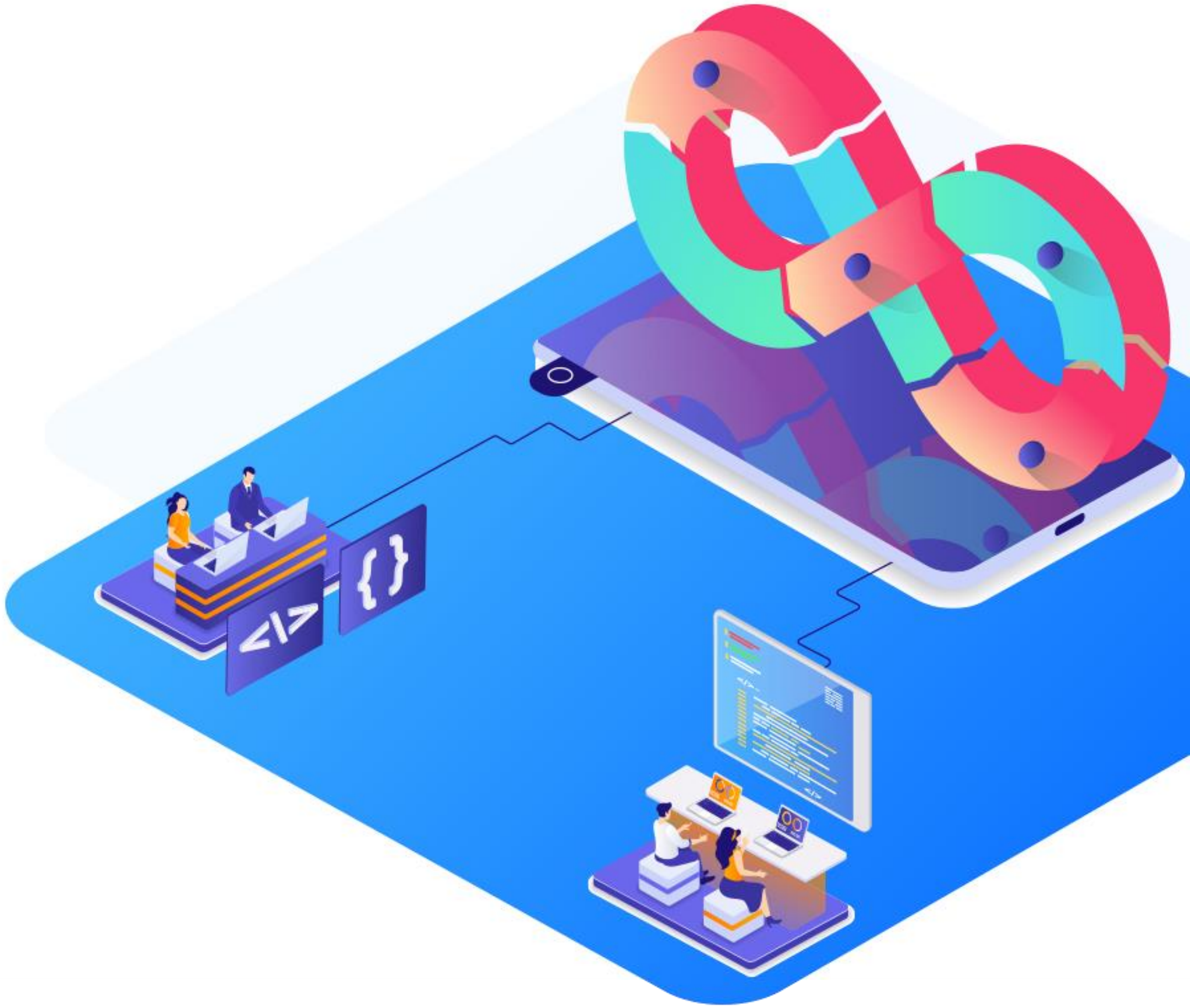
Container Orchestration Using Kubernetes



Scheduling

Learning Objectives

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

- Identify the key features of scheduler frameworks used in Kubernetes for workload management
- Apply node selectors, node affinity, and taints or tolerations to determine pod placement in a Kubernetes cluster
- Evaluate how topology management policies can be used to enhance fault tolerance and high availability in Kubernetes applications
- Demonstrate techniques to reduce pod overhead and optimize resource utilization within a Kubernetes environment

Scheduling: Overview

What Is a Scheduler?

The Kubernetes Scheduler is a control plane component that automatically assigns unscheduled Pods to the most appropriate nodes based on resource availability and constraints.

It ensures optimal resource utilization by selecting nodes that meet the Pods' specific resource and policy requirements.

How Does a Scheduler Work?

The default scheduler for Kubernetes is **kube-scheduler**, which operates as an integral component of the control plane.

Whenever a new pod is created or remains unscheduled, **kube-scheduler** picks the best node for it to run on.

Within a cluster, eligible nodes are those that fulfill the scheduling criteria specified for a pod.

The scheduler identifies suitable nodes for a pod and assesses them using a series of functions to assign scores. Among these nodes, it selects the one with the highest score to execute the pod.

Quick Check

You have deployed a new batch of pods in your Kubernetes cluster which remain unscheduled. You are tasked with analyzing why these pods have not been assigned to nodes yet.

Based on the kube-scheduler's role, which of the following steps is most likely responsible for the pods remaining unscheduled?

A. The kube-scheduler has not yet picked the best node due to high system load.

B. There are no nodes currently fulfilling the specified scheduling criteria for these pods.

C. The nodes have not been scored yet due to a temporary malfunction in the scoring functions.

D. The pods do not have the necessary taints or tolerations defined, preventing scheduling.

Frameworks in Scheduling

Scheduling: Framework

The scheduling framework in Kubernetes is designed as a flexible, pluggable architecture for the scheduler.

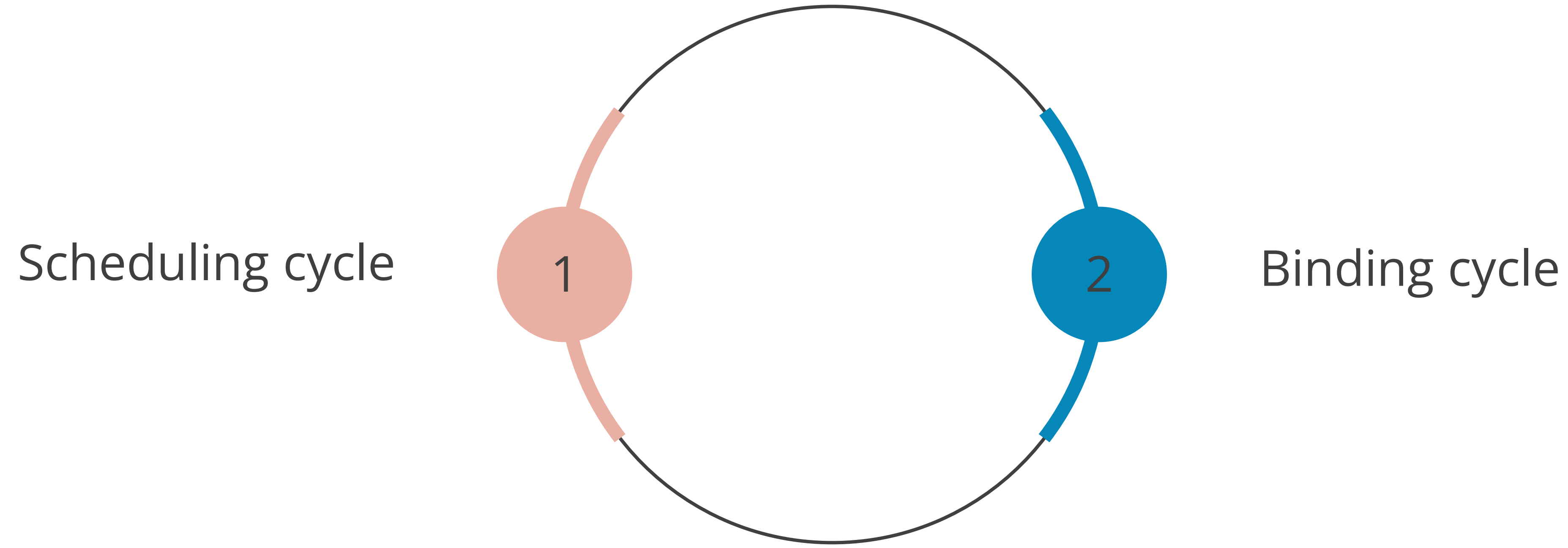
1. The scheduler introduces additional plugin APIs into the current scheduler infrastructure.

