.

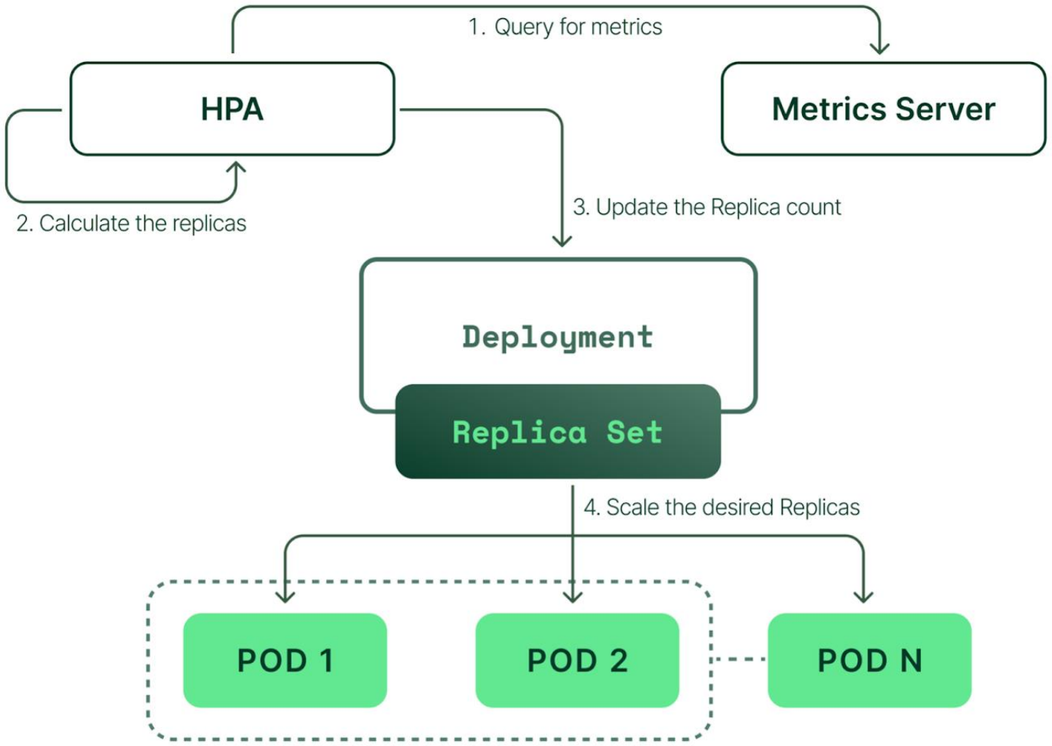
Periodically, the controller monitors metrics provided by the metrics-server to scale the deployment up or down based on the pods' load.

It is possible to scale while configuring a Horizontal pod Autoscaler based on CPU and memory usage. But custom metrics servers can help extend the metrics available.



Using HPA for Dynamic Workload Management

HPA works in a **check, update, and check again** style loop.



Each of the steps in the loop works as follows:

1. Monitors the metrics server for resource usage
2. Calculates the desired number of replicas required based on the collected resource usage
3. Decides to scale up the application to the desired number of replicas
4. Changes the desired number of replicas

Since HPA is continuously monitoring, the process repeats from Step 1.

Understanding Horizontal Pod Auto Scaling

Duration: 10 mins

Problem Statement:

You've been asked to create a horizontal pod autoscaler.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create an HPA in the master node
2. Check the Deployment Service
3. Verify HPA

Building Self-Healing Pods with Restart Policies

Introduction: No Time for Downtime

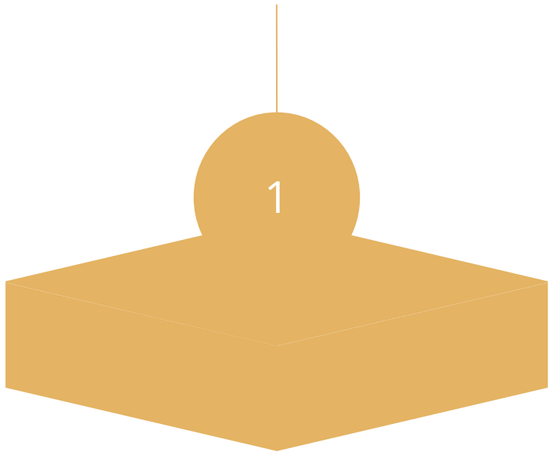
An efficient application must run non-stop, 24/7, despite technical problems, updates, or natural events.

