Container Orchestration using Kubernetes

TECHNOLOGY

Scheduling

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

A Day in the Life of a DevOps Engineer

You are working as a DevOps Developer in an organization which has decided test Kubernetes commands and develop a configuration file. The company has decided to test different Docker images of Httpd web server using the rollout process.

You need to run the Kubernetes command sequentially, followed by setting up the cluster to create a Httpd Deployment along with updating the image version from httpd:2 to httpd:2.2 and verifying the rollout status

To achieve all the above, along with some additional features, 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:

- Summarize an overview of scheduler and scheduling frameworks
- Describe kube-scheduler, rolling updates, rollbacks and DaemonSet
- Explain scheduling policies, scheduling profiles and management policies
- Classify the steps involved for the best performance tuning
- Associate on how to manage the resources for workloads

TECHNOLOGY

Scheduling Overview

Scheduler Introduction

A scheduler looks for Pods that have been freshly created and have no Nodes assigned to them.

For each Pod discovered, it becomes the responsibility of the scheduler to find the best Node for that *Pod* to run on.

Scheduler Introduction

kube-scheduler is the default scheduler for Kubernetes and runs as a part of the control plane.

For each newly created or unscheduled Pod, kube-scheduler selects an optimal Node for it to run on.

In a cluster, viable Nodes are the Nodes that meet the scheduling requirements for a Pod.

The scheduler finds viable Nodes for a Pod and runs a set of functions to score the viable Nodes. From the viable Nodes, it picks the Node with the highest score to run the Pod.

TECHNOLOGY

Scheduling Frameworks

Scheduling Framework

For the Kubernetes scheduler, the scheduling framework is a pluggable architecture.

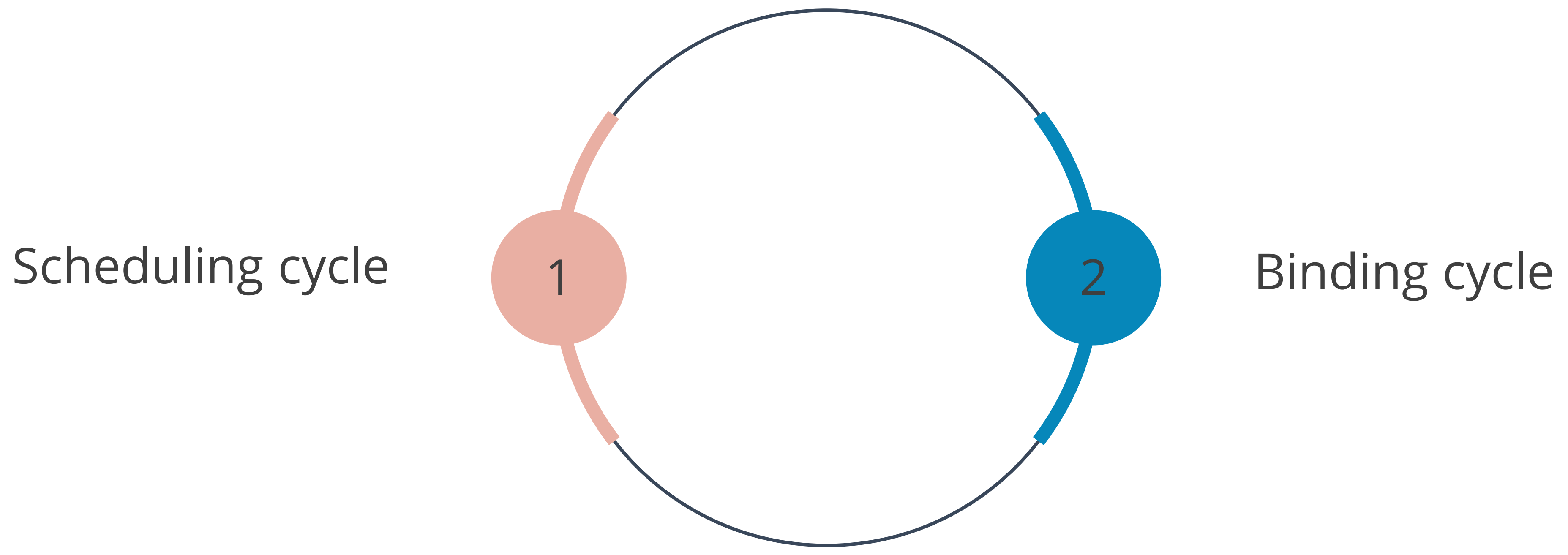
This scheduler adds a set of plugin APIs to the existing scheduler.

Most scheduling features can be implemented by APIs due to the plugins.

Framework Workflow

Extension points are defined by the scheduling framework, and one or more of these invoke the registered scheduler plug-ins.

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.
- Binding cycles can be run concurrently. Scheduling cycles are run serially.
- If there is an internal error or a Pod is nonschedulable, the cycles may be aborted.

