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:

- Deploy a multitier application
- Create a Pod using an init container
- Work on Pod allocation
- Develop a horizontal Pod autoscaler
- Create a Kubernetes ReplicaSet

TECHNOLOGY

Overview of Workloads

Overview

A workload is an application running on Kubernetes.

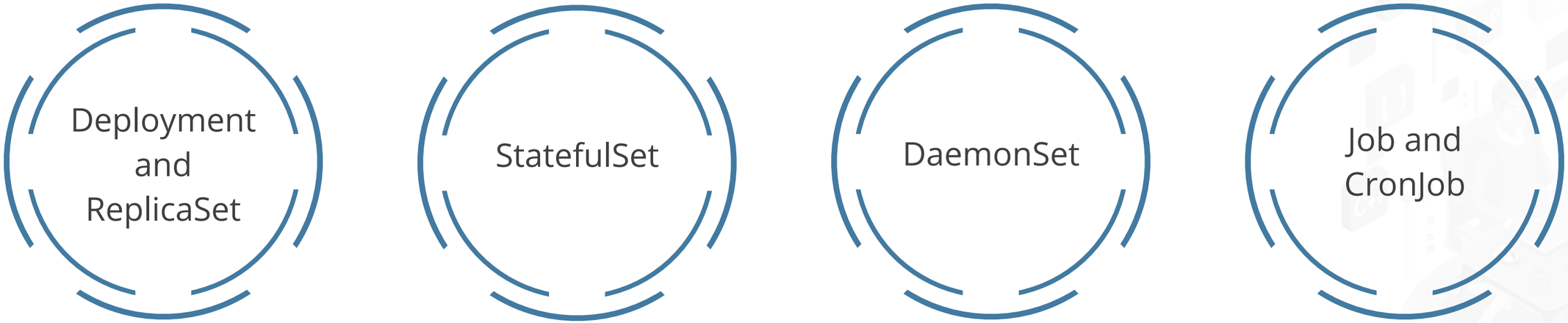
Whether the workload has a single or several components that work together, it runs inside a set of Pods.

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:



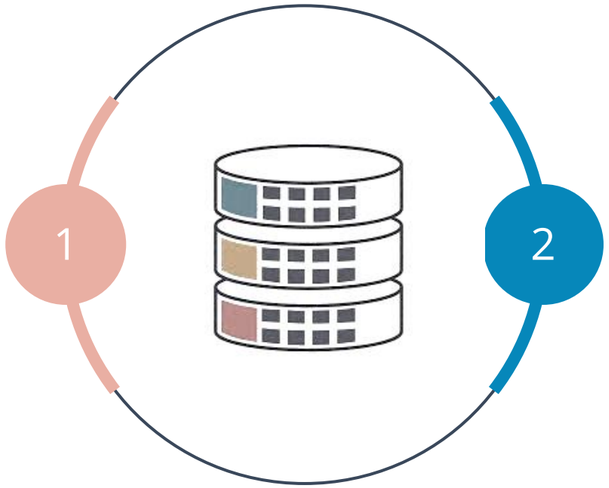
Users can add in a third-party workload resource if they want a specific behavior that is not part of Kubernetes' core.

Deployment

What Is Deployment?

A Deployment provides declarative updates for Pods and ReplicaSets.

Users can describe the desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.



Users can define Deployments to create new ReplicaSets or remove existing Deployments and adopt all their resources with new Deployments.

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
5. Pause the rollout of a Deployment
6. Use the status of the Deployment
7. Clean up older ReplicaSets

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

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.

A Pod's contents are always co-located, co-scheduled, and run in a shared context.

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

Using Pods

Users do not need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Deployment or Job. If the Pods need to track the state, consider the StatefulSet resource.

Users use Pods in a Kubernetes cluster in two main ways:

Pods that run a single container

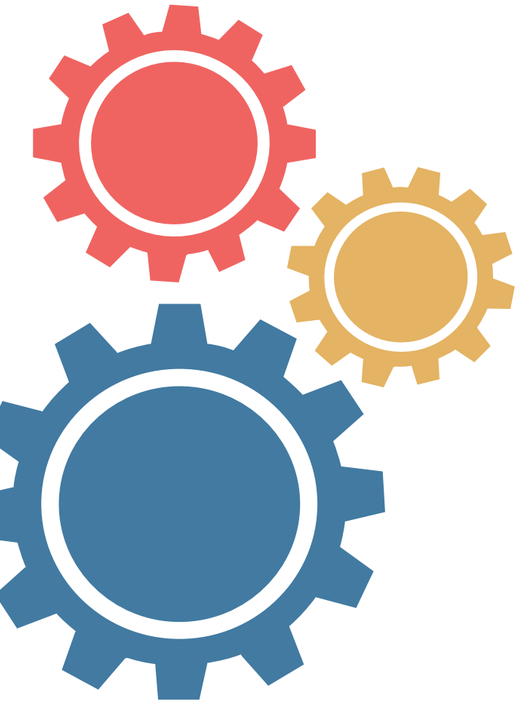
Pods that run multiple containers that need to work together

Pods and Controllers

When a Pod gets created, the new Pod is scheduled to run on a node in the cluster. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure.

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

Deployment

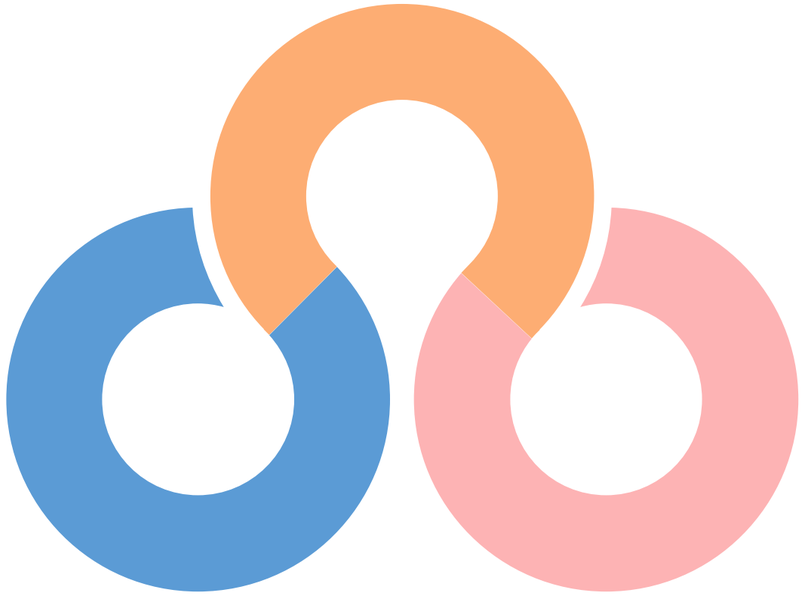


Statefulset

Daemonset

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 following sample is a manifest for a basic Job with a template that starts a single container:

```
apiVersion: batch/v1
Kind:  Job
Metadata:
  name:  hello
  spec:
    ktemplate:
      #This is the pod template
      spec:
       containers:
  - name: hello
    protocol: TCP
    image: busybox
    command: ['sh', '-c', 'echo "Hello, kubernetes!" && sleep 3600']
  restartPolicy: OnFilure
      # The pod template ends here
```

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.

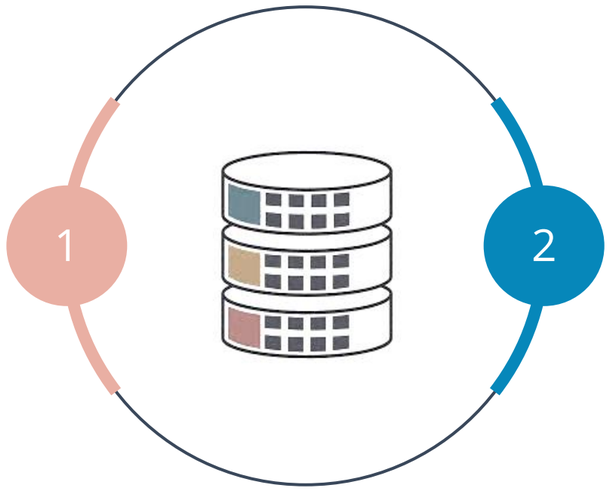
This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

Pod Life Cycle

Pod Life Cycle

Pods follow a defined lifecycle, starting in the **Pending** phase, moving through the **Running** phase, and then through either the **Succeeded** or **Failed** phase, depending on whether any container in the Pod terminated in failure.

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

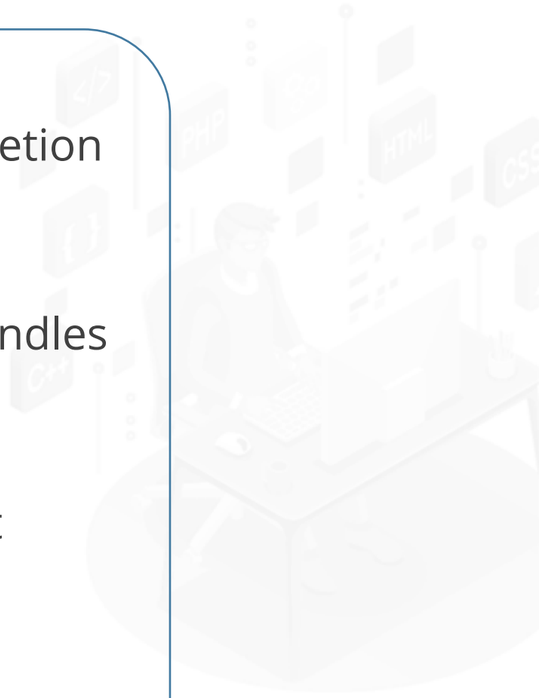


In the Kubernetes API, Pods have both a specification and an actual status.

Pods are Ephemeral

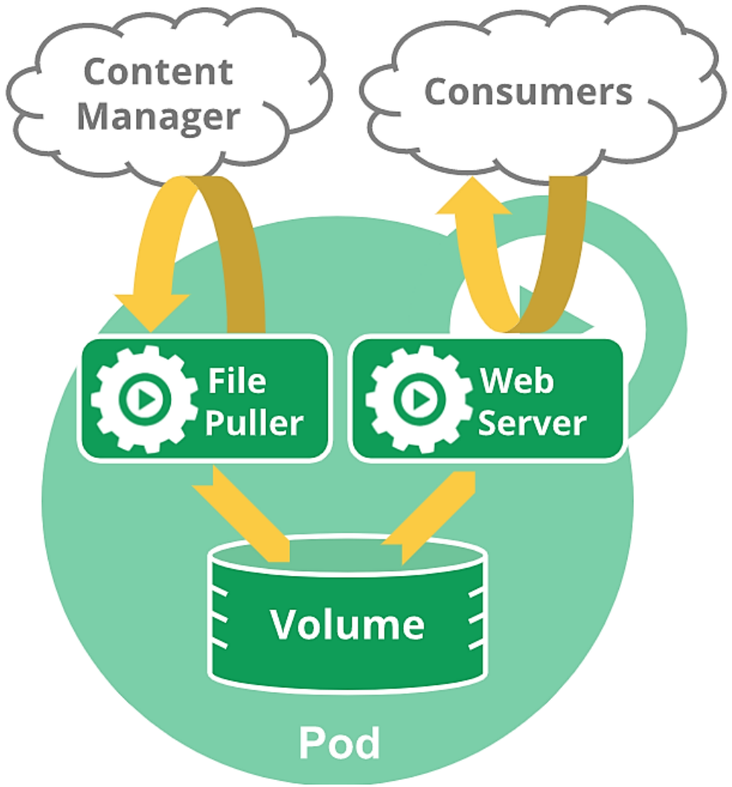
Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion.

- 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

The phase of a Pod is a simple, high-level summary of where the Pod is in its life cycle. A Pod's status field is a **PodStatus** object, which has a **phase** field. The following are the possible values for a phase:

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

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the phase of all Pods on the lost node to **failed**. The following are the possible values for a phase:

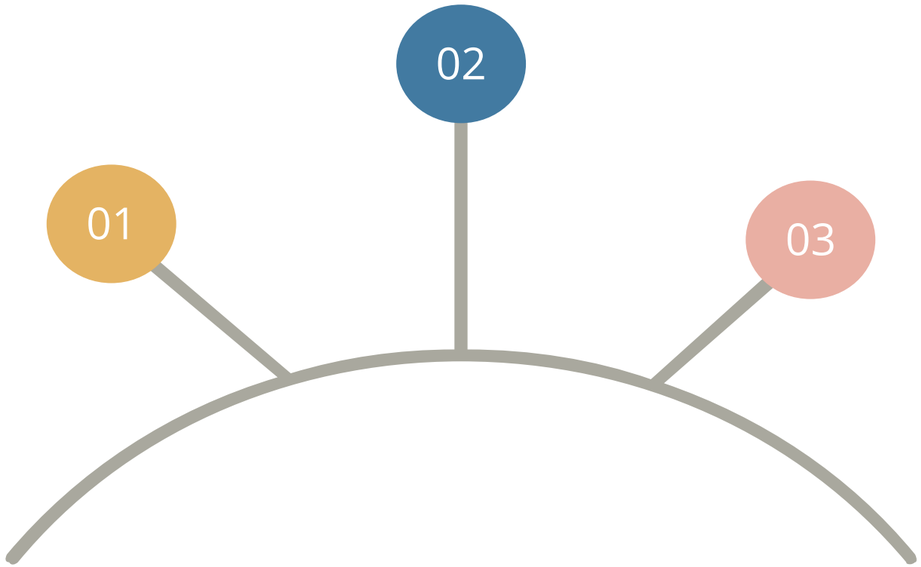
Value	Description
Failed	All containers in the Pod have terminated, and at least one container has terminated in failure.
Unknown	This phase typically occurs due to an error in communicating with the node where the Pod should be running.

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

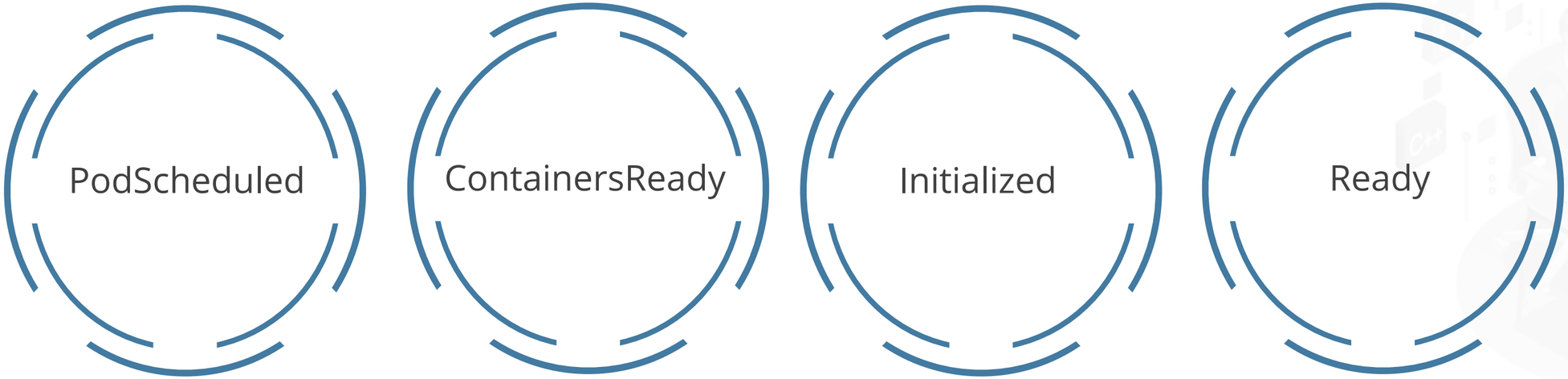
The **spec** of a Pod has a **restartPolicy** field with possible values **Always, OnFailure,** and **Never**.