2. Plugins enable the implementation of many scheduling features through APIs.

Scheduling Framework Workflow

The scheduling framework defines extension points, and one or more of these points activate the registered scheduler plugins.

Scheduling a pod happens in two phases (at each attempt):



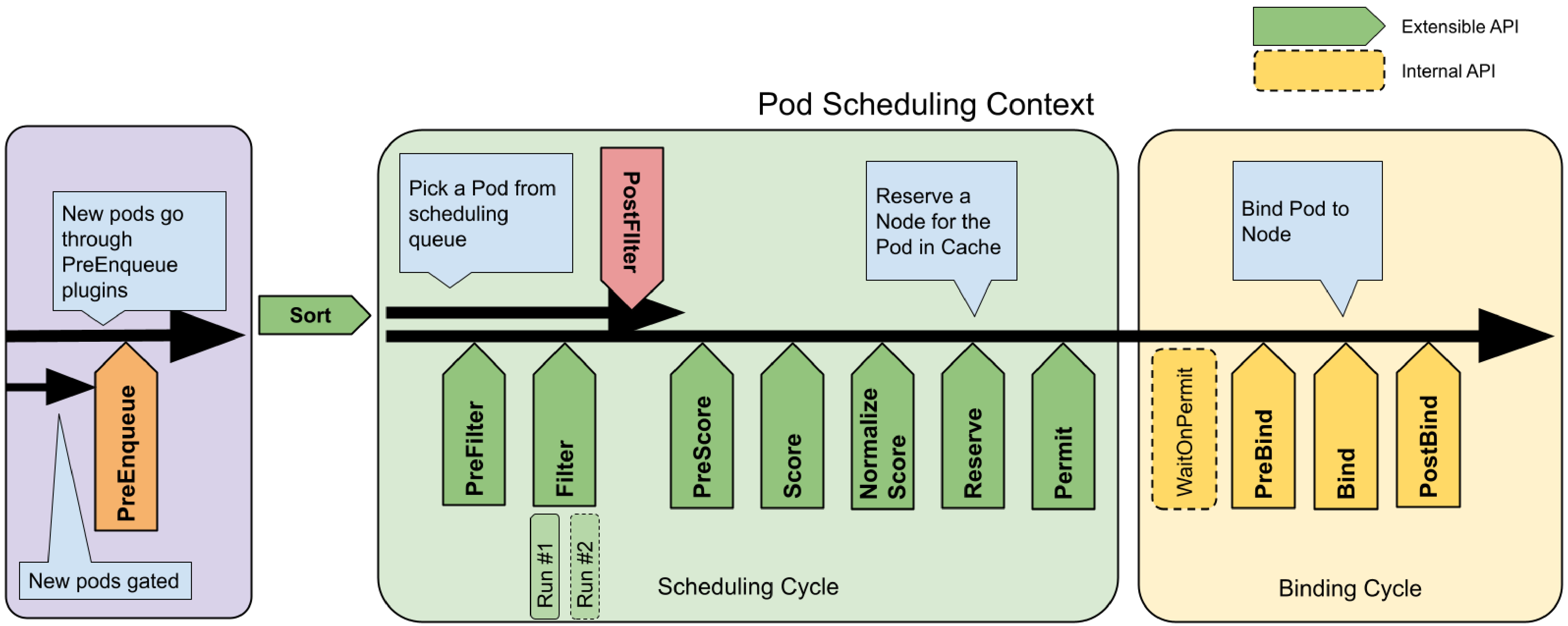
Binding Cycle and Scheduling Cycle

The binding cycle applies the decision to the cluster, and the scheduling cycle selects a node for the pod.

- The binding cycle and the scheduling cycle together form the scheduling context.
- Scheduling cycles are run serially, while binding cycles may run concurrently.
- If there is an internal error or a pod cannot be scheduled, the cycles may be aborted.

Extension Points

This diagram represents the pod's scheduling context. The scheduling framework provides the following extension points:



Source: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

Plugins for Scheduling Cycle and Binding Cycle

QueueSort

This sorts the pod in the scheduling queue. Essentially, this plug-in gives a less (Pod1, Pod2) function, ensuring the remaining plugins are not called.

PreFilter

This preprocesses information about the pod or checks the conditions that a pod or cluster must meet.

Filter

This filters out nodes that cannot run the pod. For the rest of the nodes, the remaining plugins are called.

PostFilter

These plugins are called after the filter phase when no viable nodes are found for the pod.

Plugins for Scheduling Cycle and Binding Cycle

Reserve

This implements reserve extension; it uses reserve and unreserve methods.

PreScore

This generates a shareable state for score plugins to use. If an error is returned by it, the scheduling cycle gets aborted.

Score

This ranks nodes that have passed the filtering phase. A range is defined which represents the minimum and maximum scores.

NormalizeScore

This changes scores before the scheduler computes the final ranks for the nodes.

Plugins for Scheduling Cycle and Binding Cycle

Permit

This is invoked at the end of the scheduling cycle for each pod to prevent or delay the binding to the candidate node.

PreBind

This performs any work needed before a pod is bound.

PostBind

This is an extension point (informational) and is called after a pod is successfully bound.

Bind

This is called only after PreBind plug-ins have completed their part; it binds a pod to a node.

Plugin Configuration

Plugins can be disabled or enabled in the scheduler configuration.

In Kubernetes v1.18 or later:

- Most scheduling plugins are utilized and enabled by default.
- New scheduling plugins can be configured and implemented along with default plugins.
- A set of plugins can be configured as a scheduler profile, and multiple profiles may be defined to fit different types of workloads.

Quick Check

You are scheduling a new batch of application pods with varying resource requirements in your Kubernetes cluster to optimize utilization and performance.
After the pods have passed the filtering phase, which Kubernetes scheduling plugin is crucial for determining the final node assignment by ranking the nodes?