Kubernetes facilitates the smooth working of the application by abstracting machines physically. It is a container orchestration tool. The pods and containers in Kubernetes are self-healing.

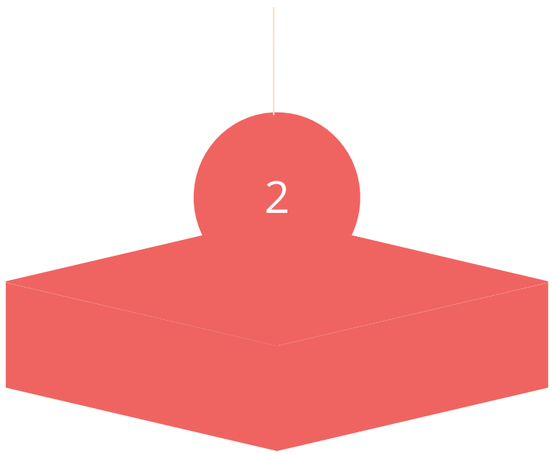
Pod Readiness

An application can inject extra feedback or signals into podStatus: **pod readiness**.

To use this, set **readinessGates** in the pod's **spec** to specify a list of additional conditions that the **kubelet** evaluates for pod readiness.



Readiness gates are determined by the current state of **status.condition** fields for the pod.



Status for Pod Readiness

In case of a pod that uses custom conditions, it is evaluated to be ready only when both the statements given below apply:

All containers in the pod are ready.

All conditions specified in **readinessGates** are **true**.

The Three Container States-1

Waiting: Created but not running

A container in the waiting stage can still run operations such as pulling images or applying Secrets. To check the status of waiting, use:

```
kubectl describe pod [podname]
```

Along with the pod's state, a message and reason for the state are displayed as:

```
...
State: Waiting
Reason: ErrImagePull
...
```

The Three Container States-2

Running:

These are the containers running without issues. The given command is run before the pod enters the running state:

postStart

Running pods display the time of starting of the container:

```
...
State: Running
Started: Sat, 30 Jan 2021 16:48:38 +0530
...
```

The Three Container States-3

Terminated:

It is a container that has failed or completed its execution. The given command is executed before the pod is moved to be terminated:

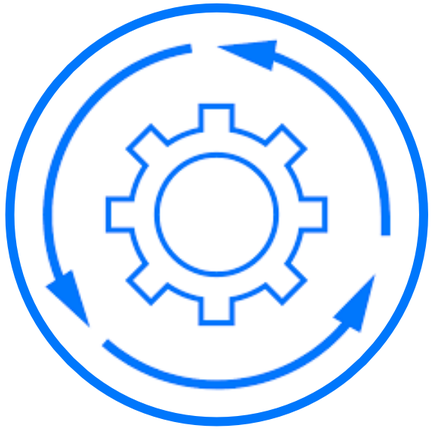
preStop

Terminated pods will display the time of the entrance of the container:

```
...
State: Terminated
Reason: Completed
Exit Code: 0
Started: Mon, 11 Jan 2021 12:23:45 +0530
Finished: Mon, 18 Jan 2021 21:32:54 +0530
...
```

Self-Healing in Kubernetes

The pod phase in Kubernetes offers insights into the pod's placement. It can have:



- **Pending pods:** The container is waiting to be started.
- **Running pods:** The container has been created, and processes are running in it.
- **Succeeded pods:** The pods have completed the container life cycle.
- **Failed pods:** The container either exited with non-zero status or was terminated by the system.
- **Unknown pods:** The state of the pod could not be obtained.

Container Probes

The kubelet can optionally perform and react to three kinds of probes on running containers:

livenessProbe	Indicates whether the container is running
readinessProbe	Indicates whether the container is ready to respond to requests
startupProbe	Indicates whether the application within the container is started

Self-Healing in Kubernetes

Each probe provides one of three results:

Failure	The container failed the diagnostic.
Unknown	The diagnostic failed, so no action can be taken.
Success	The container passed the diagnostic.