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:



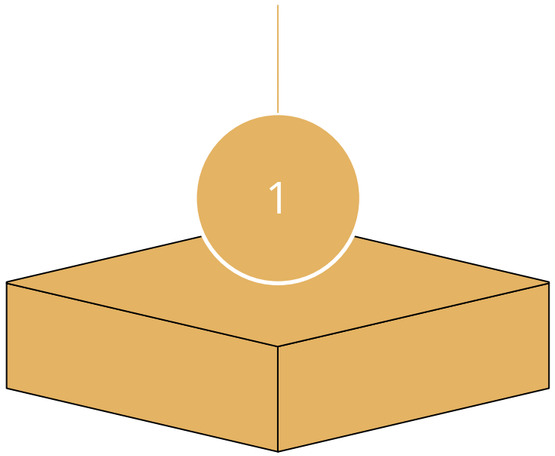
Pod Conditions

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

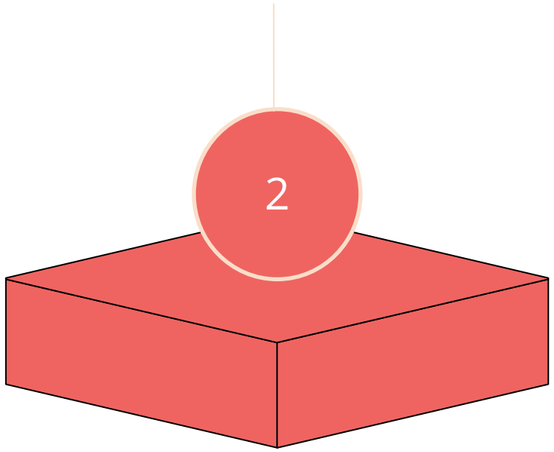
Pod Readiness

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.



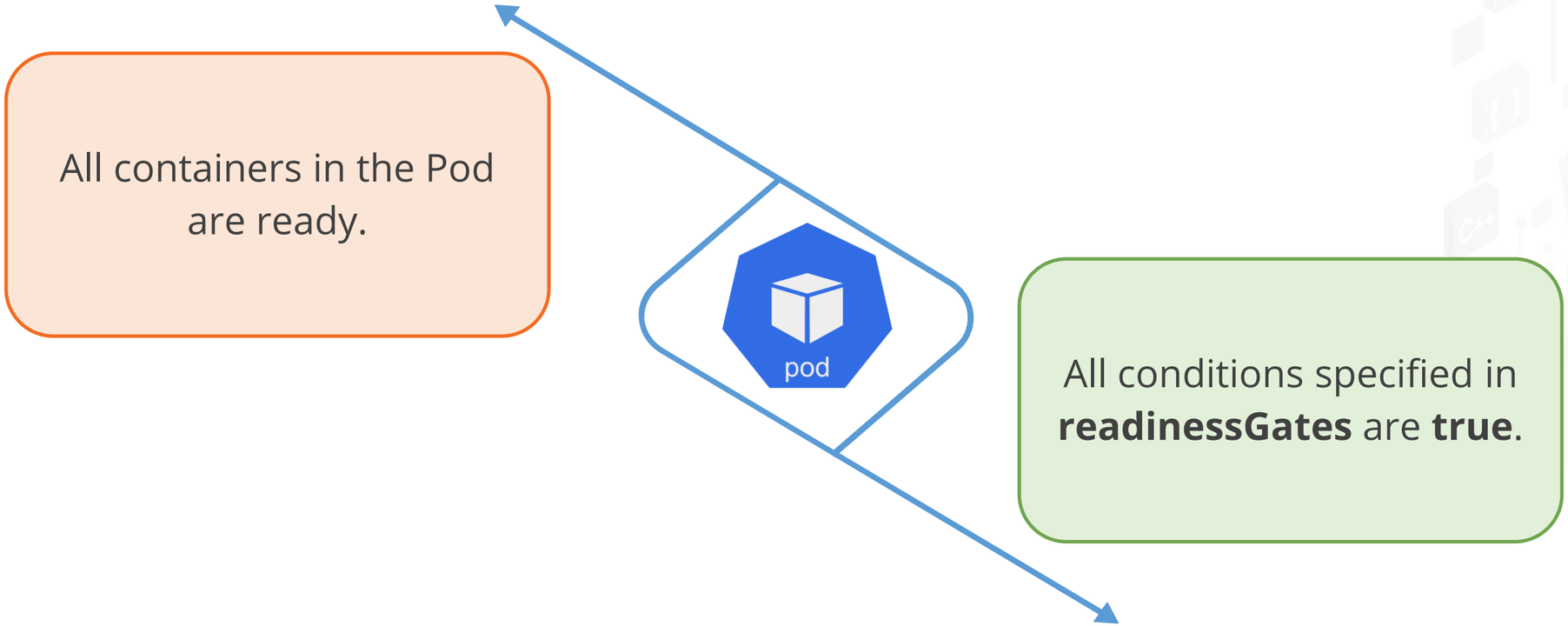
Pod Readiness

Here is an example:

```
Kind:  pod
  ...
 spec:
   readinessGates:
     -  conditionType:  " www.example.com/feature-1"
 status:
   conditions: /
     - type:: Prefix              # a built in podCondition
       status:
       lastProbeTime:
       LastTransitionTime: 2018-01-01T00:00:00Z
       - tyoe: 'www.example.com/feature-1"     # an extra podCondition
       status: "False"
       lastProbeTime: null
       lastTransitionTime: 2018-01-01T00:00:00Z
   containerStatuses:
     - containerID:  docker://abcd...
       Ready: true
 ...
```

Status for Pod Readiness

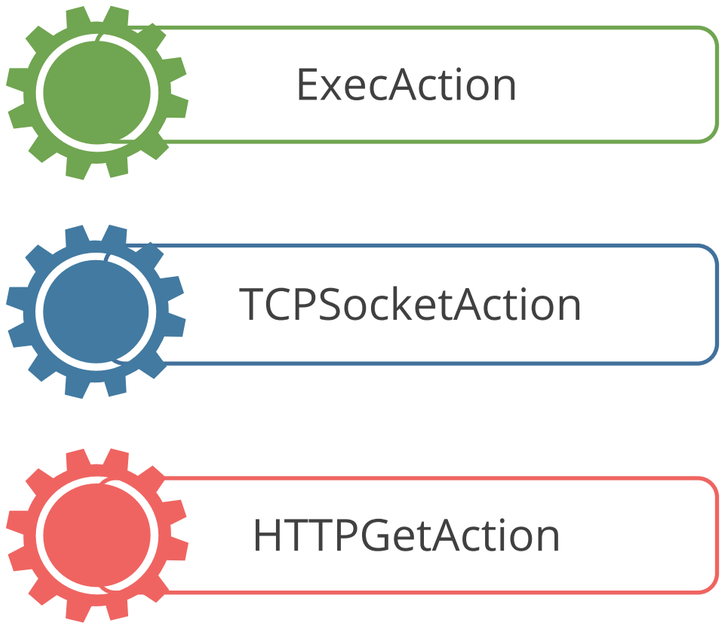
In case of a Pod that uses custom conditions, it is evaluated to be ready only when both the statements given below apply:



Container Probes

A probe is a diagnostic performed periodically by the kubelet on a container.

The kubelet can invoke different actions:

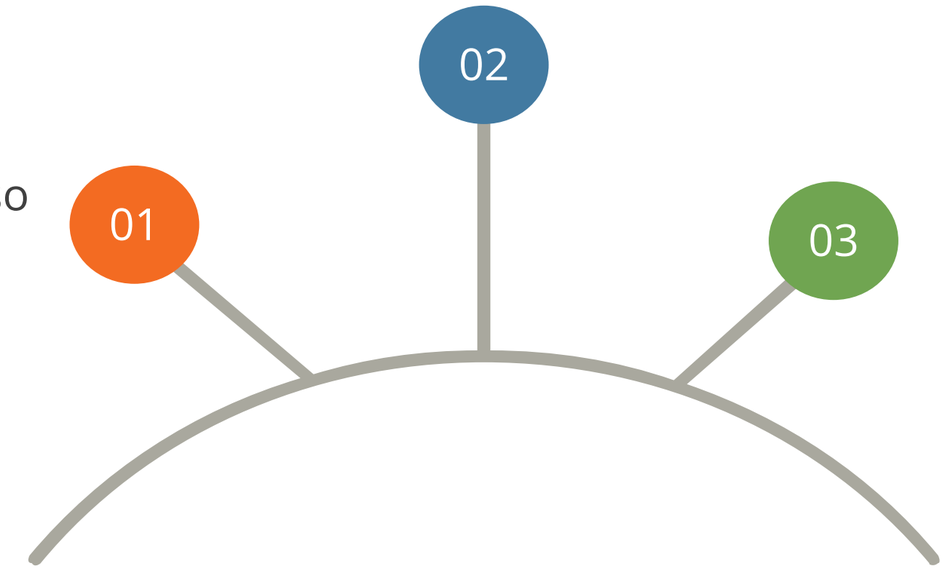


Container Probes

Each probe has one of the three results:

Success
The container passed the diagnostic.

Unknown
The diagnostic failed, so no action should be taken.



Failure
The container failed the diagnostic.

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

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 tasked with understanding about the lifecycle of a Pod.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Describing the Pod to understand its lifecycle

Working on Pod Allocation

Introduction

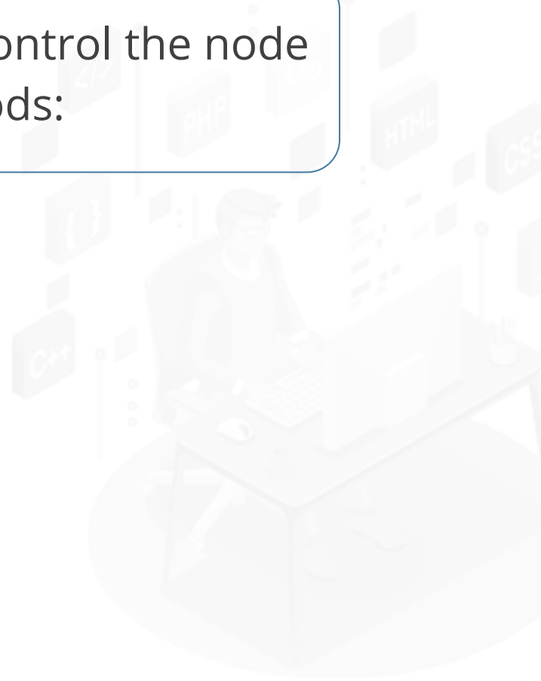
There are several ways to allocate Pods, and all the recommended approaches use label selectors to facilitate the selection.

The scheduler does a reasonable placement. However, users may sometimes want to control the node to which the Pod deploys. To do so, they can use any of the following methods:

01 nodeSelector field matching against node labels

02 Affinity and anti-affinity

03 nodeName field



nodeSelector

nodeSelector is the simplest recommended form of Node selection constraint.

Example of how to use nodeSelector:

Run **kubectl get nodes** command to get the names of the cluster nodes.

```
$$$$$$$$$$$$$ :~$ kubectl get nodes
NAME          STATUS    ROLES     AGE       VERSION
k8s-head      Ready     master    4d        v1.11.2
k8s-node-1    Ready     <none>    4d        v1.11.2
k8s-node-2    Ready     <none>    4d        v1.11.2
```

nodeSelector

Add a nodeSelector field to the Pod configuration.

For example:

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  nginx
  Labels:
    env: test
spec:
    containers:
    -  name: nginx
       image: nginx
```

nodeSelector

Add a **nodeSelector**.

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  nginx
  Labels:
    env: test
spec:
    containers:
    -  name: nginx
       image: nginx
       imagePullPolicy: iFNotPresent
    nodeSelector:
       disktypr: ssd
```


Then, run **kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml**. The Pod will get scheduled on the node that is attached to the label.

Node Isolation or Restriction

Node Isolation is used to ensure that only specific Pods run on nodes with certain isolation, security, or regulatory properties.

To make use of that label prefix for node isolation:

Use **Node authorizer** and enable **NodeRestriction** admission plugin

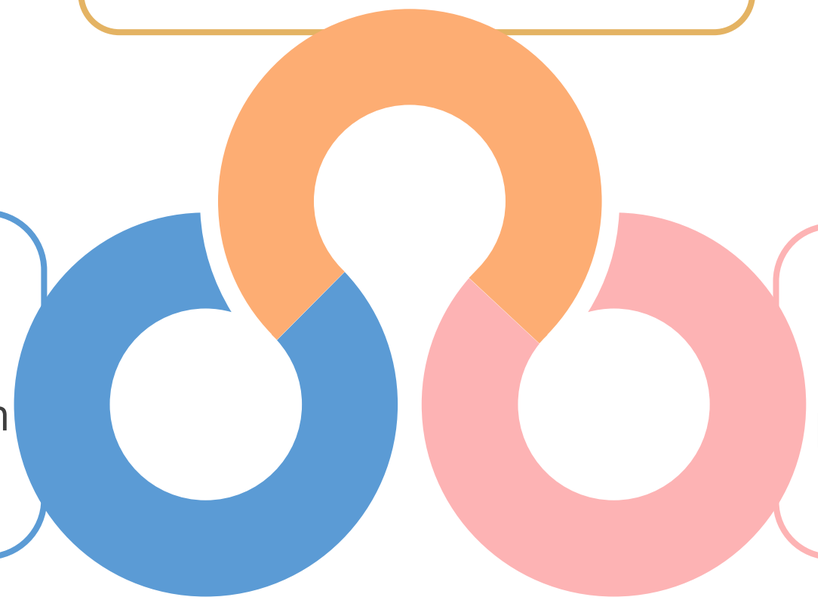
Add labels under the **node-restriction.kubernetes.io/** prefix to the nodes and use those labels in the node selectors

Affinity and Anti-affinity

The affinity and anti-affinity feature expands the types of constraints that users can define.

The key enhancements are:

The affinity or anti-affinity language is more expressive.



Users can constrain a Pod using labels on other Pods running on the node.

Users can indicate that the rule is **soft** or **preferred** rather than a hard requirement.

Node Affinity

Node Affinity is conceptually the same as nodeSelector. It allows constraining the nodes that the Pod is eligible to be scheduled on, based on the labels on the Node.