A. The Score plugin ranks the nodes based on their suitability, using a defined range of scores.

B. The NormalizeScore plugin adjusts node scores for final ranking.

C. The PreScore plugin sets up the state for the Score plugins.

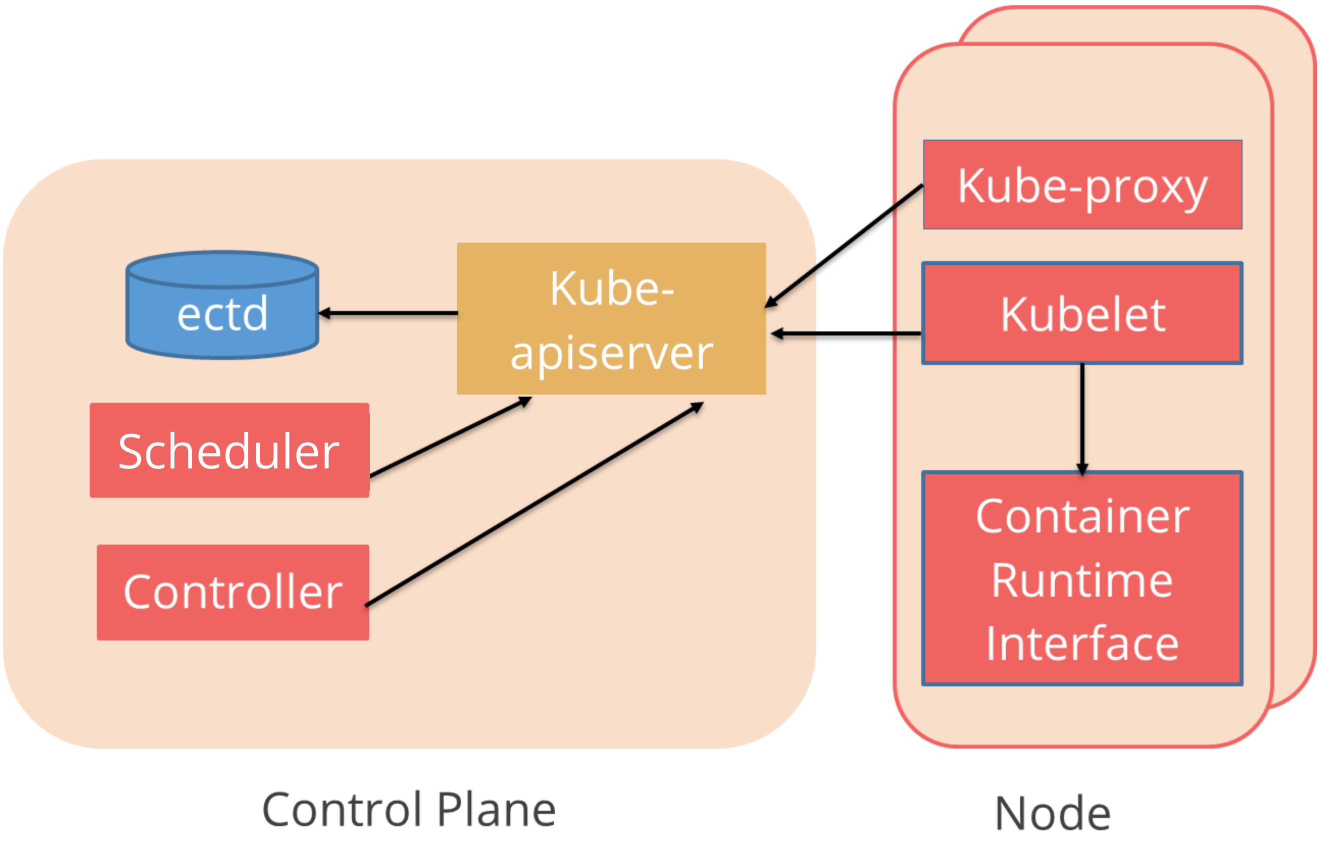
D. The Reserve plugin reserves resources on potential nodes.

Kube-scheduler

Kubernetes Scheduler

It is a control plane process that assigns pods to the nodes.

Below is the high-level view of a Kubernetes scheduler:



How Kubernetes Scheduler Works

The scheduler’s role is to ensure that each pod is assigned to a node on which it must run.

Step 01

Each pod that needs to be scheduled is added to a queue.

Step 02

New pod, if it is to be created, is also added to the queue.

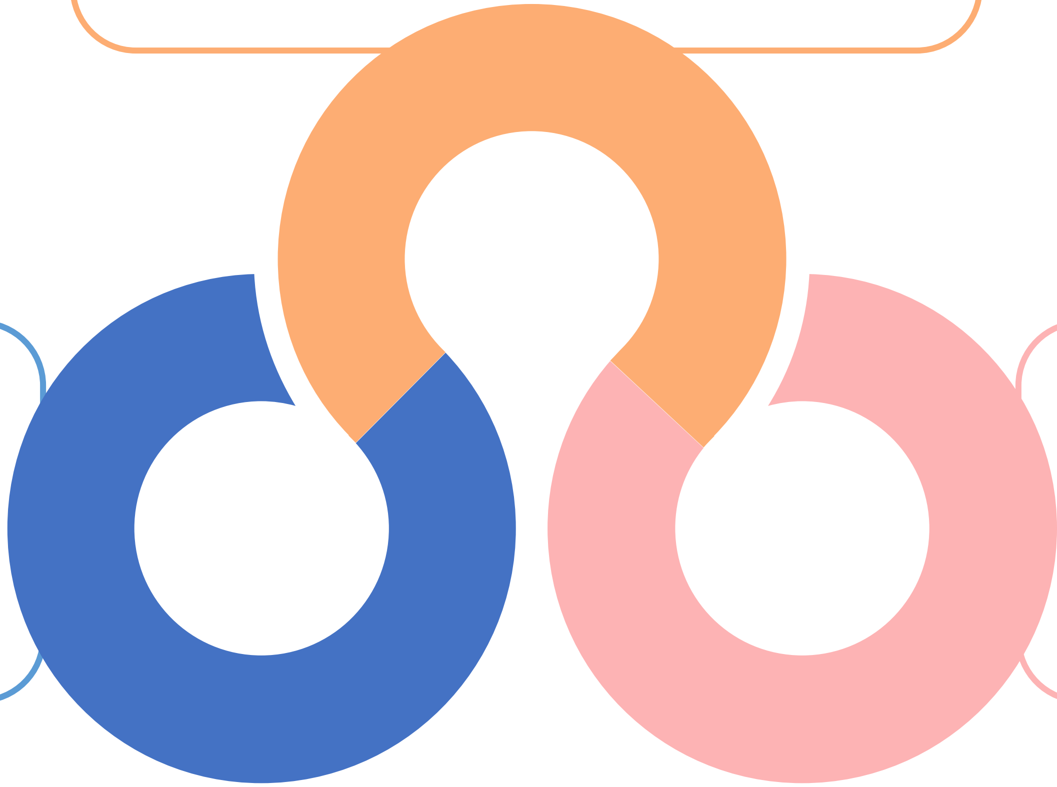
Step 03

The scheduler continuously dequeues pods and proceeds to schedule them.

Working of a Kubernetes Scheduler

It is responsible for the following three tasks:

Scheduling the newly created pods on nodes



Observing the unscheduled pods and binding them to nodes

Monitoring the **kube-apiserver** and controller for recently created pods and handling their scheduling

Running Multiple Schedulers

As the default scheduler runs on a default algorithm, in some specific use cases it might not function accordingly. Hence, a custom scheduler is required.

Furthermore, multiple schedulers along with the default scheduler can be run in a single cluster simultaneously.

For this, one needs to specify which schedulers should be used for each pod.

Node Selection in Kubernetes Scheduler

Node Selection in Kubernetes Scheduler

The Kubernetes scheduler selects a node for the pod in a two-step operation:

Filtering

It finds the set of nodes where it is viable to schedule a pod.

Scoring

It ranks the remaining nodes to choose the most apt pod placement.

kube-scheduler assigns the pod to the node with the highest ranking.

Node Selection in Kubernetes Scheduler

There are two methods to configure the scheduler's filtering and scoring behavior:

Scheduling policies	Permit the setup of criteria for filtering nodes and establishing scoring priorities
Scheduling profiles	Enable the configuration of plugins to implement different scheduling stages

Quick Check

You are setting up a new Kubernetes cluster and considering configuration options for the scheduler based on your application requirements. Which method should you choose to configure different plugins for various stages of the scheduling process?