Self-Healing in Kubernetes

There are four distinct ways to inspect a container using a probe. Each probe must specify one of the following four mechanisms:

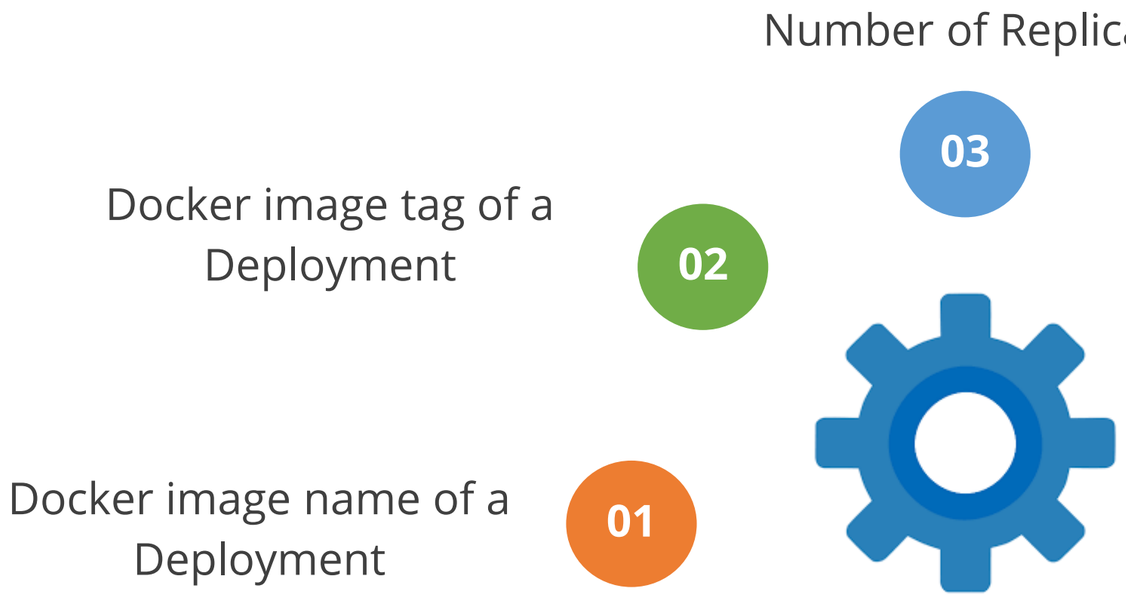
exec	Executes a specified command inside the container
grpc	Performs a remote procedure call using gRPC
httpGet	Performs an HTTP request against the pod's IP address on a specified port and path
tcpSocket	Performs a TCP check against the pod's IP address on a specified port

TECHNOLOGY

Manifest Management and Common Templating Tools

Overview

There is a need to use templates in Kubernetes manifests for common parameters. These include:



Overview

Kubernetes does not have any default templating mechanism. Deployed manifests are static YAML files. If the parameters need to pass in the manifests, an external solution is needed.

Helm is the package manager for Kubernetes. It has templating capabilities. It is the Kubernetes equivalent of **yum** or **apt**.

Helm deploys charts. It is a collection of all pre-configured, versioned application resources that may be deployed as a single unit.

Key Takeaways

- Node Isolation is used to ensure that only specific pods run on nodes with certain isolation, security, or regulatory properties.
- Topology spread constraints manage pod distribution across a cluster's failure domains like regions and nodes.
- A ReplicaSet ensures that a specified number of pod replicas are running at any given time.
- The Kubernetes control plane scales apps by resource use, auto-adding replicas for busier pods.

MySQL and WordPress Installation in Kubernetes

Duration: 30 Min

Project agenda: To create a highly available MySQL and WordPress deployment using autoscaling.

Description:

Your team is going to deploy MySQL and WordPress containers. All the users should be added using ConfigMaps and all the sensitive data should be added using Secrets. The service should be on the NodePort. The WordPress pod should not deploy if the MySQL service is not deployed. Also, ensure autoscaling is on.

LESSON-END PROJECT

MySQL and WordPress Installation in Kubernetes

Duration: 30 Min

Steps to perform:

1. Creating a cluster
2. Creating a Secret for storing the MySQL password securely
3. Creating a MySQL YAML manifest file for deploying MySQL
4. Deploying the WordPress application and designing the manifest YAML file
5. Accessing the WordPress application using the NodePort
6. Enabling autoscaling on a WordPress pod to ensure scaling if the CPU utilization is greater than 50 percent

LESSON-END PROJECT