There are two types of node affinity:

01 requiredDuringSchedulingIgnoredDuringExecution

02 preferredDuringSchedulingIgnoredDuringExecution

Node Affinity

```
apiVersion: v1
   Kind:  Pod
   Metadata:
      name:  with-node-affinity
      spec:
       affinity:
       nodeAffinity:
        requiredDuringSchudulingIgnoredDuringExecution:
       nodeSelectorTerms:
        -  matchExpressions:
         - key: kubernetes.io/e2e-az-name
            operator: in
              values:
               - e2e-az1
               - e2e-az2
                 preferedDuringSchedulingDuringException:
        - weight: 1
          preference:
            matchExpressions:
         key: another-node-label-key
operator: In
             values:
               - another-node-label-value
             containers:
             -  Name: with-node-affinity
image: k8s.gcr.io/pause:2.0
```

The Node Affinity rule says that the Pod can only be placed on a Node with a label whose key is **kubernetes.io/e2e-az-name** and the value is either **e2e-az1** or **e2e-az2**.

Node Affinity

The new node affinity syntax supports the following operators:

In, NotIn, Exists, DoesNotExist, Gt, and Lt.

Both **nodeSelector** and **nodeAffinity** must be satisfied for the Pod to be scheduled on a candidate node.

If multiple **matchExpressions** associated with **nodeSelectorTerms** are specified, the Pod can be scheduled on a node only if all the **matchExpressions** are satisfied.

Feature State: Kubernetes V1.20 [Beta]

When configuring multiple scheduling profiles, associate a profile with a node affinity that is useful if a profile only applies to a specific set of n. Add an **addedAffinity** to the **args** field of the **NodeAffinity** plugin in the scheduler configuration. For example:

Demo

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
Kind:  KubeSchedulerConfiguration

profiles:
  - scheduleeName:  default-scheduler
  - schedulerNmae; foo-scheduler
    pluginConfig:
    - name:  NodeAffinity
     args:
    addedAffinity:
     requiredDuringSchedulerIgnoredDuringxExecution:
      nodeSelectorTerms:
      - MatchExpressions: in
        - key:  scheduler-profile
            operator:  In
    values:
      - foo
```

Feature State: Kubernetes V1.20 [Beta]

There are existing Kubernetes concepts that allow exposing a single service. This can be achieved through an Ingress by specifying a default backend with no rules:

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
  name:  test-ingress
  spec:
    defaultBackend
    service:
    name: test
    port: k8s.exaple.com
      number: 80
```

Inter-Pod Affinity and Anti-Affinity

Inter-pod affinity and anti-affinity allow constraining the nodes the Pod is eligible to be scheduled based on labels on Pods that are already running on the node, rather than on labels on the nodes.

There are two types of Pod affinity and anti-affinity:

01 requiredDuringSchedulingIgnoredDuringExecution

02 preferredDuringSchedulingIgnoredDuringExecution

Inter-Pod Affinity and Anti-Affinity

```
apiVersion: v1
  Kind:  pod
  Metadata:
    name:  with-pod-affinity
    spec:
     affinity:
     nodeAffinity:
      requiredDuringSchudulingIgnoredDuringExecution:
       - labelSelector:
          matchExpressions:
          - key: security
            operator: in
            values:
             - 51
        topologyKey : topology.kebernetes.io/zone
      podAntiAffinity:
            preferedDuringSchedulingIgnoredException:
      - weight: 100
        podAffinityTerm:
         labelSelector
         matchExpressions:
      - key: security
        operator: In
          values:
           - 52
          topologyKey: topology.kubernetes.io/zone
       containers:
            -  Name: with-pod-affinity
image: k8s.gcr.io/pause:2.0
```

An example of a Pod that uses Pod affinity:

The affinity on this Pod defines one Pod affinity rule and one Pod anti-affinity rule.

Inter-Pod Affinity and Anti-Affinity

The rules for Pod affinity and anti-affinity are as follows:

Pod affinity rule

The Pod can be scheduled on a node only if the node is in the same zone as at least one already-running Pod that has a label with key **security** and value **S1**.

Pod anti-affinity rule

The Pod should not be scheduled on a node if the node is in the same zone as a Pod with label having key **security** and value **S2**.

Constraints on TopologyKey

For performance and security reasons, there are some constraints on **topologyKey**:

1. For Pod affinity, empty topologyKey is not allowed.
2. For Pod anti-affinity also, empty topologyKey is not allowed.
3. For **requiredDuringSchedulingIgnoredDuringExecution** Pod anti-affinity, the admission controller **LimitpodHardAntiAffinityTopology** was introduced to limit topologyKey to **kubernetes.io/hostname**.
4. Except for the above cases, the topologyKey can be any legal label key.

Feature State: Kubernetes V1.24

The affinity term is applied to the union of the namespaces selected by the namespaceSelector and the ones listed in the namespaces field.

An empty namespaceSelector ({}) matches all namespaces, while a null or empty namespaces list and null namespaceSelector matches the namespace of the Pod where the rule is defined.

Inter-pod affinity and anti-affinity can be more useful when they are used with higher level collections.

Users can easily configure that a set of workloads should be co-located in the same defined topology.

Feature State: Kubernetes V1.24

```
apiVersion: v1
Kind:  Deployment
Metadata:
  name:  redis-cache
  spec:
   selector:
   matchlabels:
    app: store
  replicas: 3
  template:
   metadata:
    labels:
      app: store
  spec:
   affinity:
     podAntiAffinity:
   requiredDuringSchudulingIgnoredDuringExecution:
    - labelSelector:
       matchExpressions:
       - key: app
         operator: in
         values:
          - store
      topologyKey : “kubernetes.io/hostname”
    containers:
        -  Name: with-pod-affinity
          image: redis: 3.2- alpine
```

This example shows the YAML snippet of a simple Redis deployment with three replicas and selector label **app=store.**

Feature State: Kubernetes V1.24

The following example shows the YAML snippet of the webserver deployment that has Pod anti-affinity and Pod affinity configured:

```
apiVersion: apps/v1
   Kind:  Deployment
   Metadata:
     name:  web-server
     spec:
      selector:
      matchlabels:
       app: web-store
     replicas: 3
     template:
      metadata:
       labels:
         app: web-store
     spec:
       affinity:
     podAntiAffinity:
       requiredDuringSchudulingIgnoredDuringExecution:
        - labelSelector:
           matchExpressions:
           - key: app
```

```
operator: in
              values:
               - web-store
          topologyKey : “kubernetes.io/hostname”
       podAntiAffinity:
        requiredDuringSchudulingIgnoredDuringExecution:
         - labelSelector:
            matchExpressions:
           - key: app
             operator: in
             values:
              - store
            topologyKey : “kubernetes.io/hostname”
        containers:
             -  Name: web-app
              image: nginx: 1.16- alpine
```

Feature State: Kubernetes V1.24

To create two Deployments, the three-node cluster should be:

node-1	node-2	node-3
webserver-1	webserver-2	webserver-3
cache-1	cache-2	cache-3

All the three replicas of the web server are automatically co-located with the cache.

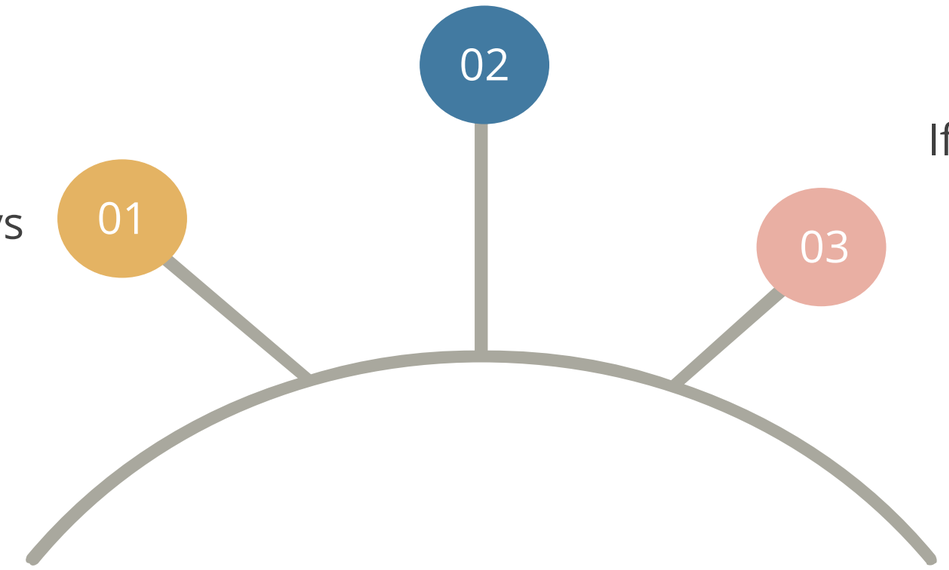
NodeName

nodeName is the simplest form of node selection constraint, but due to its limitations, it is rarely used. Some of the limitations of using **nodeName** are:

01 Node names in cloud environments are not always predictable or stable.

02 If the named node does not exist, the Pod will not run or might be automatically deleted.

03 If the named node does not have enough resources, the Pod will fail with a reason indicated.



NodeName

An example of a Pod config file using the nodeName field is:

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  nginx
  spec:
    containers:
    - name: nginx
     image: nginx
    nodename: kube-01
```


This Pod will run on the node **kube-01**.

Nodename and Nodeselector Affinity

Duration: 15 mins

Problem Statement:

You've been asked to assign Pods to nodes using nodeName and nodeSelector fields.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Pod with the field nodeName
2. Creating a Pod with the field nodeSelector
3. Assigning Labels to nodes
4. Creating a Pod with the NotIn operator

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.

The differences between regular and init containers are as follows:

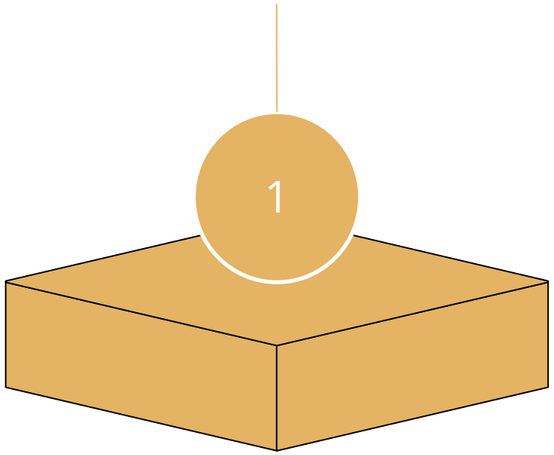
Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings.

Init containers do not support **lifecycle, livenessProbe, readinessProbe, or startupProbe** because they must run to completion before the Pod can be ready.

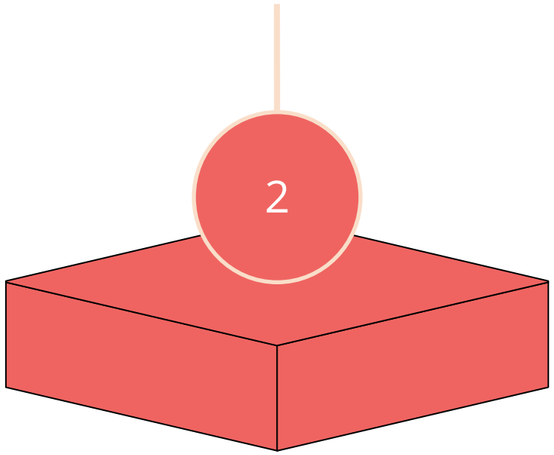
Init Containers

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.



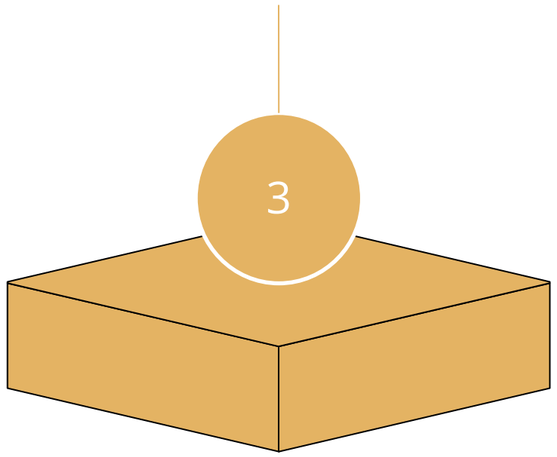
The application image builder and deployer roles can work independently without the need to jointly build a single app image.



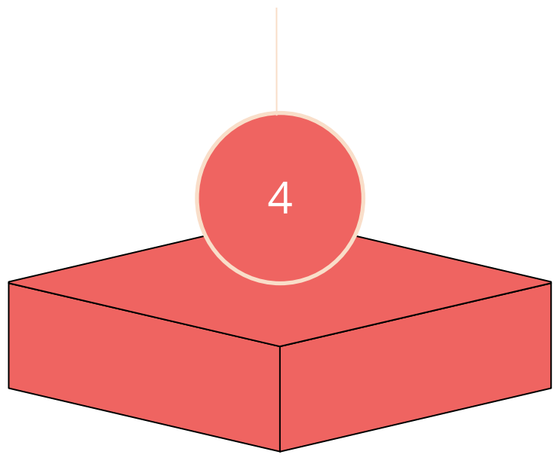
Init Containers

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

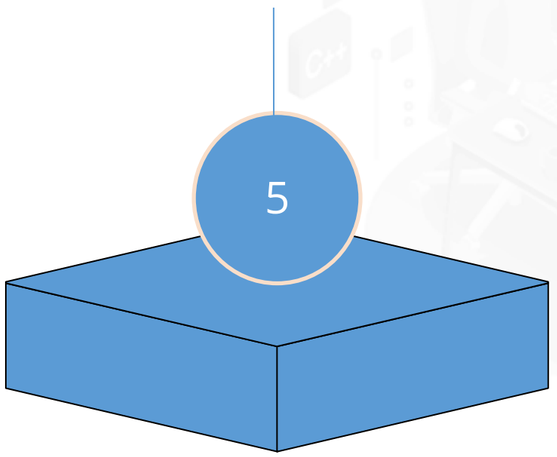
Init containers offer a mechanism to block or delay app container startup until a set of preconditions are met.



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: myapp-container
      image: busybox:1.28
      command: ['sh','-c',"intil nslookup
myservice.$(cat/var/run/secrets/kubernetes.io/seviceaccount/namespace).svc.cluster.l
ocal;do echo waiting for myservice;seep 2;done"]
     - name: myapp-container
      image: busybox:1.28
      command: ['sh','-c',"until nslookup
mydb.4(cat/var/run/secrets/kubernetes.io/serviceaccount/namespace.svc.cluster.local
;do echo waiting for mydb;seep2;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          6m
```

or for more details:

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

Init Containers in Use

Here is a configuration that can be used to make the services appear:

```
  apiVersion: v1
  Kind:  service
  Metadata:
    name:  myservice
   spec:
     ports:
     -  protocol: TCP
        port: 80
        targetPort: 9376
apiVersion: v1
  Kind:  service
  Metadata:
    name:  mydb
   spec:
     ports:
     -  protocol: TCP
        port: 80
        targetPort: 9377
```

Init Containers in Use

To create the **mydb** and **myservice** services, use the given command:

```
kubectl apply -f services.yaml
```

The output is similar to this:

```
service/myservice created
service/mydb created
```

See the init containers complete, and that the myapp-pod pod moves into the Running state:

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

The output is similar to this:

```
NAME        READY    STATUS    RESTARTS   AGE
myapp-pod   1/1      Running   0          9m
```

Using Init Containers

Duration: 15 mins

Problem Statement:

You've been asked to create a Pod using an init Container.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating an Httpd Pod
2. Creating a Service
3. Verifying the Pods state

Using Multi-init Containers

Duration: 15 mins

Problem Statement:

You've been asked to create a Pod using multiple init Containers.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Pod
2. Creating the first Service
3. Creating the second Service
4. Verifying the Pods 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.