A. Scheduling policies that allow configuration of predicates for filtering and priorities for scoring

B. Scheduling profiles that enable the configuration of plugins to implement different scheduling stages

C. Either method, as both allow detailed configuration of the scheduling process

D. Neither, as Kubernetes does not allow customizable scheduling behaviors

Working on Pod Allocation

Introduction

There are several ways to allocate pods, and all the recommended approaches use label and 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

04 Pod topology spread constraints

nodeName

It is the simplest form of node selection constraint, but is rarely used due to several limitations. These include:

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

02 If the specified node does not exist, the pod will not run or may be automatically deleted.

03 Insufficient resources on the specified node will cause the pod to fail, with the reason clearly indicated.

nodeName: Example

Here is an example of a pod configuration file that includes the nodeName field:

```
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**.

nodeSelector

It is the simplest recommended form of node selection constraint.

Run **kubectl get nodes –show-labels** command to get all the labels of the cluster nodes

```
labsuser@master:~$ kubectl get nodes --show-labels
NAME                 STATUS   ROLES           AGE   VERSION   LABELS
ip-172-31-29-25      Ready    <none>          11d   v1.28.3   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=ip
-172-31-29-25,kubernetes.io/os=linux
master.example.com   Ready    control-plane   11d   v1.28.3   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=ma
ster.example.com,kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=,node.kubernetes.io/exclude-from-external-load-balancers=
labsuser@master:~$
```

nodeSelector: Example

Here is an example of how to add a nodeSelector:

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

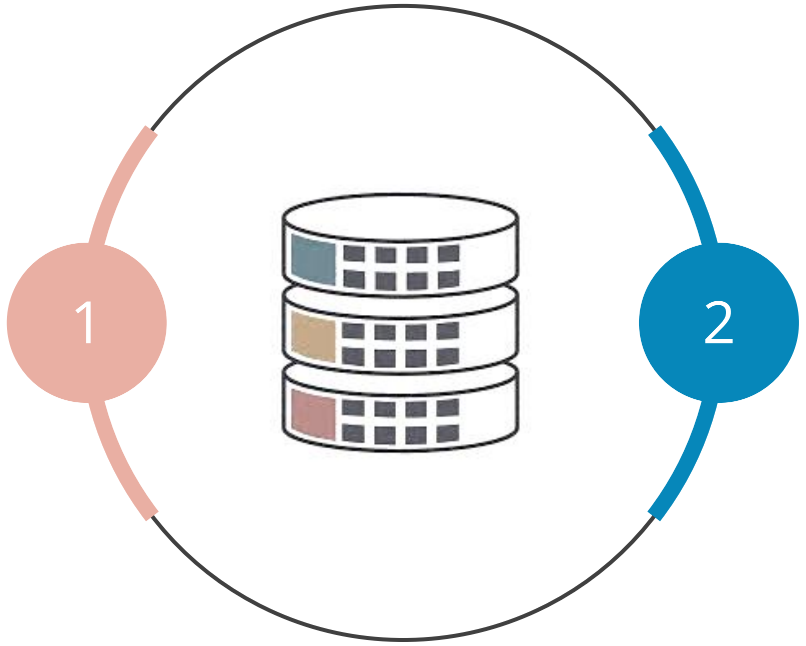
Run **kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml**; the pod gets scheduled on the node

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 as follows:

The affinity or anti-affinity language is more expressive.

By applying labels to other pods running on the node, users can restrict a pod.

Users have the option to specify that the rule is flexible or preferred, as opposed to being an inflexible requirement.

Inter-Pod Affinity and Anti-affinity

They enable the restriction of node selection for pod scheduling based on labels present on other pods that are already running on the node, instead of labels on the nodes themselves.

There are two types of inter-pod affinity and anti-affinity:

01 requiredDuringSchedulingIgnoredDuringExecution

02 preferredDuringSchedulingIgnoredDuringExecution

Inter-Pod Affinity and Anti-affinity: Example

Below is the example of a pod that uses pod affinity:

```
apiVersion: v1
   Kind:  pod
   Metadata:
      name:  with-pod-affinity
      spec:
        affinity:
        podAffinity:
         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
```

Inter-Pod Affinity and Anti-affinity

The rules for inter-pod affinity and anti-affinity are as follows:

Inter-pod affinity rule

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

Inter-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.

Pod Affinity and Anti-affinity: Redis-Cache Pod

```
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.**

Pod Affinity and Anti-affinity: Web-Server Pod

The following example shows the YAML snippet of the webserver deployment that has pod affinity and anti-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"
        podAffinity:

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

Constraints on TopologyKey

The Topologykey is a label used by Kubernetes to specify the rules that govern the placement of pods across different physical or logical groupings of nodes within a cluster.

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

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

Taints and Tolerations

Taints and tolerations collaborate to prevent pods from being scheduled on unsuitable nodes.

Taints

- Taints prevent a specific set of pods from being scheduled on a node, in contrast to Node affinity.
- They are applied to nodes.

Tolerations

- Tolerations enable the scheduler to assign pods to nodes that have compatible taints.
- They are applied to pods.

Taints and Tolerations

The **kubectl taint nodes** command in Kubernetes is used to apply a taint to a node.

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

In the given command,

- node1 is the node name to which the taint is being applied.
- key1=value1 represents the key-value pair associated with the taint.
- NoSchedule indicates the effect of the taint.

Taints and Tolerations: Example

Example to specify tolerations in the pod:

```
apiVersion: v1
kind:  pod
metadata:
   name:  nginx
 spec:
     containers:
     -  name: nginx
        image: nginx
      tolerations:
       - key: "key1"
         operator: "Equal"
         value: "value1"
         effect: ”NoSchedule"
```

Taints and Tolerations

Below are the effects used in taints and tolerations:

NoExecute

When a node is tainted with a NoExecute effect, it will result in the immediate eviction of any existing pods running on that node. This action affects the pods already present on the node.

NoSchedule

New pods will not be scheduled on the tainted node unless they possess a corresponding toleration. However, existing pods running on the node will not be evicted.

Assisted Practice

Configuring Pods with nodeName and nodeSelector Fields **Duration: 10 Min.**

Problem statement:

You have been given a task to configure pods with nodeName and nodeSelector fields for efficient resource use, compliance, and specific application needs in a cluster.

Outcome:

By the end of this demo, you will be able to configure pods with nodeName and nodeSelector fields to optimize resource usage, ensure compliance, and meet specific application requirements within a Kubernetes cluster.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create pods with the fields nodeName and nodeSelector
2. Assign label to the nodes
3. Create a pod with the NotIn operator

Assisted Practice

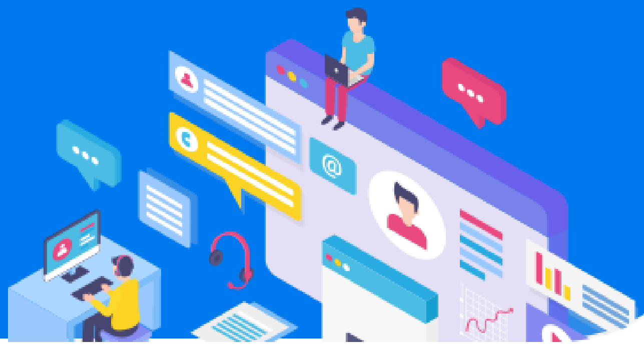
Configuring Pod Affinity and Anti-Affinity in Kubernetes **Duration: 10 Min.**

Problem statement:
You have been assigned a task to configure pod affinity and anti-affinity rules in a Kubernetes cluster to ensure specific deployment patterns of pods across nodes.

Outcome:
By the end of this demo, you will be able to configure pod affinity and anti-affinity rules to manage the deployment patterns of pods across nodes in a Kubernetes cluster.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Deploy redis-cache with anti-affinity
2. Colocate web server with redis-cache using affinity

ASSISTED PRACTICE

Quick Check

You are configuring a Kubernetes pod that requires a high-security environment and want to ensure it's only scheduled on appropriate nodes.
To restrict this pod to specific secure nodes, what should you apply to the other nodes?

A. Taints that prevent unsuitable pods from being scheduled
B. Tolerations matching the secure nodes' taints
C. Node affinity rules for high-security nodes
D. Labels indicating security levels on nodes

Scheduler Performance Tuning

Kubernetes Scheduler

Kubernetes scheduler is the default scheduler for Kubernetes.