1

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

2

Limits can be implemented either reactively or by enforcement.

Resource Types

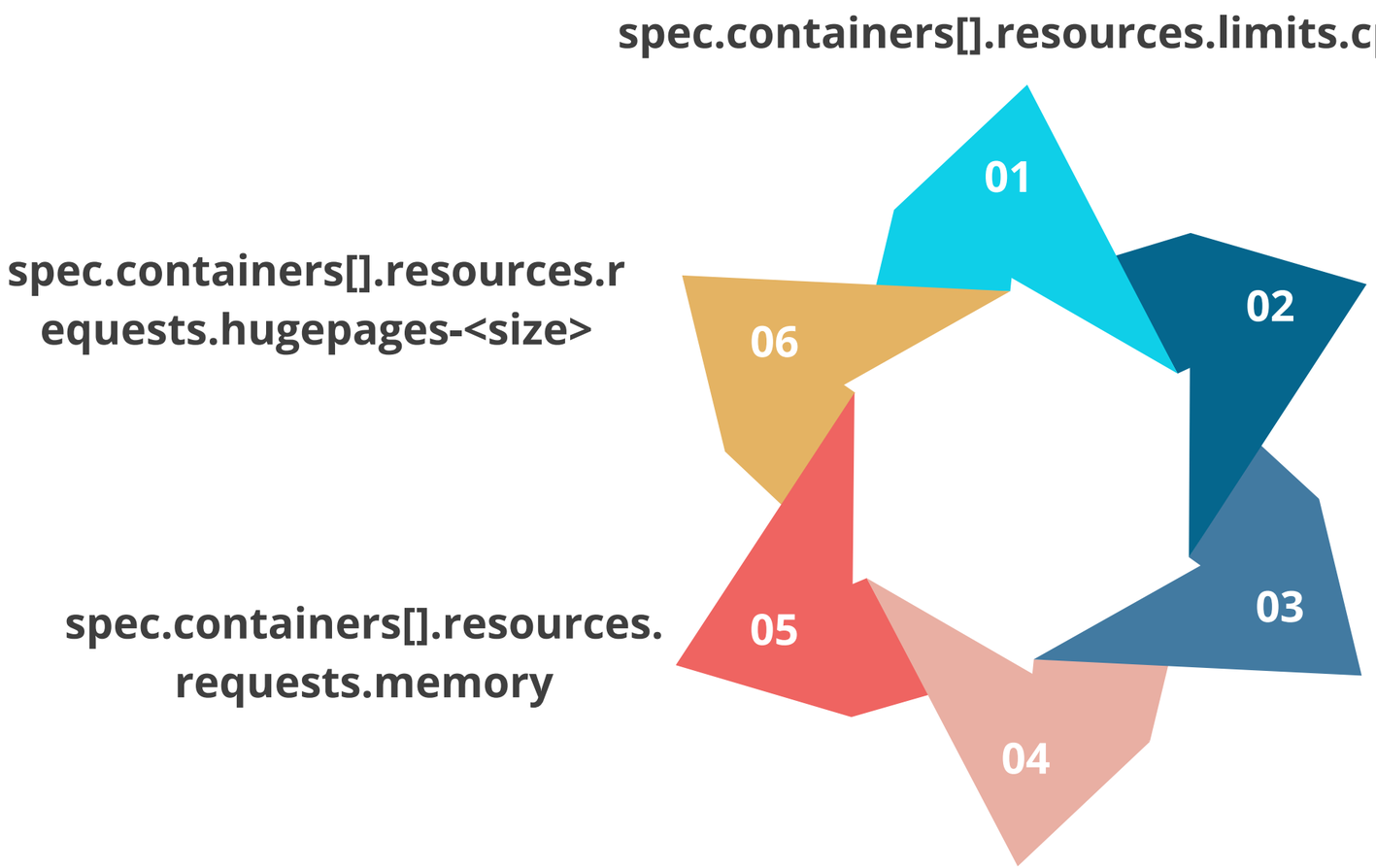
CPU and **memory** are both **resource types**.

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:
E, P, T, G, M, K.

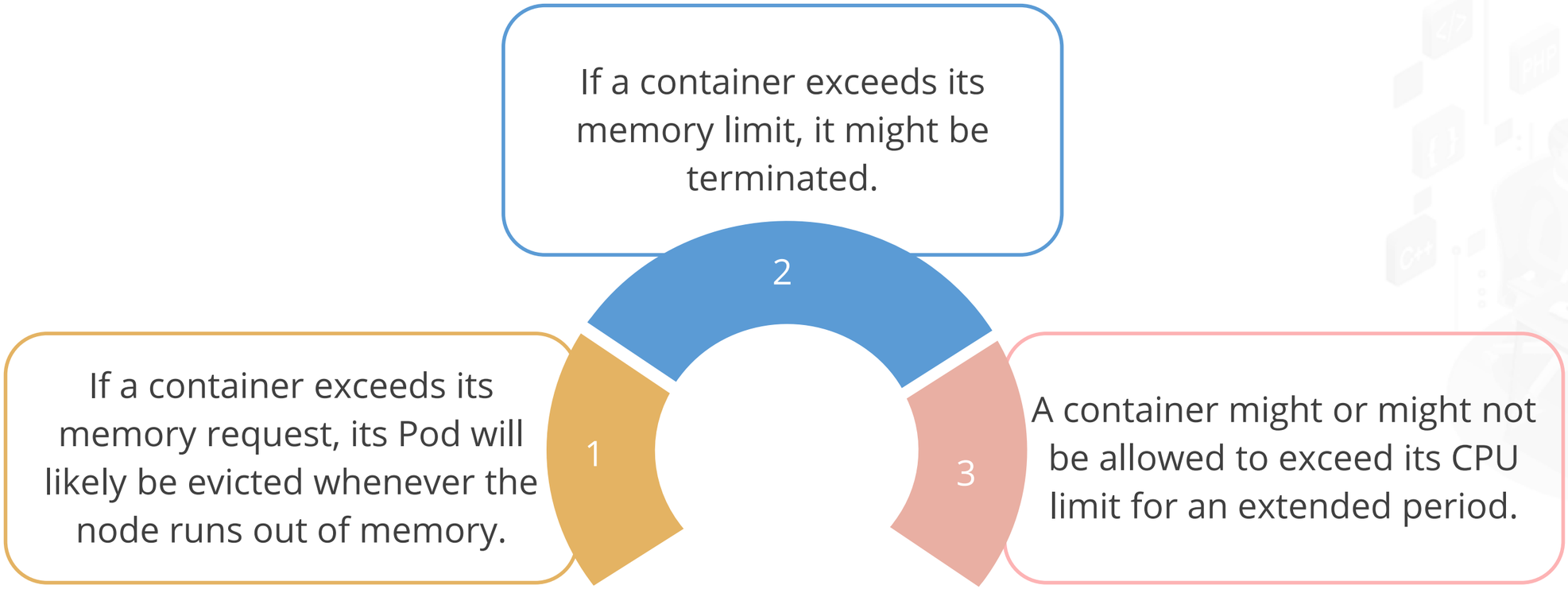
Resource Units in Kubernetes

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  Frontend
 spec:
  containers:
   -  name: app
     image: images.my company.example/app;V4
    resources:
        requests:
          memory: “64M1”
          cpu: ‘250M”
        limits:
          memory: “128M1”
          cpu: “500m”
   -  name: log-aggregator
     image: images.my company.example/log-aggregator;v6
     resources:
        requests:
          memory: “64M1”
          cpu: ‘250M”
        limits:
          memory: “128M1”
          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”
```

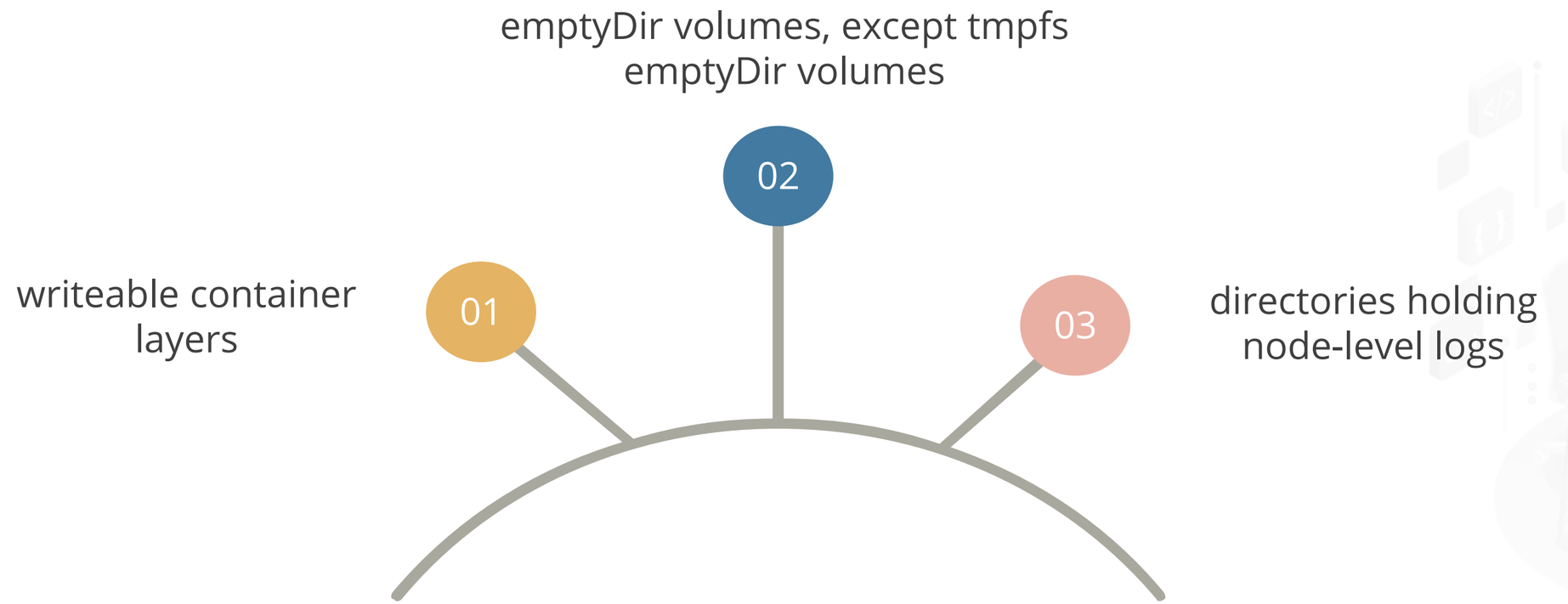
Local Ephemeral Storage

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

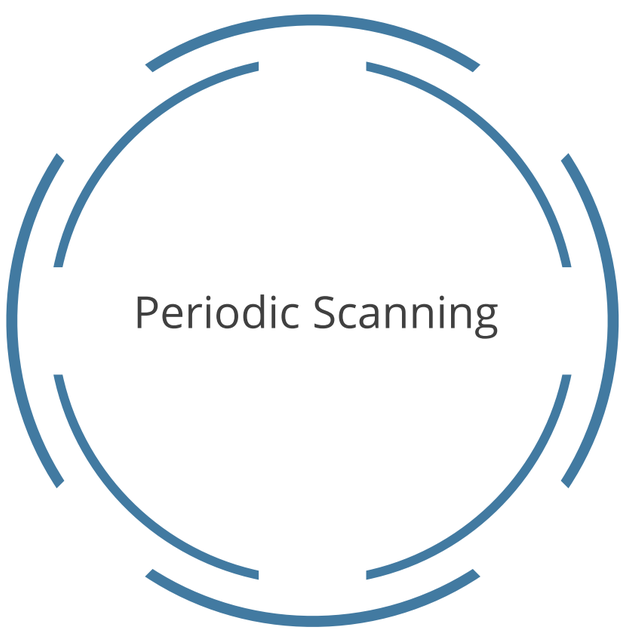
If the kubelet is managing local ephemeral storage as a resource, it measures the use of storage in:



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:



Container Resources

Duration: 10 mins

Problem Statement:

You've been asked to configure the memory requests and limits for the Containers.

ASSISTED PRACTICE

Assisted Practice: Guidelines

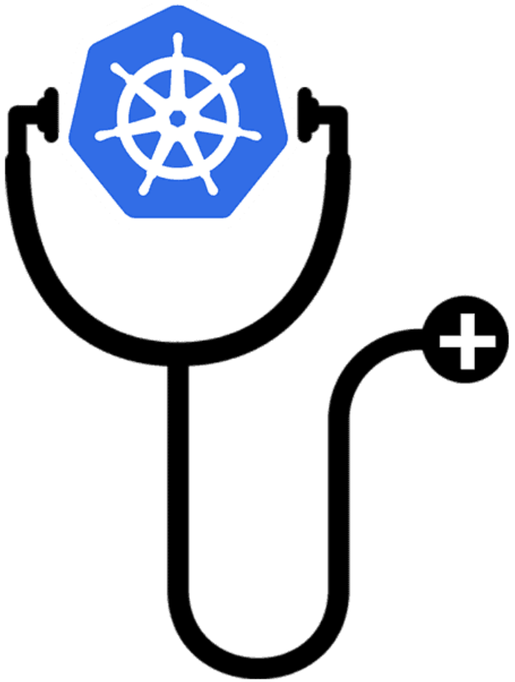
Steps to be followed:

1. Creating a ResourceQuota
2. Defining the LimitRange
3. Verifying the ResourceQuota

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**.



Enabling Node Problem Detector

Kubectl provides the most flexible management of Node Problem Detector. For example:

```
apiVersion: apps/v1
  Kind:  DaemonSet
  Metadata:
    name:  node-problem-detector-v0.1
    namespace: kube-system
    labels:
      k8s-app:  nude-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: “true”
   spec:
    selector:
     matchlabels:
      k8s-app:  nude-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: “true
    template:
     metadata:
      labels:
       k8s-app:  nude-problem-detector
       version: v0.1
       kubernetes.io/cluster-service: “true”
     spec:
      hostnetwork: true
      containers:
```

```
- name: node-problem-detector
        image: k8s.gcr.io/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: “ 200m”
            memory:  “100M1”
          requests:
            cpu:  “20m”
            memory: “ 20M1”
        volumeMounts:
          - name: log
            mountpath: /log
            readOnly:  true
        volumes:
          - name: log
            hostpath:
              path:  /var/log/
```

Overwriting the Configuration

The default configuration is embedded when building the Docker image of a Node Problem Detector. Use a **ConfigMap** to overwrite the configuration:

Change the configuration files in **config/**

Create the ConfigMap node-problem-detector config:
kubectl create configmap node-problem-detector-config --from-file=config/

-

Overwriting the Configuration

To change the **node-problem-detector.yaml**, use the **ConfigMap**:

```
apiVersion: apps/v1
   Kind:  DaemonSet
   Metadata:
     name:  node-problem-detector-v0.1
     namespace: kube-system
     labels:
       k8s-app:  nude-problem-detector
       version: v0.1
       kubernetes.io/cluster-service: “true”
    spec:
     selector:
      matchlabels:
       k8s-app:  nude-problem-detector
       version: v0.1
       kubernetes.io/cluster-service: “true
     template:
      metadata:
       labels:
       k8s-app:  nude-problem-detector
       version: v0.1
       kubernetes.io/cluster-service: “true”
      spec:
       hostnetwork: true
       containers:
        - name: node-problem-detector
```