- It places the Pods on Nodes in a cluster.
- Viable Nodes for the Pods are Nodes in a cluster that meet the scheduling requirements of a Pod.
- In large clusters, users can tune the scheduler's behavior, balancing scheduling outcomes between accuracy and latency.
- Users may configure this setting via the **kube-scheduler** by setting the **percentageOfnodesToScore**.

Setting the Threshold

The percentageOfnodesToScore option accepts whole numeric values between 0 and 100.

To change the value, users may first edit the **kube-scheduler** configuration file and then restart the scheduler.

Configuration file path:

/etc/kubernetes/config/kube-scheduler.yaml.

Users can run the following command to check the health of **kube-scheduler**:

```
kubectl get Pod -n kube-system | grep kube-scheduler
```

Node Scoring Threshold

When enough nodes are found, the **kube-scheduler** stops looking for viable nodes, which improves the scheduling process's performance.

Threshold scoring is performed as follows:

Users set a threshold as a percentage (whole number) of all the nodes present in the cluster.

While scheduling, if **kube-scheduler** identifies enough viable nodes to exceed the configured percentage, it moves to the scoring phase.

If users do not set a threshold, Kubernetes calculates a figure using a linear formula that yields 50% for a 100-Node cluster and 10% for a 5000-node cluster.

Example

Here is an example of a configuration that sets **percentageOfnodesToScore** to 50%:

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

percentageOfnodesToScore: 50
```

Tuning percentageOfnodesToScore

The **percentageOfnodesToScore** must be a value between 1 to 100. The default value is calculated based on the cluster size.

Scheduler checks all nodes in clusters with less than 50 viable nodes.

The change is ineffective if a small value is set for **percentageOfnodesToScore**.

When the cluster has 100 nodes or less, it is optimal to set the configuration option at the default value.

How the Scheduler Iterates Over Nodes

If nodes are in multiple zones, the scheduler iterates over them to ensure that the nodes from different zones are considered in feasibility checks.

The example shown below has six nodes in two zones:

```
Zone 1: Node 1, Node 2, Node 3, Node 4
Zone 2: Node 5, Node 6

// The scheduler evaluates the feasibility of the Nodes
in the following order:

Node 1, Node 5, Node 2, Node 6, Node 3, Node 4

// After going over all Nodes, it goes back to Node 1.
```

Quick Check

You are optimizing node scoring in a Kubernetes cluster by adjusting the percentageOfNodesToScore setting. The cluster has varying numbers of nodes.
What happens when you set percentageOfNodesToScore to a very low value in a small cluster with less than 50 nodes?

A. All nodes are still scored, making the change ineffective.

B. Only a few nodes are scored, which might affect the scheduling accuracy.

C. The scheduler cannot find enough nodes to score, causing errors.

D. The scheduling process speeds up significantly due to fewer scores.

Scheduling Policies

Scheduling Policies

A scheduling policy can be used to set the predicates and priorities that the kube-scheduler runs to filter and provide scores to the nodes, respectively.

Setting a scheduling policy

- **Run:** kube-scheduler --policy-config-file <filename> or kube-scheduler --policy-configmap <ConfigMap>
- **Use:** Policy type

Note

Scheduling policies are unsupported in Kubernetes starting from version 1.23.

Predicates

The predicates that implement filtering are:

podFitsHostPorts

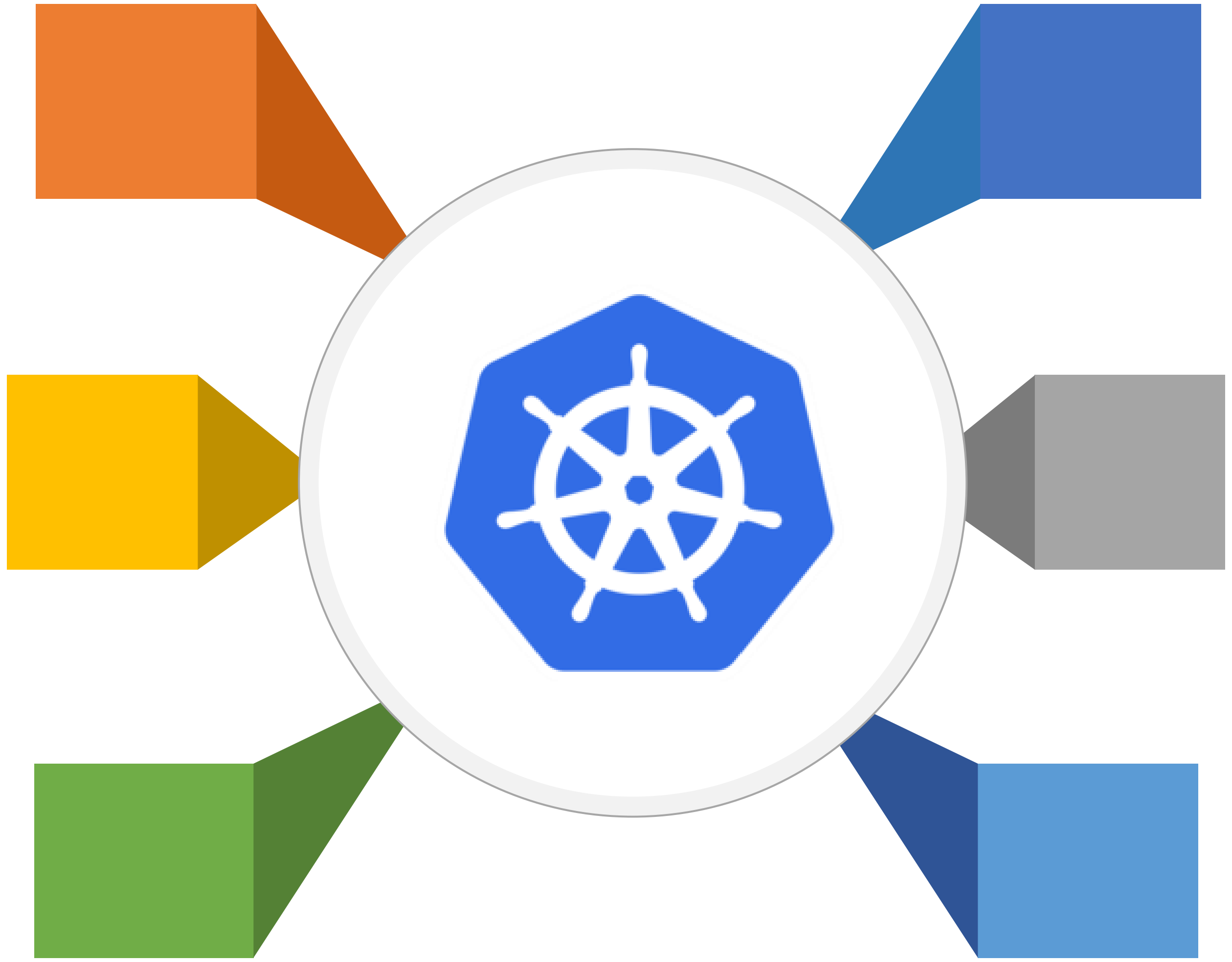
Matchnodeselector

podFitsHost

ChecknodePIDPressure

podFitsResources

ChecknodeDiskPressure



Predicates

The predicates that implement filtering are:

- NoVolumeZoneConflict
- NoDiskConflict
- MaxCSIVolumeCount
- ChecknodeMemoryPressure
- ChecknodeCondition
- podToleratesnodeTaints

Priorities

The priorities that implement scoring are:

1. SelectorSpreadPriority
2. InterpodAffinityPriority
3. LeastRequestedPriority
4. MostRequestedPriority
5. RequestedToCapacityRatioPriority
6. BalancedResourceAllocation
7. nodePreferAvoidpodPriority
8. TaintTolerationPriority
9. ImageLocalityPriority
10. ServiceSpreadingPriority
11. nodeAffinityPriority
12. EqualPriority
13. EvenpodspreadPriority

Quick Check

You are configuring a Kubernetes scheduler to improve pod placement efficiency in your cluster. You are considering using a scheduling policy for fine-tuning the scheduler's behavior. How would you apply a scheduling policy using a file to configure predicates and priorities in the kube-scheduler?

A. Use a command-line option with the kube-scheduler to specify a policy-config-file

B. Directly modify the kube-scheduler's source code to include the policy

C. Deploy a special ConfigMap in Kubernetes to automatically apply the policy

D. None of the above, as you cannot set scheduling policies in Kubernetes

Scheduling Profiles

Scheduling Profiles: Introduction

A scheduling profile permits the configuration of different stages of scheduling in the **kube-scheduler**.

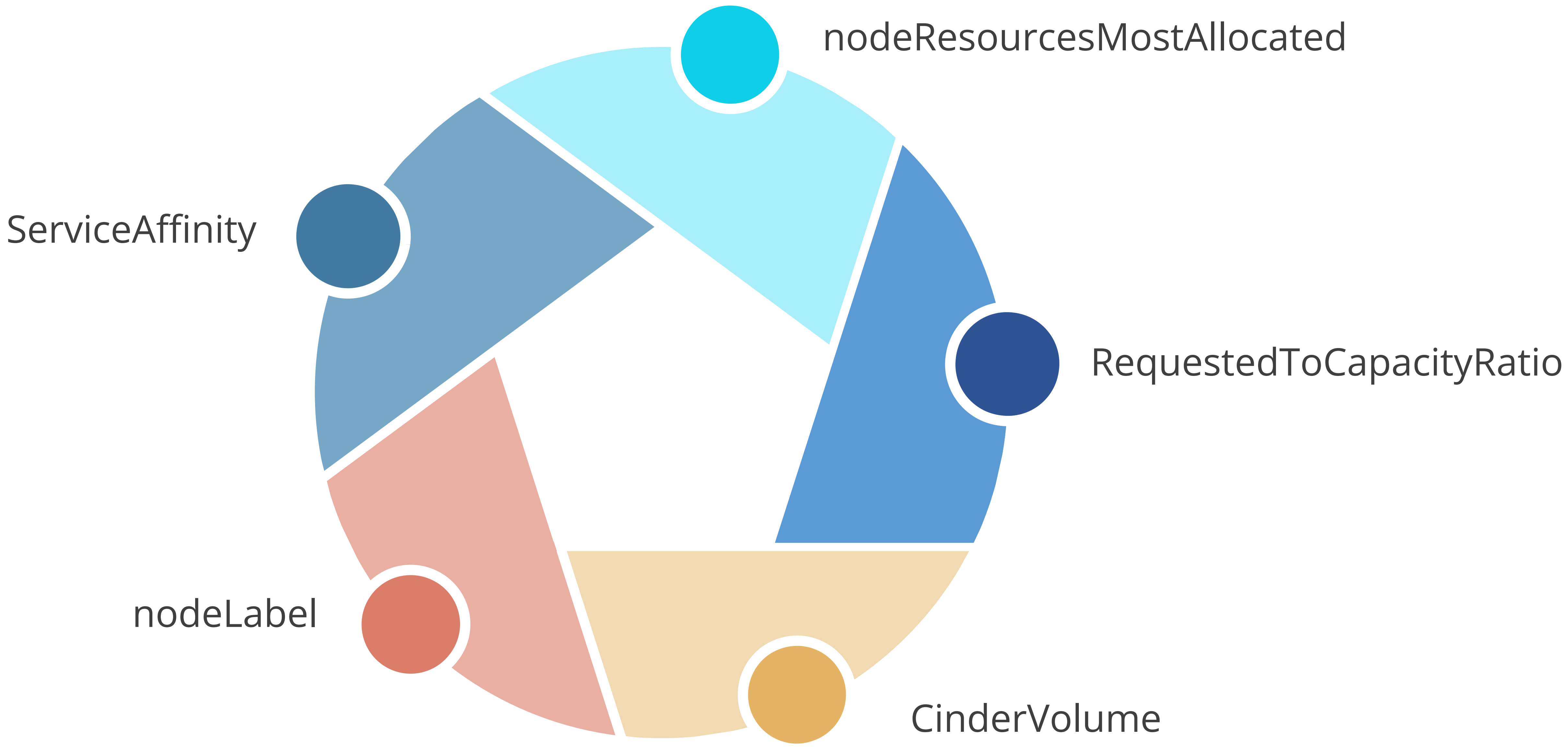
Each stage is exposed at an extension point.

Plugins provide the scheduling behavior via the implementation of one or more extension points.

A single instance of **kube-scheduler** may be configured to run multiple profiles.

Scheduling Plugins

Below are some plugins that are not enabled by default and need to be enabled through the component config APIs:



Multiple Profiles

Using a configuration (as shown below), the scheduler will operate with two profiles: one enabling default plugins and the other with all scoring plugins disabled.