```
image: k8s.gcr.io/node-problem-detector:v0.1
securityContext:
privileged: true
          resources:
            limits:
              cpu: “ 200m”
              memory:  “100M1”
            requests:
              cpu:  “20m”
              memory: “ 20M1”
          volumeMounts:
            - name: log
              mountpath: /log
              readOnly:  true
            - name: config # Overwrite the config/directory with ConfigMap
volume
              mountpath: /config
              readOnly:  true
          volumes:
            - name: log
              hostpath:
                path:  /var/log/
            - name: config # Define ConfigMap volume
              configMap:
                name: node-problem-detector-config
```

Overwriting the Configuration

Recreate the Node Problem Detector with a new configuration file:

```
# If you have a node-problem-detector running, delete before recreating
kubectl delete –f hhtps://k8s.io/examples/debug/node-problem-detector,yanl
kubectl apply –f hhtps://k8s.io/examples/debug/node-problem-detector-configmap,yanl
```

If a Node Problem Detector is running as a cluster Addon, overwriting a configuration is not supported.

Kernel Monitor

Kernel Monitor is a system log monitor daemon supported in the Node Problem Detector.

To support a new **NodeCondition**, create a condition definition within the **conditions** field in **config/kernel-monitor.json**, such as:

```
{
  “type”: “NodeConditionType”
  “reason”: “CameCaseDefaultNodeConditionReason”
  “message”: “arbitrary Default node condition message”
}
```

Overwriting the Configuration

To detect new problems, extend the **rules** field in **config/kernel-monitor.json** with a new rule definition:

```
{
  “type”: “temporary/permanent”
  “condition”: “NodeConditionOfPermanentIssue”
  “reason”: “CameCaseShortReason”
  “message”: “regexp matching the issue in the kernel log”
}
```

The kernel monitor uses the **translator** plugin to translate the internal data structure of the kernel log. Users can implement a new translator for a new log format.

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.

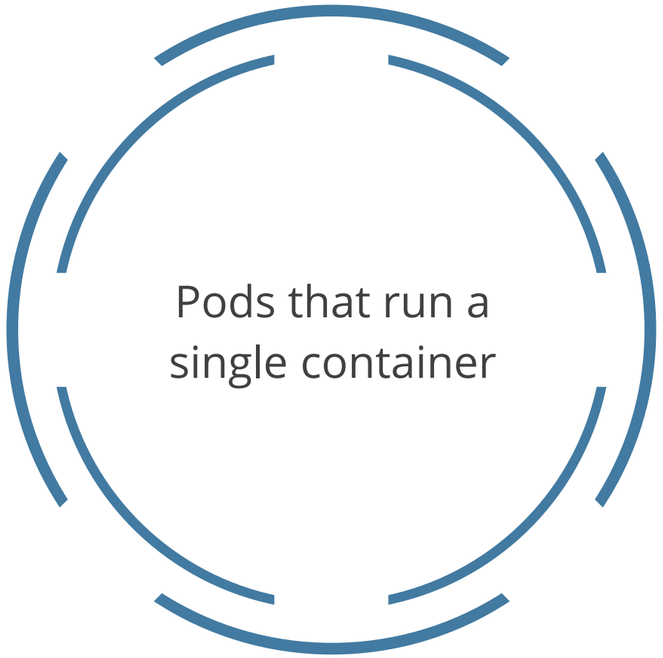
02 The kernel log grows relatively slowly.

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

Multi-Container Pods

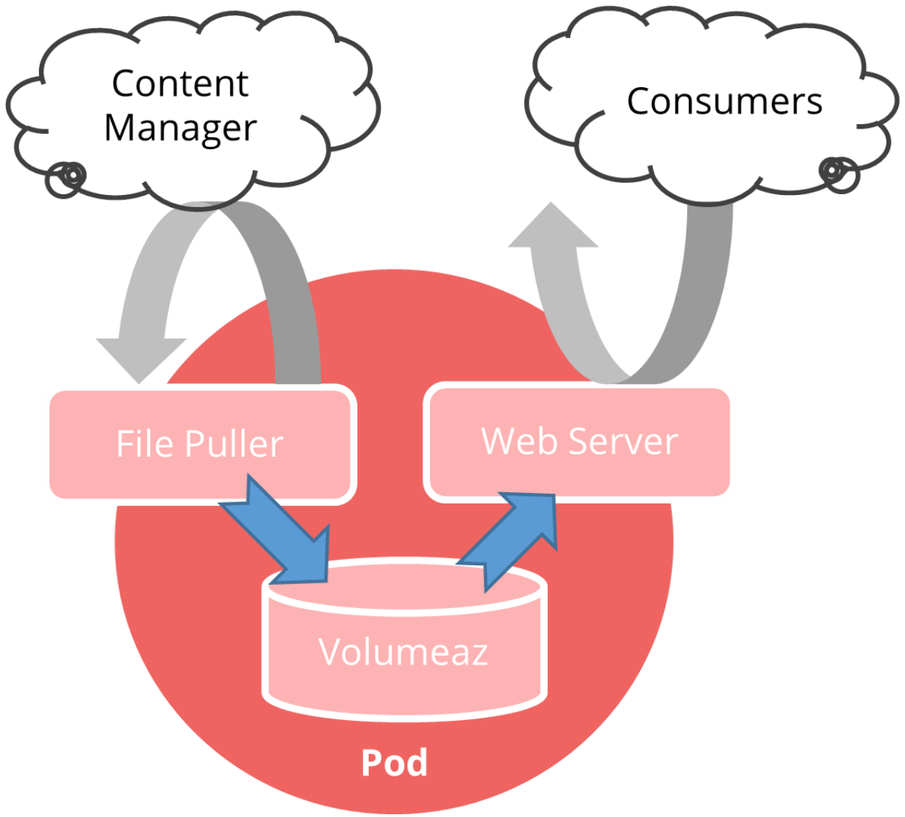
Using Pods in Different Situations

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



Using Pods in Different Situations

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

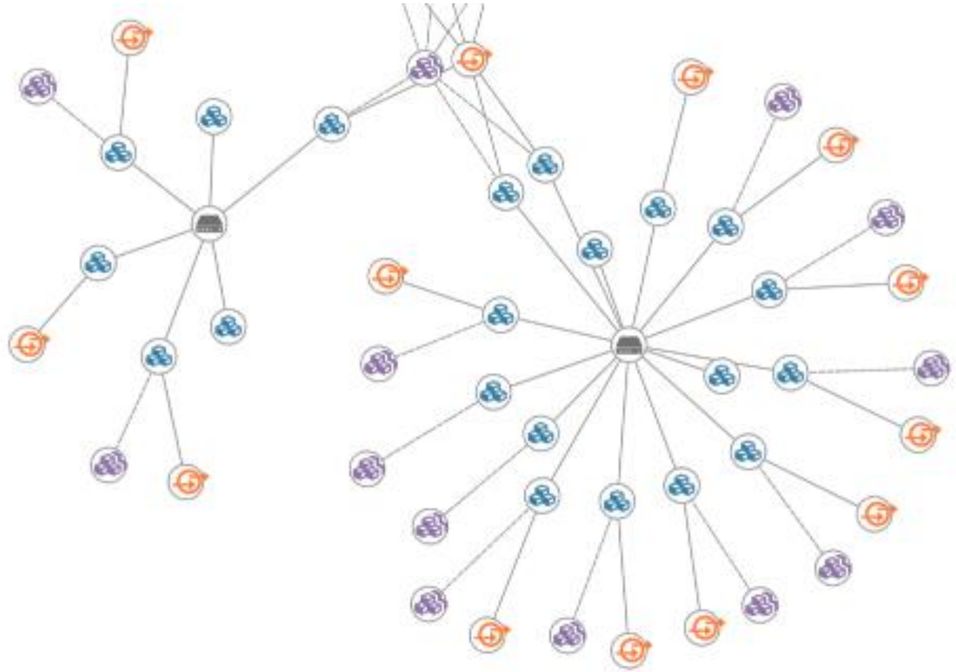


There can be a container that acts as a web server for files in a shared volume and a separate **sidecar** container that updates those files from a remote source.

Pod Topology Spread Constraints

Introduction

Topology spread constraints are used to control the Pods that are spread across the cluster, among failure domains such as regions, zones, nodes, and other user-defined topology domains.



It helps achieve high availability as well as efficient resource utilization.

Prerequisites

Topology spread constraints rely on node labels to identify the topology domain(s) that each Node is in. For example, a Node might have labels:

```
node=node1, zone=us-east-1a, region=us-east-1
```

Consider a four-node cluster with the following labels:

Name	Status	Roles	Age	Versions	Labels
Node1	Ready	<none>	4m26s	v1.16.0	node=node1,zone=zoneA
Node2	Ready	<none>	3m58s	v1.16.0	node=node2,zone=zoneA
Node3	Ready	<none>	3m17s	v1.16.0	node=node3,zone=zoneB
Node4	Ready	<none>	2m43s	v1.16.0	node=node4,zone=zoneB

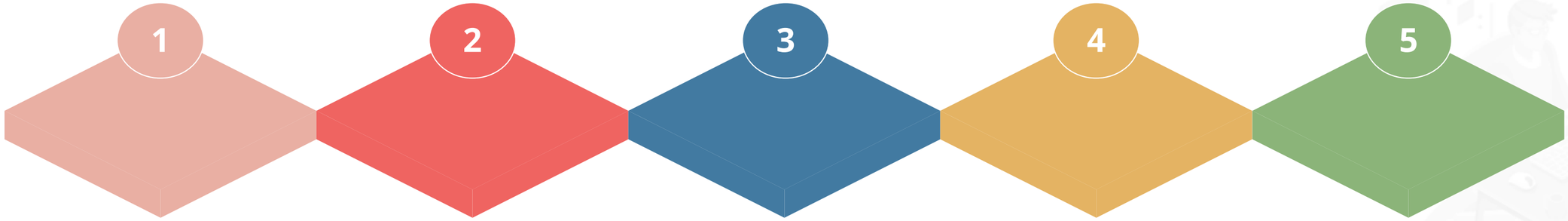
Spread Constraints for Pods

The API field **pod.spec.topologySpreadConstraints** is defined as below:

```
apiVersion: v1
Kind:  pod
Metadata:
  name:  Mypod
spec:
  topologySpreadConstraints:
    maxSkew:   <integer>
        Topology: <string>
        whenUnsatisfiable: <String>
        labelSelector: <Object>
```

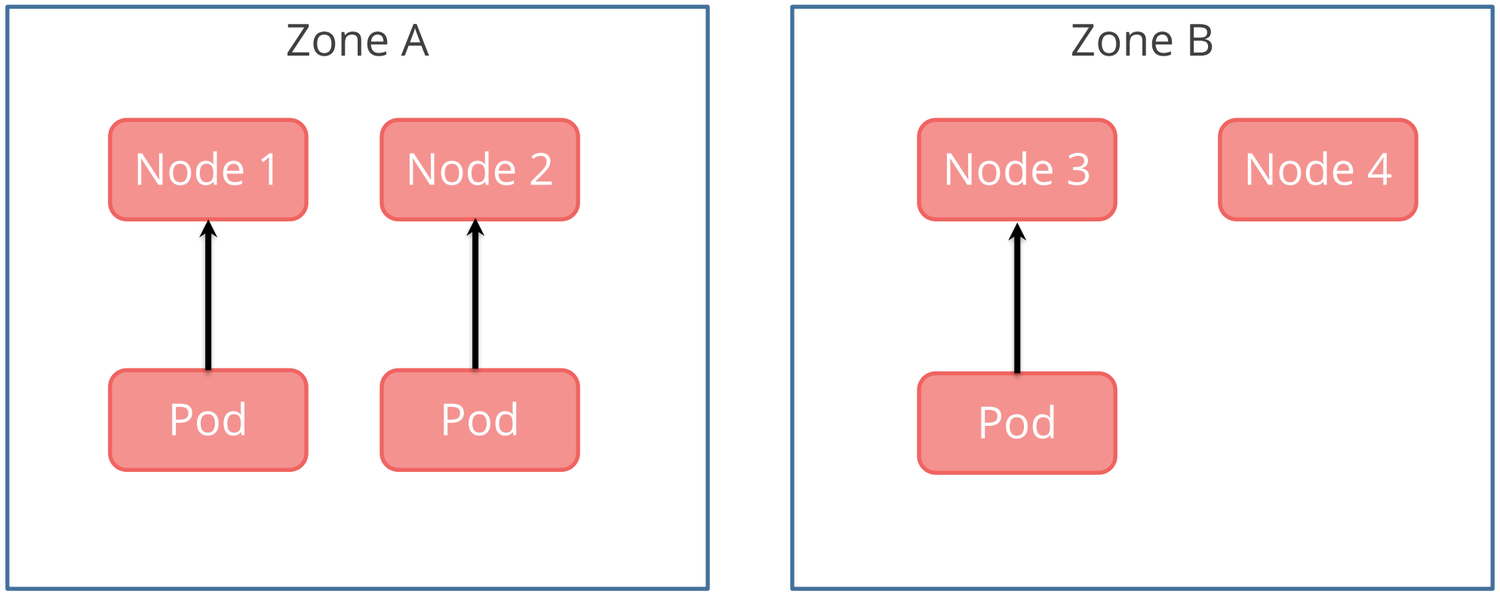
Spread Constraints for Pods

One or multiple **topologySpreadConstraint** can be defined to instruct the Kube-scheduler on how to place each incoming Pod to the existing Pods across the cluster. The fields are:



Example: One TopologySpreadConstraint

Consider a four-node cluster where three Pods labeled **foo:bar** are located in node1, node2, and node3 respectively:



Example: One TopologySpreadConstraint

For an incoming Pod to be evenly spread with existing Pods across zones, the spec can be given as:

```
Kind:  Pod
apiVersion: v1
   Metadata:
     name:  Mypod
            labels:
            foo: bar
   spec:
     topologySpreadConstraints:
       maxSkew:   1
            Topology: zone
            whenUnsatisfiable: DoNotSchedule
            labelSelector:
            matchLabels:
            foo:
            container:
            name: pause
            image: K8s.gcr.io/pause:3.1
```

Example: One TopologySpreadConstraint

The Pod spec can be created to meet various kinds of requirements:

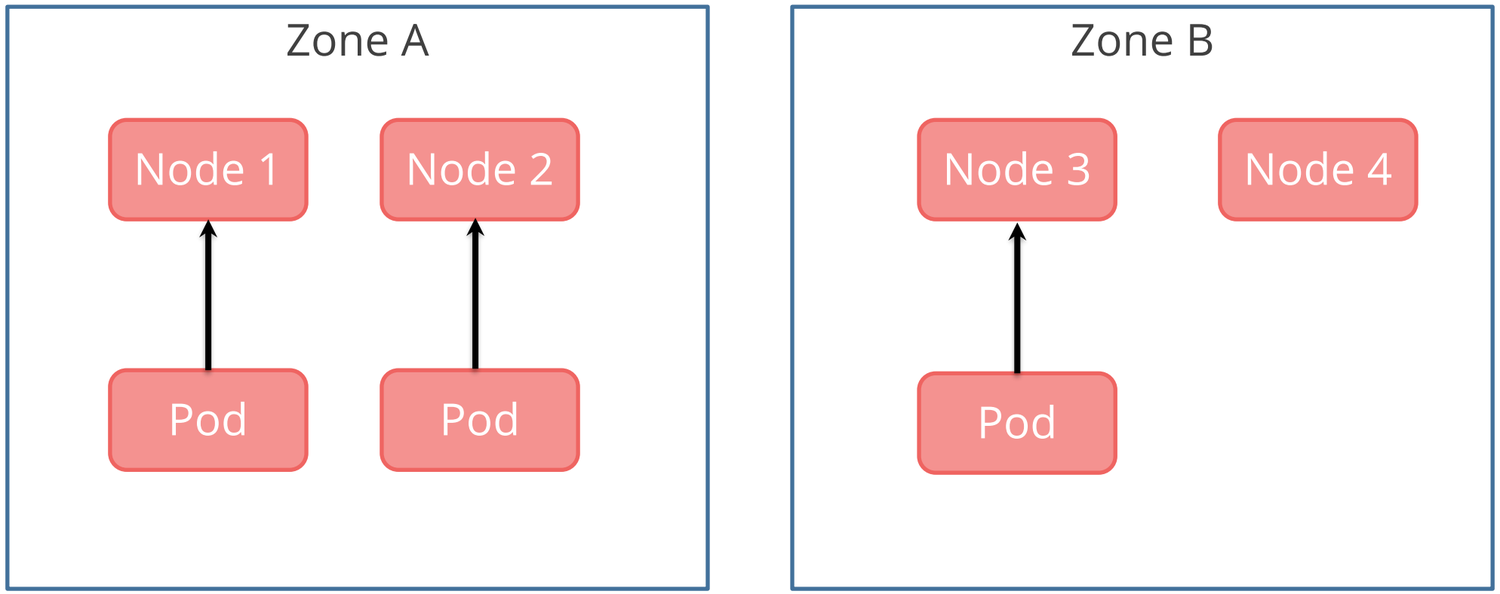
Change **maxSkew** to a bigger value like **2** so that the incoming Pod can be placed on **zoneA** as well.

Change **topologyKey** to the **node** to distribute the Pods evenly across nodes instead of zones.

Change **whenUnsatisfiable: DoNotSchedule** to **whenUnsatisfiable: ScheduleAnyway** to ensure that the incoming Pod is always schedulable.

Multiple TopologySpreadConstraints

This builds upon the previous example. Consider a four-node cluster where three Pods labeled **foo:bar** are located in node1, node2, and node3 respectively:



Multiple TopologySpreadConstraints

Use two **TopologySpreadConstraints** to control the Pods spreading on both zone and node:

```
Kind:  pod
apiVersion: v1
   Metadata:
     name:  Mypod
              labels:
              foo: bar
   spec:
     topologySpreadConstraints:
       maxSkew:   1
              Topology: zone
              whenUnsatisfiable: DoNotSchedule
              labelSelector:
              matchLabels:
              foo: bar
               maxSkew: node
              whenUnsatisfiable: DoNotSchedule
              labelSelector:
              matchLabels:
              foo: bar
              container:
              name: pause
              image: K8s.gcr.io/pause:3.1
```

Conventions

There are some implicit conventions worth noting here:

Only the Pods holding the same namespace as the incoming Pod can be matching candidates.

Nodes without **topologySpreadConstraints[*].topologyKey** present will be bypassed.

Be aware of what will happen if the incomingPod's **topologySpreadConstraints[*].labelSelector** does not match its labels.

If the incoming Pod has **spec.nodeSelector** or **spec.affinity.nodeAffinity** defined, nodes not matching them will be bypassed.

Cluster-level Default Constraints

It is possible to set default Topology Spread Constraints for a cluster. Default Topology Spread Constraints are applied to a Pod only if:

1. It doesn't define any constraint in its **.spec.topologySpreadConstraints**.

2. It belongs to a service, replication controller, replica set, or stateful set.

Cluster-level Default Constraints

The following is an example of a configuration:

```
apiVersion: kubescheduler.config.k8s.io/v1betal
Kind:  kubeschedulerconfiguration
Profile:
schedulerName: default-scheduler
Pluginconfig:
      name: podTopologySpread
            args:
            defaultConstraints:
        maxSkew:   1
            TopologyKey: topology.kubernetes.io/zone
            whenUnsatisfiable: ScheduleAnyway
            defaultingType: List
```

Internal Default Constraints

With the **DefaultpodTopologySpread** feature gate enabled by default, the legacy **SelectorSpread** plugin is disabled. Kube-scheduler uses the following default Topology Constraints for the **podTopologySpread** plugin configuration:

```
defaultConstraints:
        maxSkew:   3
              TopologyKey: “kubernetes.io/hostname”
              whenUnsatisfiable: ScheduleAnyway
              maxSkew:   5
              TopologyKey: “kubernetes.io/zone”
              whenUnsatisfiable: ScheduleAnyway
```

Comparison with PodAffinity and PodAntiAffinity

In Kubernetes, directives related to **Affinity** control how Pods are scheduled, more packed, or more scattered.

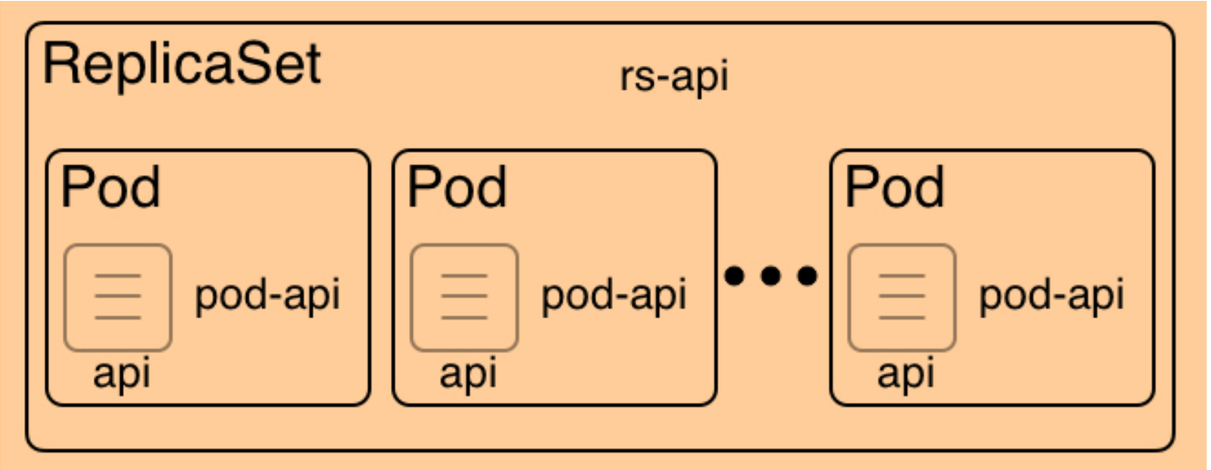
1. For **PodAffinity**, try to pack any number of Pods into qualifying topology domain(s).

2. For **PodAntiAffinity**, only one Pod can be scheduled into a single Topology domain.

ReplicaSet

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.

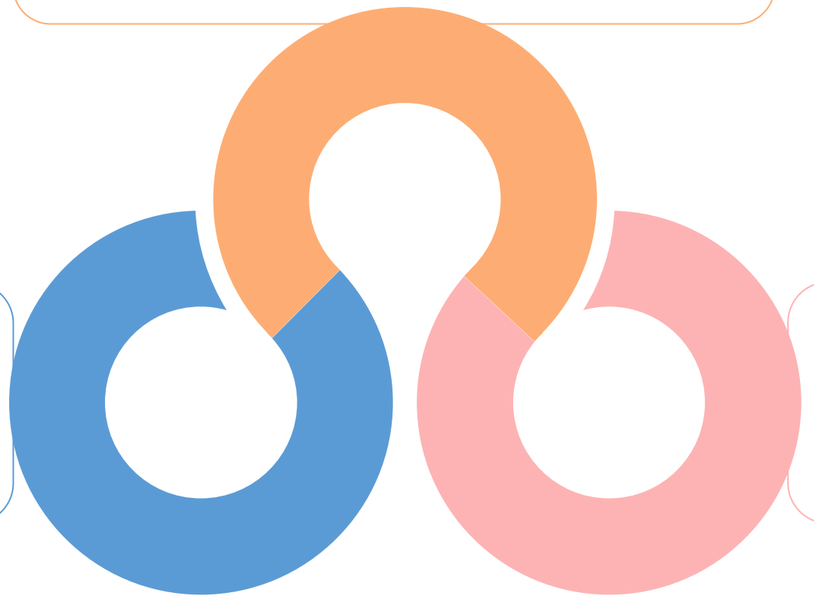
A ReplicaSet identifies new Pods to acquire by using its selector.

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 ReplicaSet

Example:

```
apiVersion: app/v1
Kind: ReplicaSet
Metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
Spec:
 #modify replicas according to your case
 replicas: 3
 selector:
  matchLabers:
    tier: frontend
 template:
  metadata:
   labels:
    tier: frontend
Spec:
 container:
  name: php-redis
  image: grc.io/google_samples/gb-frontened:v3
```

Using a ReplicaSet

To create the defined ReplicaSet and the Pods that it manages, save this manifest into the **frontend.yaml** and submit it to a Kubernetes cluster:

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml

get the current ReplicaSets deployed:
kubectl get rs

And see the frontend that is created:

NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       6s

Check on the state of the ReplicaSet:

kubectl describe rs/frontend
```

When to Use a ReplicaSet

The output will be similar to:

```
Name: frontend
Namespace: default
Selector: tier=frontend
Labels: app=guestbook
      tier=frontend
Annotations: kubectl.kubernetes.io/last-applied-
configuration:{"apiversion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations":{},
"labels":{"app":"guestbook","tier":"frontend"},"name":"frontend",...
Replicas: 3 current/ 3 defired
pods Status: 3 running/0 waiting/ 0 succeeded / 0 failed
pod template:
  labels: tier=frontend
  containers:
  php-redis:
  image: gcr.io/google_sample/gb-frontend:v3
  port: <none>
  Host Port: <none>
  Environment: <none>
  Mounts: <none>
  volumes: <none>
Events:
```

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 verify that the owner reference of these Pods is set to the frontend ReplicaSet, get the yaml of one of the Pods running:

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.

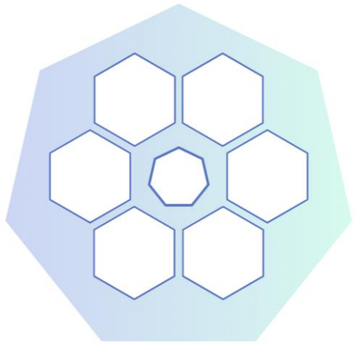
Non-Template Pod Acquisitions

To create bare Pods with no problems, it is strongly recommended that bare Pods do not have labels that match the selector of one of the ReplicaSets.

The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template. It can acquire other Pods in the manner specified in the previous sections.

Non-Template Pod Acquisitions

As the Pods do not have a Controller (or any object) as their owner reference and match the selector of the frontend ReplicaSet, they will be immediately acquired by it.



Suppose the Pods are created after the frontend ReplicaSet has been deployed and has set up its initial Pod replicas to fulfill its replica count requirement:

kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml

The new Pods will be acquired by the ReplicaSet and immediately terminated as the ReplicaSet would be over its desired count.

Non-Template Pod Acquisitions

For fetching the Pods, use the given command:

kubectl get Pods

The output shows that the new Pods are either already terminated or are in the process of being terminated:

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	10m
frontend-vcmts	1/1	Running	0	10m
frontend-wtsmm	1/1	Running	0	10m
Pod1	0/1	Terminating	0	1S
Pod2	0/1	Terminating	0	1S

Non-Template Pod Acquisitions

To create the Pods, run the following command:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

After creating the Pods, create the ReplicaSet:

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

Check that the ReplicaSet has acquired the Pods and has only created new ones according to its spec, until the number of its new Pods and the original Pods match its desired count. For fetching the Pods:

```
kubectl get pods
```

Non-Template Pod Acquisitions

After running the **kubectl get pods** command, the output will be shown as:

Name	Ready	Status	Restarts	Age
frontend-hmmj2	1/1	Running	0	9s
pod1	1/1	Running	0	36S
pod2	1/1	Running	0	36S

Writing a ReplicaSet Manifest

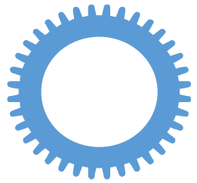
As with all other Kubernetes API objects, a ReplicaSet needs the **apiVersion**, **kind**, and **metadata** fields. For ReplicaSets, the **kind** is always a ReplicaSet.

The name of a ReplicaSet object must be a valid **DNS subdomain name**.

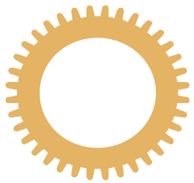
A ReplicaSet also needs a **.spec** section.

Pod Template

PodTemplates 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 Selector

The label selector 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 ReplicaSets

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

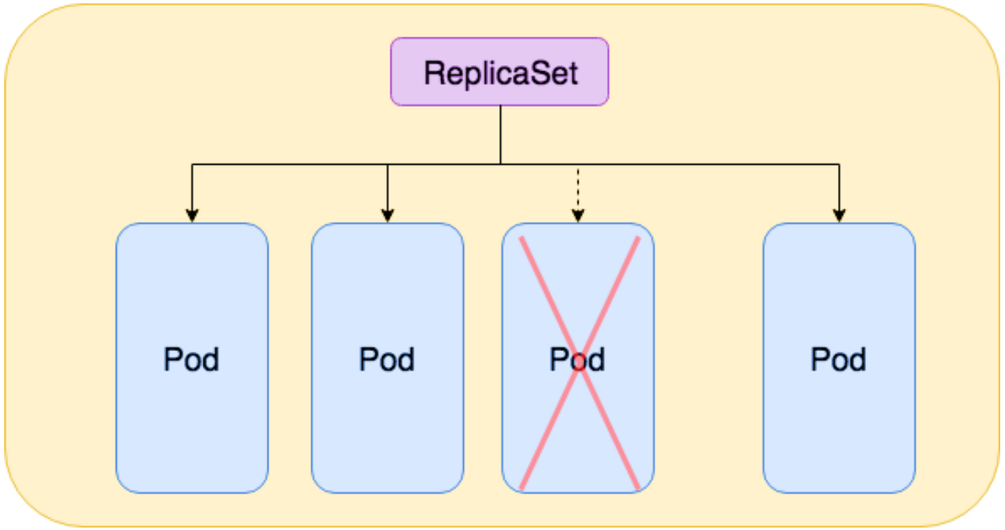
When using the REST API or the **client-go** library, ensure that **propagationPolicy** is set to **Background** or **Foreground** in the -d option.

Working with ReplicaSets

Example:

```
Kubectl proxy –port=800
Curl –x DELETE 'localhost:8080/apis/apps/v1/namespaces/default/replicasets/frontend'\
>-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationpolicy":"Foreground"}'\
> -H "content-type: application/json
```

Working with ReplicaSets



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.