```
apiVersion:
kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: no-scoring-scheduler
    plugins:
      preScore:
        disabled:
        - name: '*'
      score:
        disabled:
        - name: '*'
```

Note

For a pod to be scheduled based on a particular profile, it should include the corresponding scheduler name in its **.spec.schedulerName** field.

Quick Check

As a Kubernetes administrator configuring a multi-application cluster, you are evaluating the use of multiple scheduling profiles to optimize pod placement. Why would you configure multiple scheduling profiles within a single instance of the kube-scheduler?

A. To customize scheduling behaviors for each application using specific plugins

B. To increase the kube-scheduler's capacity

C. To reduce the kube-scheduler's resource usage

D. To speed up scheduling by running profiles simultaneously

Topology Management Policies

Topology Manager: Overview

It serves as a kubelet component that orchestrates a group of components accountable for optimization.

More and more systems are using both CPUs and hardware accelerators to handle tasks requiring low latency and high-throughput parallel computation.

To achieve optimal performance in scheduling, it is essential to focus on optimizations related to CPU isolation, device proximity, and memory usage.

How Topology Manager Works

The topology manager serves as a definitive source, guiding other kubelet components in making resource allocation choices aligned with topology.

It offers a platform for components, referred to as Hint Providers, to transmit and receive topology information through an interface.

It obtains topology information from the Hint Providers in the form of a bitmask representing the accessible NUMA Nodes and indicating the preferred allocation.

Scopes and Policies

The topology manager aligns:

- Pods of all QoS classes
- The resources for which the Hint Provider offers topology hints

The scope defines the granularity of resource alignment.

Policies

The topology manager supports four allocation policies:

1. none policy
2. restricted policy
3. single-numa-node policy
4. best-effort policy

Known Limitations

The limitations of the topology manager include:

1. The maximum number of NUMA nodes allowed by the topology manager is eight.

2. The scheduler lacks awareness of topology and is scheduled on a node; however, its reliability relies on the topology manager.

Security Context

A security context specifies privilege and access control configurations for a pod or container.

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
 runAsUser: 1000
 runAsGroup: 3000
 fsGroup: 2000
  containers:
  - name: sec-ctx-demo
    image: busybox:1.28
```

Assisted Practice

Working with Kubernetes Security Context **Duration: 10 Min.**

Problem statement:

You have been assigned a task to demonstrate the process of configuring a Kubernetes security context and validating its settings, followed by gaining shell access to a running container within the cluster.

Outcome:

By the end of this demo, you will be able to configure a Kubernetes security context, validate its settings, and gain shell access to a running container within the cluster.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create and verify the security context
2. Access the shell within the running container

ASSISTED PRACTICE

Quick Check

As a Kubernetes administrator, you are setting up an application in Kubernetes that requires high computational resources and precise resource allocation, considering the use of the Topology Manager. Why is the Topology Manager crucial for scheduling a Kubernetes pod that demands low latency and high-throughput computation?