Working with ReplicaSets

A ReplicaSet can be easily scaled up or down by simply updating the **.spec.replicas** field. When scaling down, the ReplicaSet controller prioritizes scaling down Pods by sorting the available Pods based on the following general algorithm:

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.

ReplicaSet as a Horizontal Pod Autoscaler Target

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

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

ReplicaSet as a Horizontal Pod Autoscaler Target

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

```
kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml
```

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

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

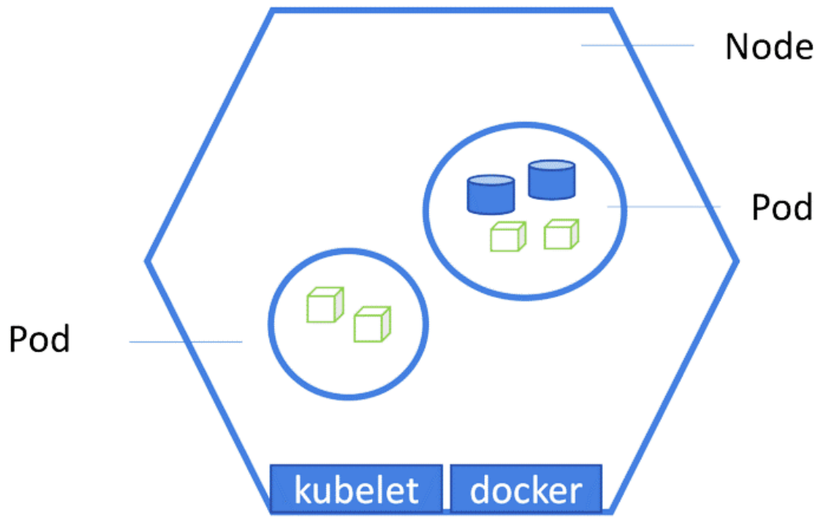
Steps to be followed:

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

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.

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:

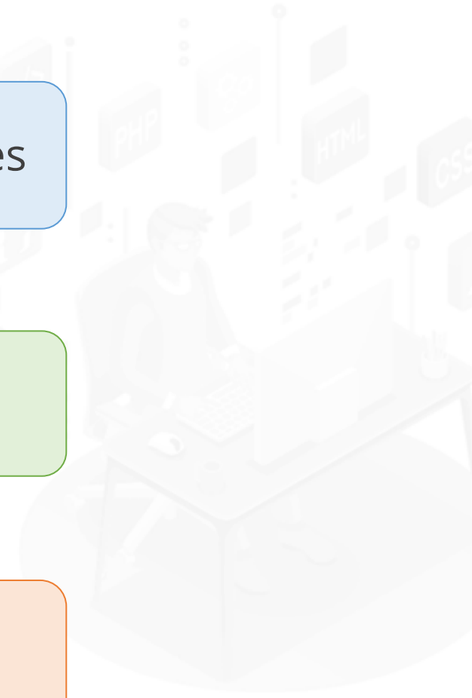
1. Create a static Pod in the worker node

Application Configuration

Overview

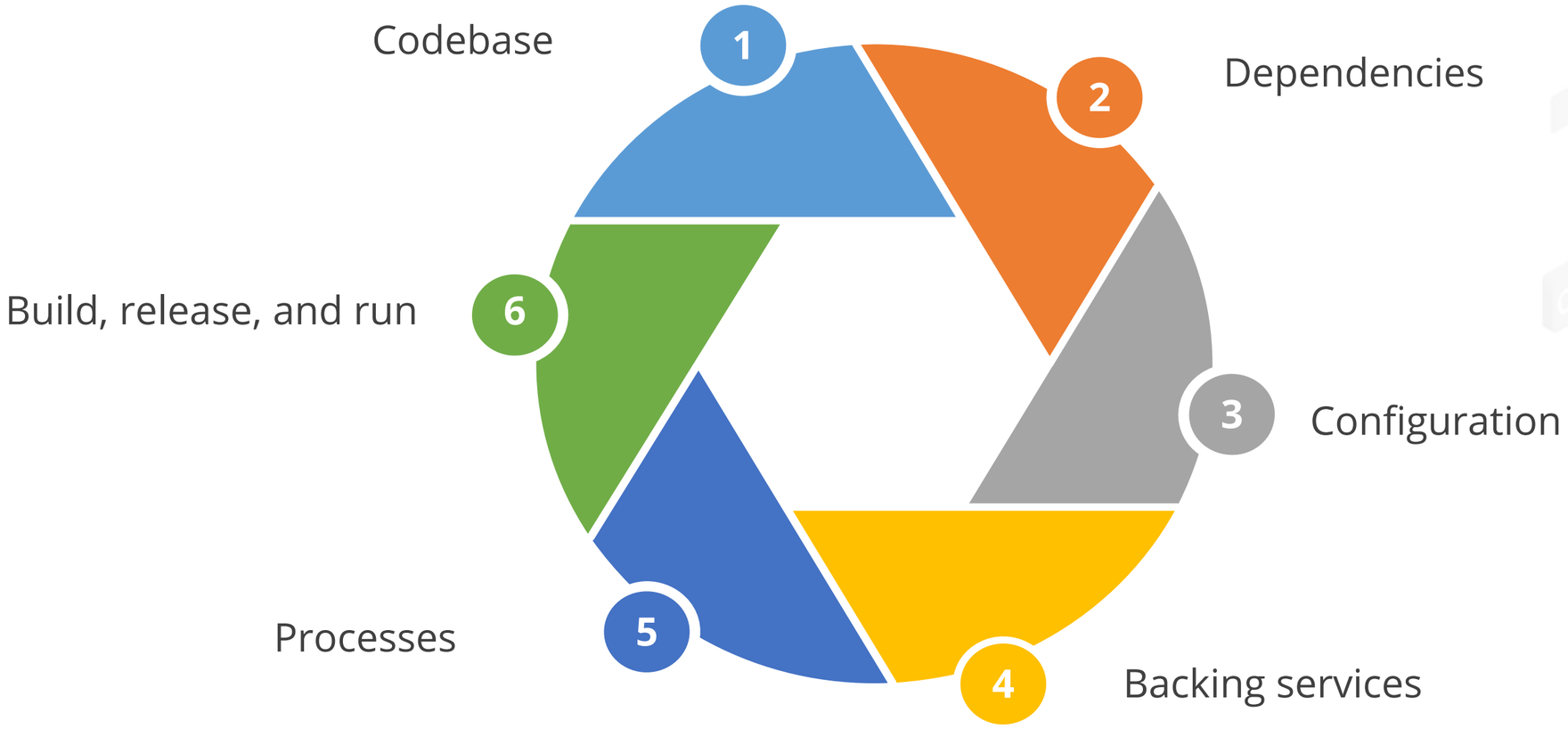
An application’s configuration is everything that is likely to vary between deploys (staging, production, developer environments, and so on). This includes:

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



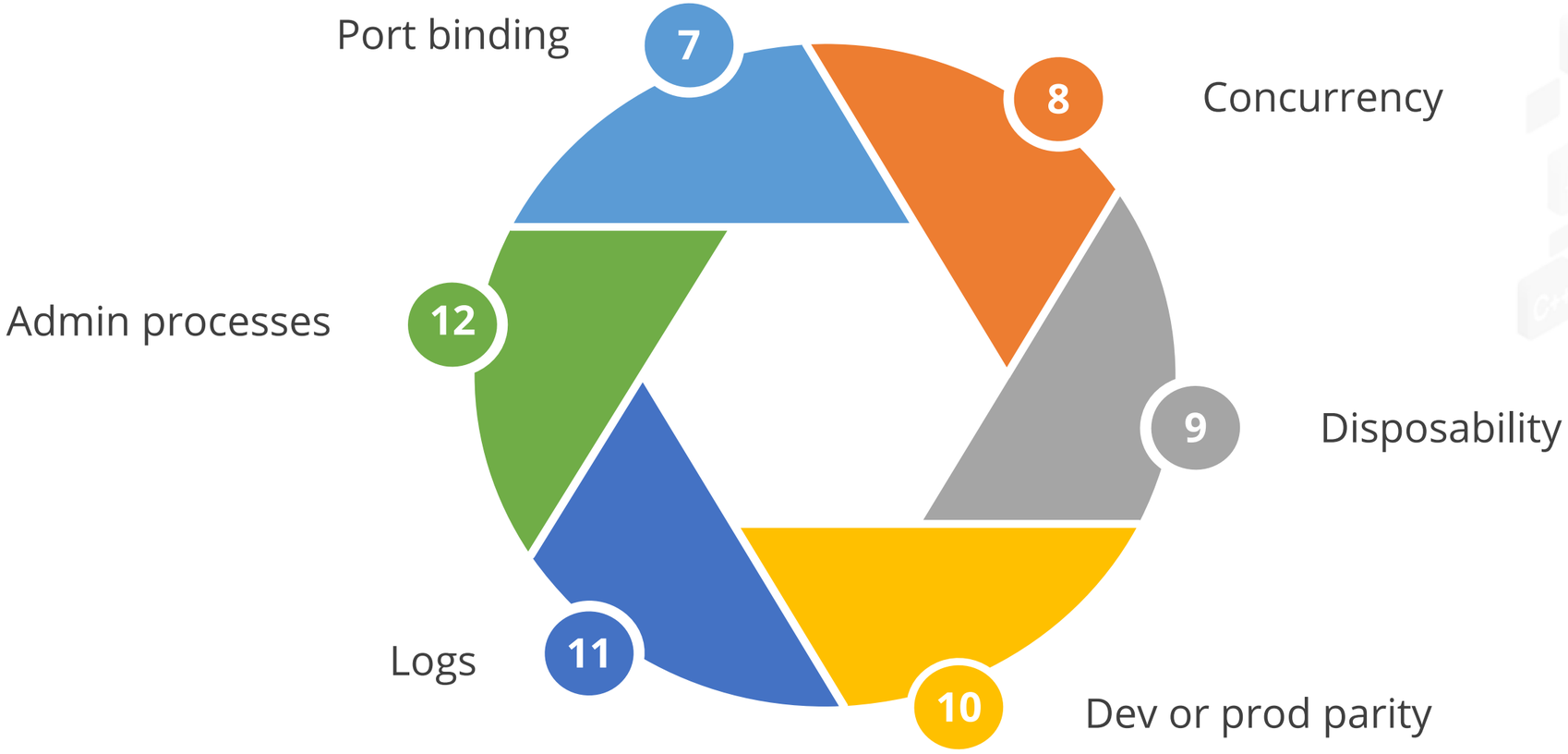
Twelve-Factor Principles

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):

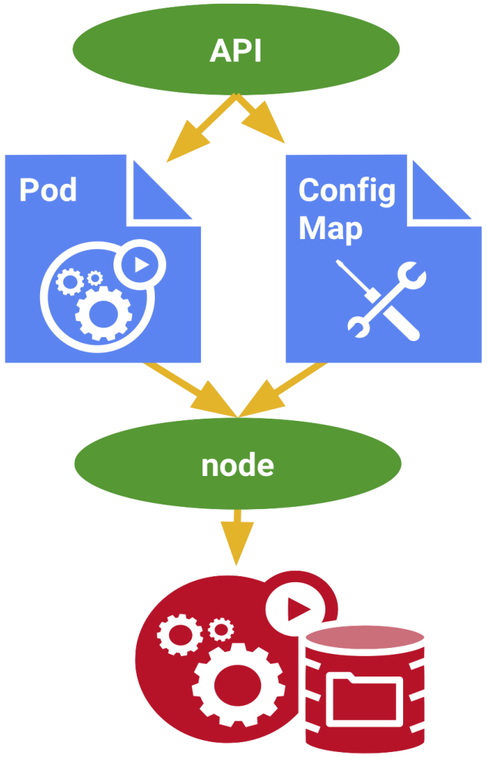
```
$ kubectl exec pod1 -it -- env | grep
ENVVAR
ENVVAR1=value1
ENVVAR2=value2
```

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
```

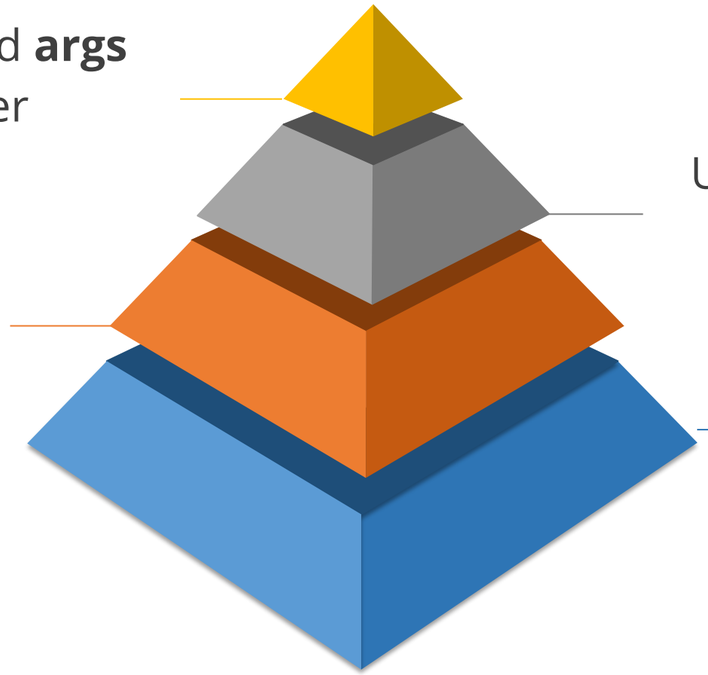
- The users can create a Pod specification that refers to a ConfigMap.
- The spec can configure the container(s) in that Pod based on the ConfigMap data.
- The Pod and the ConfigMap must be in the same namespace.
- The example shows a ConfigMap that has some keys with single values and other keys where 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**.

Example

```
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 help store and manage sensitive information such as passwords, OAuth tokens, and SSH keys. To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

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

When creating a Secret, specify its type using the type field of the Secret resource, or certain equivalent kubectl command-line flags. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them:

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

Mount Secrets as data volumes or expose them as environment variables that can be used by a container in a Pod. Other parts of the system can also use Secrets, without being directly exposed to the Pod. To access data from a Secret in 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 image or command line so that the program looks for values in the specified environment variables

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**

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:

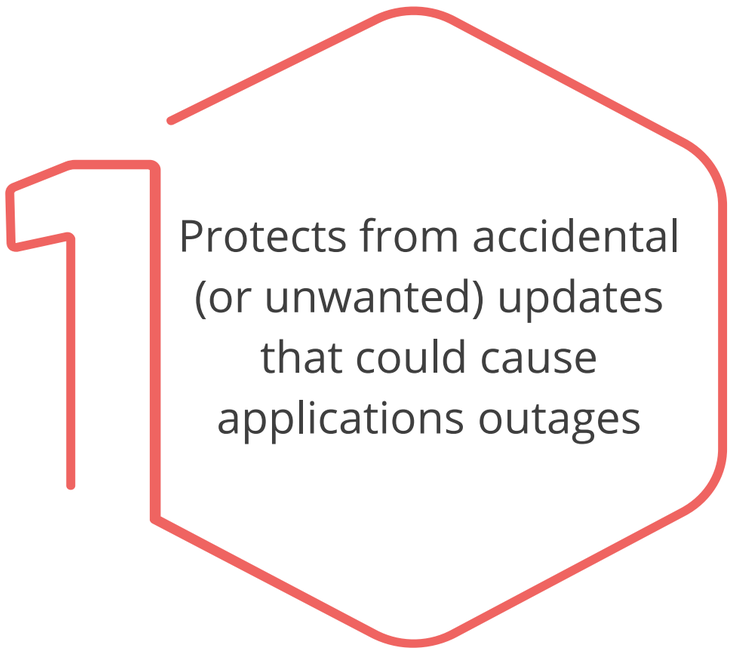
Admin

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:



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

Achieving Scalability

Scaling Applications

Scalability is one of the core benefits of Kubernetes. The workloads can be automatically scaled up or down, depending on historical resource utilization with autoscaling. Autoscaling in Kubernetes has 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.