A. It ensures optimal CPU and hardware accelerator placement for pods.

B. It reduces the overall pod capacity on a node.

C. It increases latency between pods and hardware.

D. It simplifies scheduling by disregarding device proximity.

Pod Overhead

Introduction

Pod overhead accounts for resources consumed by the pod infrastructure.

The pod overhead is set at admission time. It is based on the overhead associated with a Pod's **RuntimeClass**.

When enabled, it is considered along with all container resource requests made during a pod's scheduling.

Using Pod Overhead

To use the pod overhead feature, a RuntimeClass that defines the overhead field is necessary.

Here is a RuntimeClass definition with a virtualizing container runtime:

Example

```
kind: RuntimeClass
apiVersion: node.k8s.io/v1
metadata:
  name: kata-fc
handler: kata-fc
overhead:
  podFixed:
    memory: "120MiB"
      cpu: "250m"
```

This RuntimeClass configuration allocates approximately 120MiB per pod for the VM and the guest OS.

Using Pod Overhead

In this example, verification of the container requests is done for the workload:

Example

```
kubectl get Pod test-pod -o jsonpath='{.spec.containers[*].resources.limits}’

//  The total container requests are 2000m CPU and 200MiB of memory:

map[cpu: 500m memory:100Mi] map[cpu:1500m memory:100Mi]
```

To verify this against the node's current observations, use the below example:

Example

```
The output shows 2250m CPU and 320MiB of memory are requested, which includes
PodOverhead:

 Namespace                    Name                    CPU Requests  CPU Limits   Memory
Requests  Memory Limits  AGE
 ---------                    ----                    ------------  ----------   ------------
---  -------------  ---
(1%)          320Mi  (1%)        36m
```

Verify Pod Cgroup Limits

You can check the Pod's memory cgroups on the Node where the workload is in operation.

Here is the example:

Example

```
Pod First, on the particular Node, determines the identifier:
#Run this on the Node where the Pod is scheduled
pod_ID="$(sudo crictl Pod --name test-pod -q)"

From this, you can determines the cgroup path for the Pod:
#Run this on the Node where the Pod is scheduled
sudo crictl inspectp -o=json $pod_ID | grep cgroupsPath

The resulting cgroup path includes the Pod's pause container. The Pod-level cgroup is one
directory above.
        "cgroupsPath": "/kubepod/podd7f4b509-cf94-4951-9417-
d1087c92a5b2/7ccf55aee35dd16aca4189c952d83487297f3cd760f1bbf09620e206e7d0c27a"
```

Verify Pod Cgroup Limits

crictl is used on the node, which provides a CLI for CRI-compatible container runtime.

Example

```
In this specific case, the Pod cgroup path is kubepod/podd7f4b509-cf94-4951-9417-
d1087c92a5b2. Verify the Pod-level cgroup setting for memory:

# Run this on the Node where the Pod is scheduled.
# Also, change the name of the cgroup to match the cgroup allocated for your Pod.
 cat /sys/fs/cgroup/memory/kubepod/podd7f4b509-cf94-4951-9417-
d1087c92a5b2/memory.limit_in_bytes

This is 320 MiB, as expected:
335544320
```

Quick Check

You are configuring a Kubernetes cluster to handle workloads that include both containerized applications and virtual machines. To manage resource allocation more effectively, you plan to use the pod overhead feature. Why is it necessary to define a RuntimeClass with specified overhead for pods utilizing virtualized container runtimes?

A. To ensure precise resource allocation by accounting for additional VM and guest OS needs

B. To reduce the memory and CPU resources allocated to the pod

C. To streamline deployment by automating resource scaling

D. To enhance the overall performance of the Kubernetes cluster

Performance Tuning

Introduction

Performance tuning aims to optimize infrastructure utilization and cut costs rather than simply scaling up in response to environmental demands.

Introduction

Users can follow these ten steps to optimize their application's performance for optimal results:

1. Build container-optimized images
2. Define the resource profile to match application requirements
3. Configure node affinities
4. Configure pod affinities
5. Configure tolerances and taints
6. Configure pod priorities
7. Configure Kubernetes features
8. Optimize etcd cluster
9. Deploy the Kubernetes cluster in proximity to your customers
10. Gain insights from metrics

Build Container-Optimized Images

A container-optimized image reduces container image's size, facilitating fast retrievals. Now, Kubernetes can run the resultant container more efficiently.

A container-optimized image must:

- Have a single application or perform one operation
- Have only small images
- Use container-friendly OS(s)
- Use multistage builds to maintain a small app size
- Have health and readiness check endpoints

Configure Node Affinities

It helps in performance tuning when the Pod placement is specified.
Node Affinity helps define where the Pods must be deployed. This can be done in two ways:

1. Use a nodeSelector with the relevant label in the spec section
2. Use nodeAffinity of the affinity field in the spec section

Configure Pod Affinities

Kubernetes aids in changing and rechanging Pod Affinity configurations in terms of the currently operational Pods.

There are two fields available under podAffinity of the Affinity field in the spec section:

necessaryDuringSchedulingIgnoredDuringExecution

preferredDuringSchedulingIgnoredDuringExecution

Configure Pod Priorities

Kubernetes **PriorityClass** defines and enforces a certain order.

Here is how a priority class can be created:

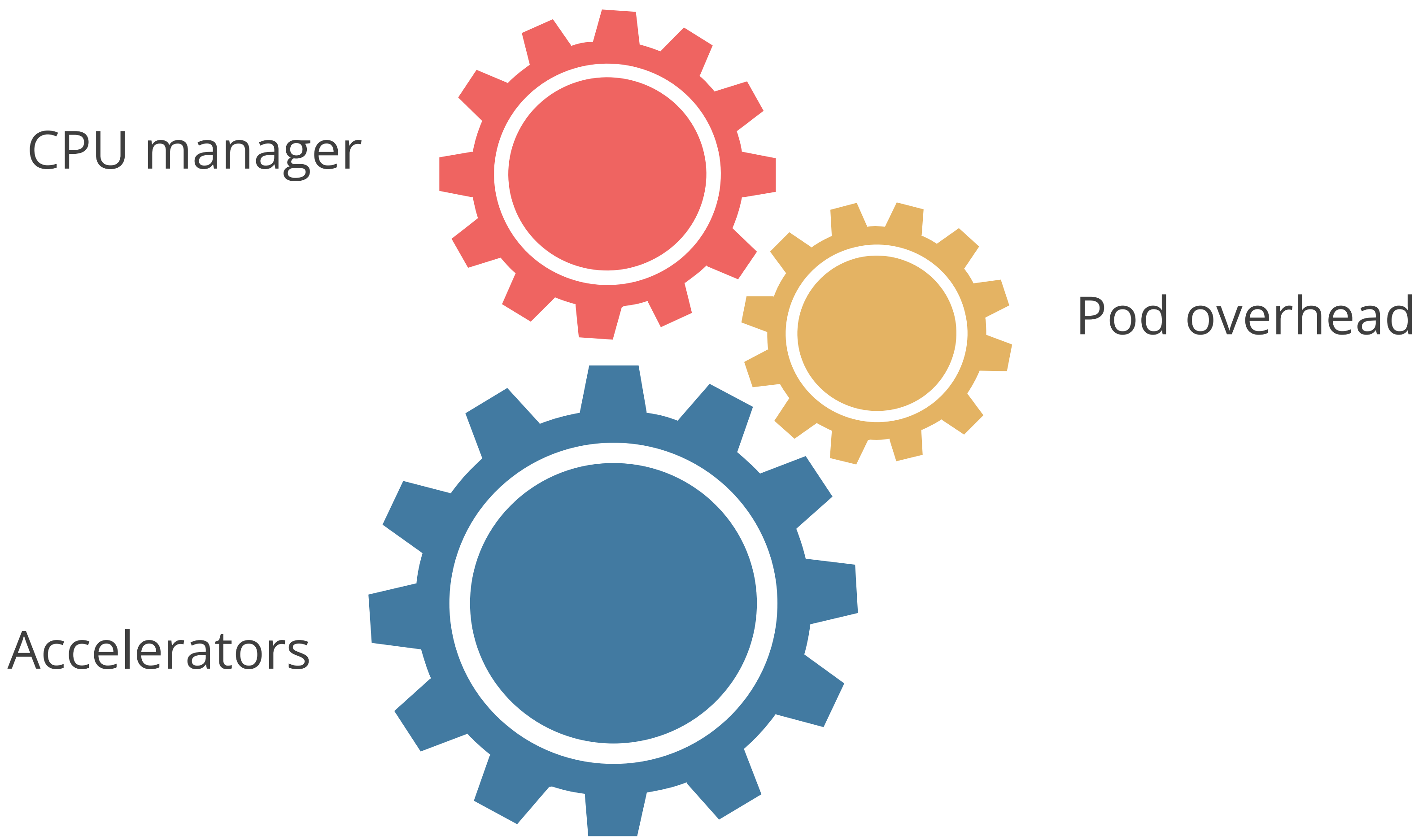
Example

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: highPriority
value: 1000000
globalDefault: false

description: "This priority class should be
used for initial Node function validation."
```

Configure Kubernetes Features

Kubernetes uses a feature gate framework that allows administrators to enable or disable environment features.

Here are the features that boost the scaling performance:



Optimize Etcd Cluster

Etcd is the brain of Kubernetes and is in the form of a distributed key-value database.

Note

Deploying nodes equipped with solid state disks (SSDs) that feature low I/O latency and high throughput optimizes database performance.

Assisted Practice

Creating and Configuring Pod Priority **Duration: 10 Min.**

Problem statement:
You have been asked to create and configure priority classes and assign them to pods in a Kubernetes environment.

Outcome:
By the end of this demo, you will be able to create and configure priority classes and effectively assign them to pods within a Kubernetes environment.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines

Steps to be followed:

1. Create and describe the priority class object
2. Create and describe the pod priority line

Quick Check

You are setting up a Kubernetes cluster and need to manage pod placement according to specific workload affinities. What are the two types of pod affinity rules available for configuring how tightly coupled certain pods should be to others during their scheduling and runtime?

A. NecessaryDuringSchedulingIgnoredDuringExecution and PreferredDuringSchedulingIgnoredDuringExecution

B. AlwaysDuringSchedulingAlwaysDuringExecution and SometimesDuringSchedulingSometimesDuringExecution

C. RequiredAtSchedulingRequiredAtExecution and OptionalAtSchedulingOptionalAtExecution

D. MandatoryDuringSchedulingMandatoryDuringExecution and VoluntaryDuringSchedulingVoluntaryDuringExecution

Managing Resources

Organizing Resource Configurations

Various resources can be streamlined by consolidating them within a single file. In a YAML file, this separation can be indicated by --- as demonstrated below:

Example

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
```

Example

```yaml
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Organizing Resource Configurations

The below code snippet can be used to create multiple resources:

Example

```
kubectl apply -f https://k8s.io/examples/application/nginx-app.yaml

service/my-nginx-svc created
deployment.apps/my-nginx created

kubectl apply also accepts multiple -f arguments:

kubectl apply -f
https://k8s.io/examples/application/nginx/nginx-svc.yaml -f
https://k8s.io/examples/application/nginx/nginx-deployment.yaml

A directory can be specified rather than or in addition to individual
files:

kubectl apply -f https://k8s.io/examples/application/nginx/
```

Bulk Operations in Kubectl

The kubectl performs several bulk operations including creating resource, extracting resource names from config files, and deleting resources. This process is shown below:

Example

```
kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml

deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted

In the case of two resources, you can set both resources on the command line using the
resource/name syntax:

kubectl delete deployments/my-nginx services/my-nginx-svc

For a larger number of resources, you'll find it easier to set the selector (label
query), specified using -l or --selector, to filter resources by their labels:

kubectl delete deployment,services -l app=nginx

deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

Using Labels

A few scenarios that use many labels are illustrated below, such as a guestbook that requires each tier to be distinguished:

Example

```
// Front end labels
labels:
        app: guestbook
        tier: frontend

// While the Redis master and slave would have different tier labels,
and perhaps, even an additional role label:
    labels:
        app: guestbook
        tier: backend
        role: master
and
    labels:
        app: guestbook
        tier: backend
        role: slave
```

Using Labels

Labels enable the ability to categorize and segment resources based on any specified dimension.

Example

```
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
```

Updating Labels

Labels are adjusted using **kubectl label** command to relabel existing pods and other resources before creating new ones.

Example

```
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 are non-identifying metadata that API clients like tools and libraries retrieve. This can be done using **kubectl annotate** as demonstrated below:

Example

```
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

Example

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

- It 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.

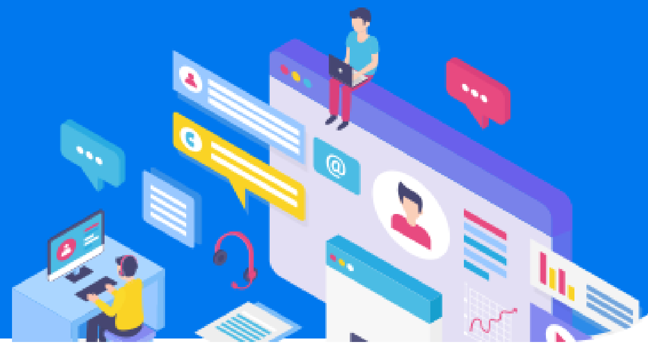
Kubectl Edit

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
```

Assisted Practice



Deploying the Flask Application with Redis **Duration: 10 Min.**

Problem statement:
You have been assigned a task to deploy and verify a Flask application integrated with Redis in a Kubernetes environment, demonstrating end-to-end containerized application setup and management.

Outcome:
By the end of this demo, you will be able to deploy and verify a Flask application integrated with Redis, demonstrating comprehensive containerized application setup and management in a Kubernetes environment.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a directory and add the necessary files
2. Create and tag the Flask image
3. Log into Docker and push the Flask image
4. Create the Redis and Flask deployments
5. Create the Redis and Flask services
6. Verify the Flask application deployment

ASSISTED PRACTICE

Quick Check

You are deploying an updated version of an Nginx application on your Kubernetes cluster using kubectl apply. What does kubectl apply do when you update your deployment configuration?

A. Overwrites all existing configurations with the new specifications

B. Deletes the previous version of the application before applying the new one

C. Pushes and compares configuration changes, applying updates without overwriting unspecified properties

D. Only updates configurations that have been manually changed

Key Takeaways

- Scheduling ensures that pods are matched to nodes so that Kubelet can run them.
- Kube-scheduler is designed to allow users to create their scheduling component if desired.
- Taints and tolerations collaborate to prevent pods from being scheduled on unsuitable nodes.
- The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of plugin APIs to the existing scheduler.

Updating Httpd Docker Images in a Kubernetes Cluster

Duration: 25 minutes

Project agenda: To systematically update the Docker image versions of the httpd web server in a Kubernetes cluster, ensuring seamless and controlled rollouts to enhance server capabilities without disrupting service

Description: The project involves testing the rollout of different Docker images for the httpd web server within a Kubernetes cluster to ensure the cluster's efficient management of updates and versions of web server applications.

LESSON-END PROJECT

Updating Httpd Docker Images in a Kubernetes Cluster

Duration: 25 minutes

LESSON-END PROJECT

Steps to perform:

1. Create the httpd deployment
2. Update the image version from httpd:2 to httpd:2.2
3. Update the image version from httpd:2.2 to httpd:2.4

Expected deliverables: A Kubernetes cluster with the testing of httpd docker images

Thank You