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.

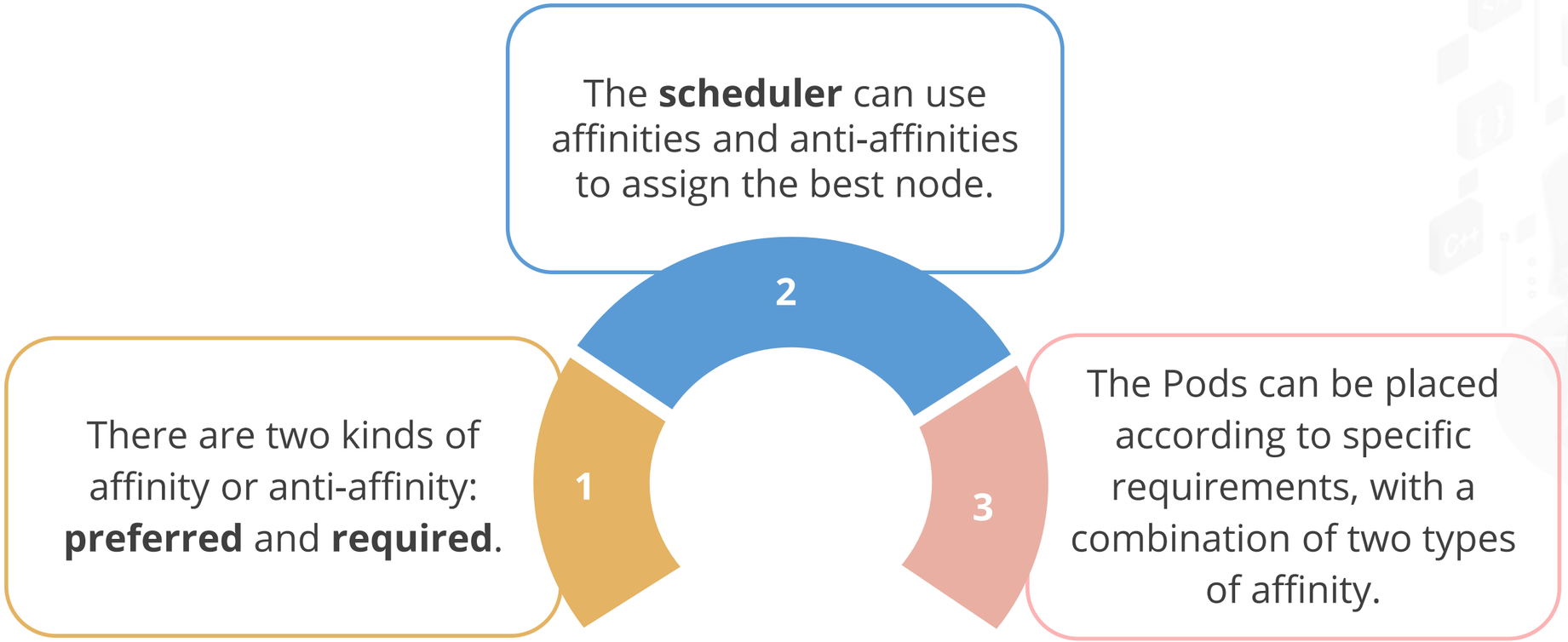
It allows **autoscalers** to estimate the capacity and ensure that the cluster expands (and contracts) as the demand changes.

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.



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

Deployment Strategies

Deployment strategies dictate how the replacement of running Pods is done on Kubernetes when configuration updates are happening. There are two kinds of strategies:

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.

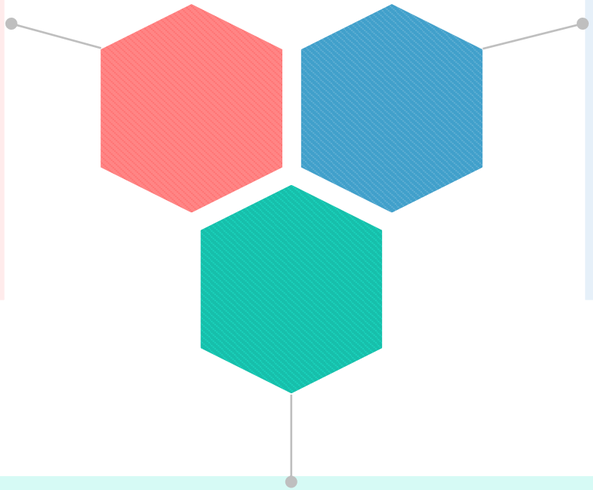
Pod Disruption Budgets

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

Reasons for disruptions:

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

.

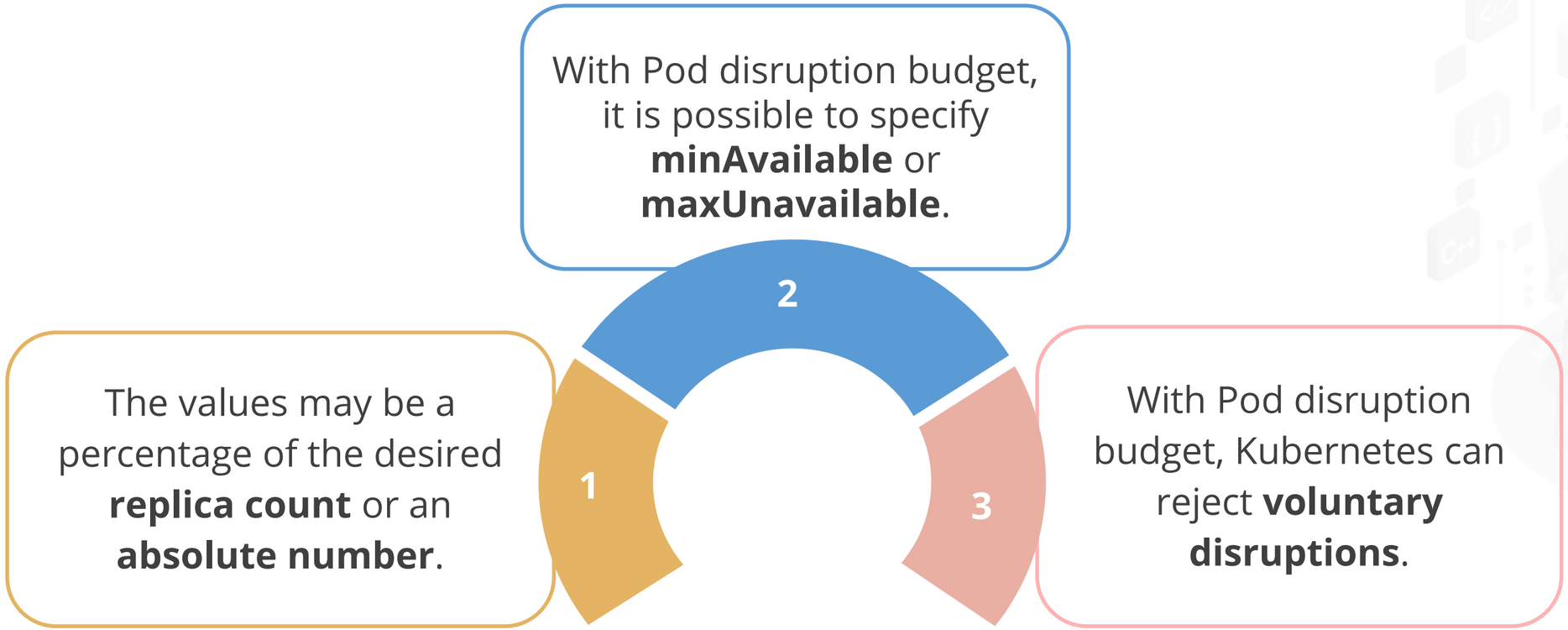


An autoscaler removes any underutilized node.

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.



Horizontal Pod Autoscalers

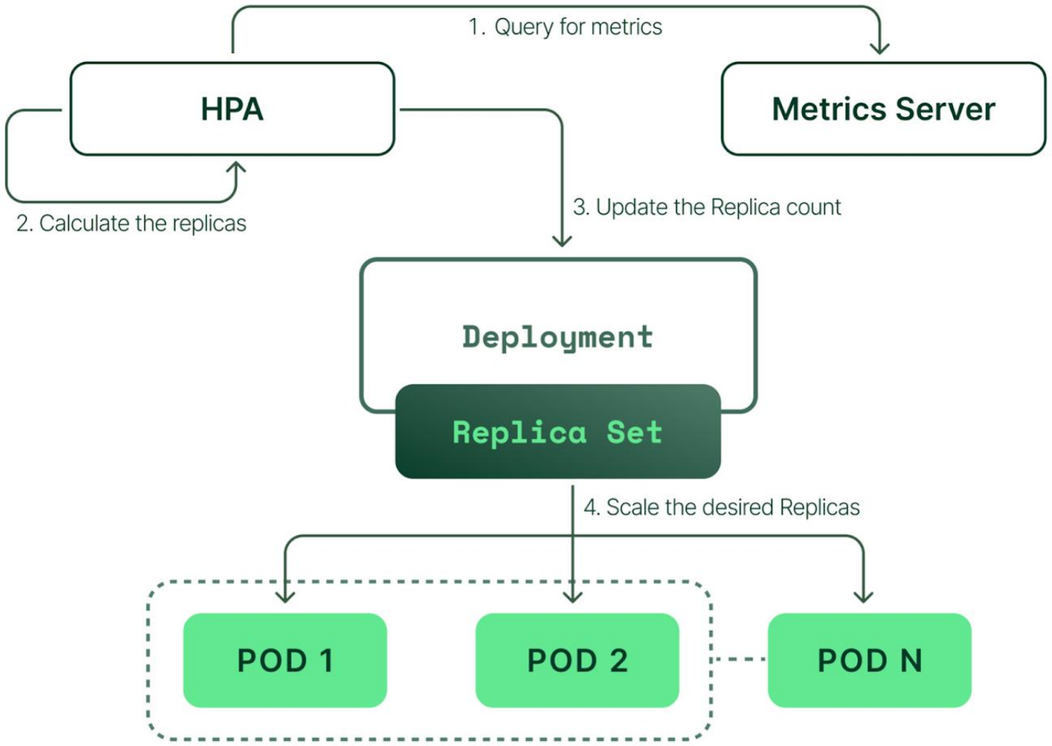
Kubernetes control plane can scale applications based on their resource utilization. As Pods become busier, the control plane brings up new replicas to share the load automatically.

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

Any application that does not operate 24-7 is considered inefficient. The applications are expected to run uninterrupted, even if there are technical glitches, feature updates, or natural disasters.

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.

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:

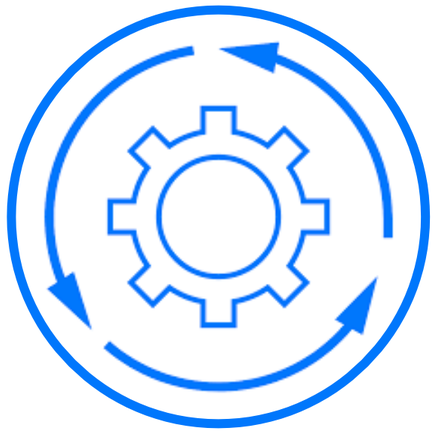
preStop

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.

Kubernetes' Self-Healing

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

Kubernetes' Self-Healing Concepts

Each probe provides one of three results:

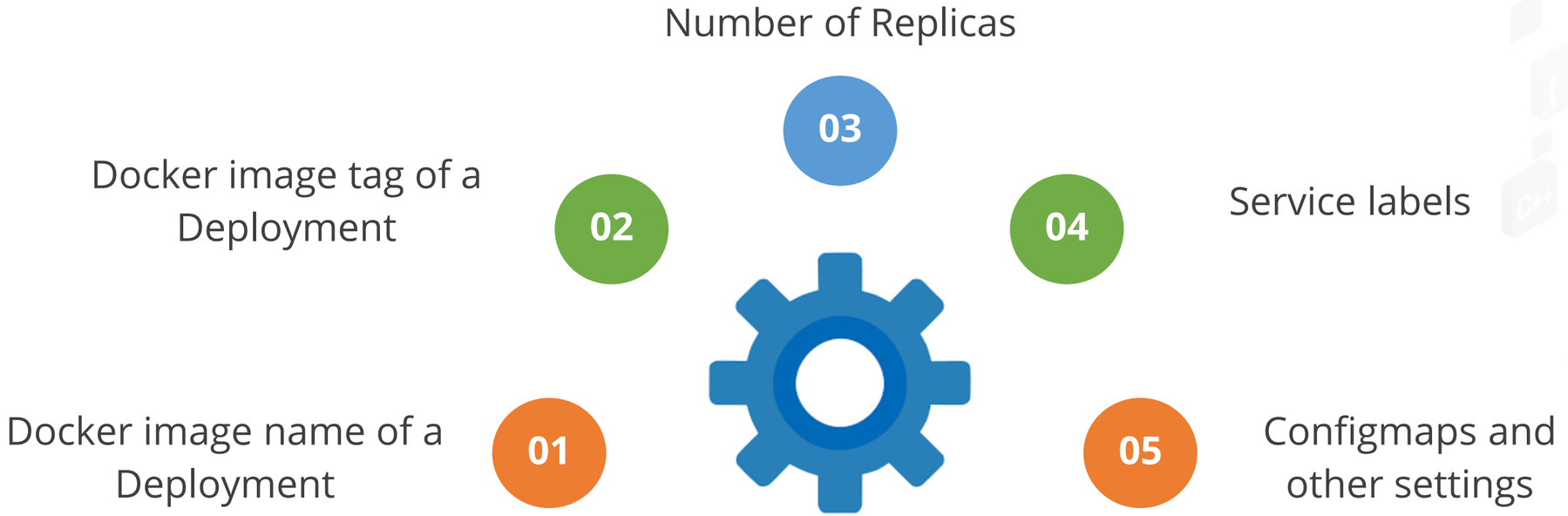
Failure	The container failed the diagnostic.
Unknown	The diagnostic failed, so no action can be taken.
Success	The container passed the diagnostic.

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 are used to control the Pods that are spread across the cluster, among failure domains such as regions, zones, nodes, and other user-defined topology domains
- A ReplicaSet ensures that a specified number of Pod replicas are running at any given time.
- Kubernetes control plane can scale applications based on their resource utilization. As Pods become busier, it automatically brings up new replicas to share the load.

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