Extension Points

This is the scheduling context of a Pod. The extension points exposed by the scheduling framework are shown below:

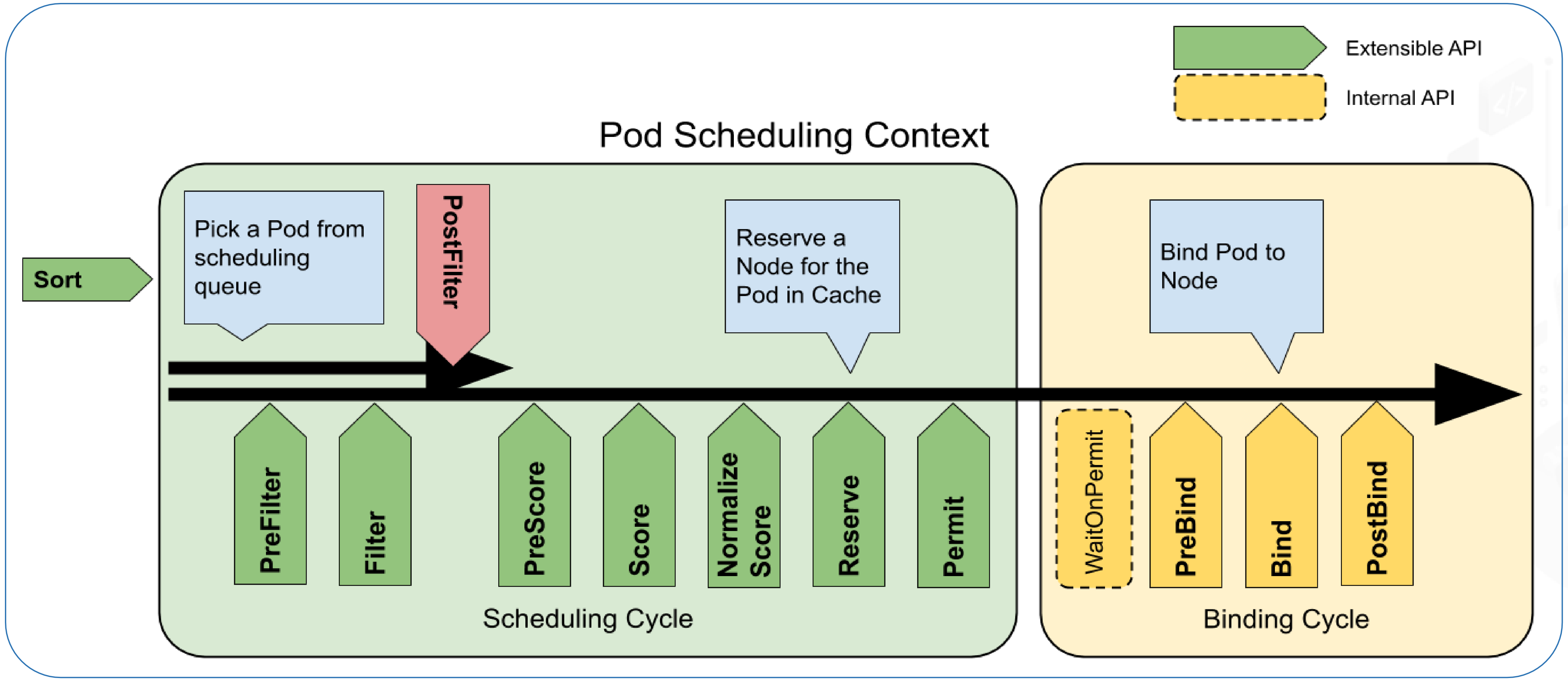


Image Source - https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

Plug-ins for Scheduling Cycle and Binding Cycle

QueueSort

This sorts the Pod in the scheduling queue. Essentially, this plug-in will give a "less(Pod1, Pod2)" function.

PreFilter

This preprocesses information about the Pod or checks particular conditions that the Pod or cluster must meet.

Filter

This filters out those Nodes that cannot run the Pod. For the rest of the Nodes, the remaining plugins will not be called.

PostFilter

These plugins are called after the filter phase when no viable Nodes are found for the Pod.

Plug-ins for Scheduling Cycle and Binding Cycle

PreScore

This generates a shareable state for Score plug-ins 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.

NormalizeScore

This plug-in ranks Nodes based on the number of blinking lights they contain:

Demo

```
 func Scorenode(_ *v1.pod, n *v1.node) (int, error) {
    return getBlinkingLightCount(n)
}

#However, the maximum count of blinking lights may be small
compared to #nodescoreMax. To fix this, BlinkingLightScorer should
also register for this #extension point.

func NormalizeScores(scores map[string]int) {
    highest : = 0
    for _, score := range scores {
        highest = max(highest, score)
    }
    for Node, score := range scores {
        scores[node] = score*nodescoreMax/highest
    }
}
```

Plug-ins for Scheduling Cycle and Binding Cycle

Reserve

Implements reserve extension; uses reserve and unreserve methods

Permit

Is invoked at the end of the scheduling cycle for each Pod in order to prevent or delay the binding to the candidate Node

PreBind

Performs any work needed before a Pod is bound

Plug-ins for Scheduling Cycle and Binding Cycle

PostBind

Is an extension point (informational) and is called after a Pod is successfully bound

Bind

Is called only after PreBind plug-ins have completed their part; binds a Pod to a Node

Plug-in API

To use extension points, plug-ins must first be registered and configured. The extension point interfaces must have the following syntax:

Demo

```
 type Plugin interface {
    Name() string
}

type QueueSortPlugin interface {
    Plugin
    Less(*v1.pod, *v1.pod) bool
}

type PreFilterPlugin interface {
    Plugin
    PreFilter(context.Context, *framework.CycleState,
*v1.pod) error
}

// ...
```

Plug-in Configuration

Plug-ins can be disabled or enabled in the scheduler configuration.

In Kubernetes v1.18 or later:

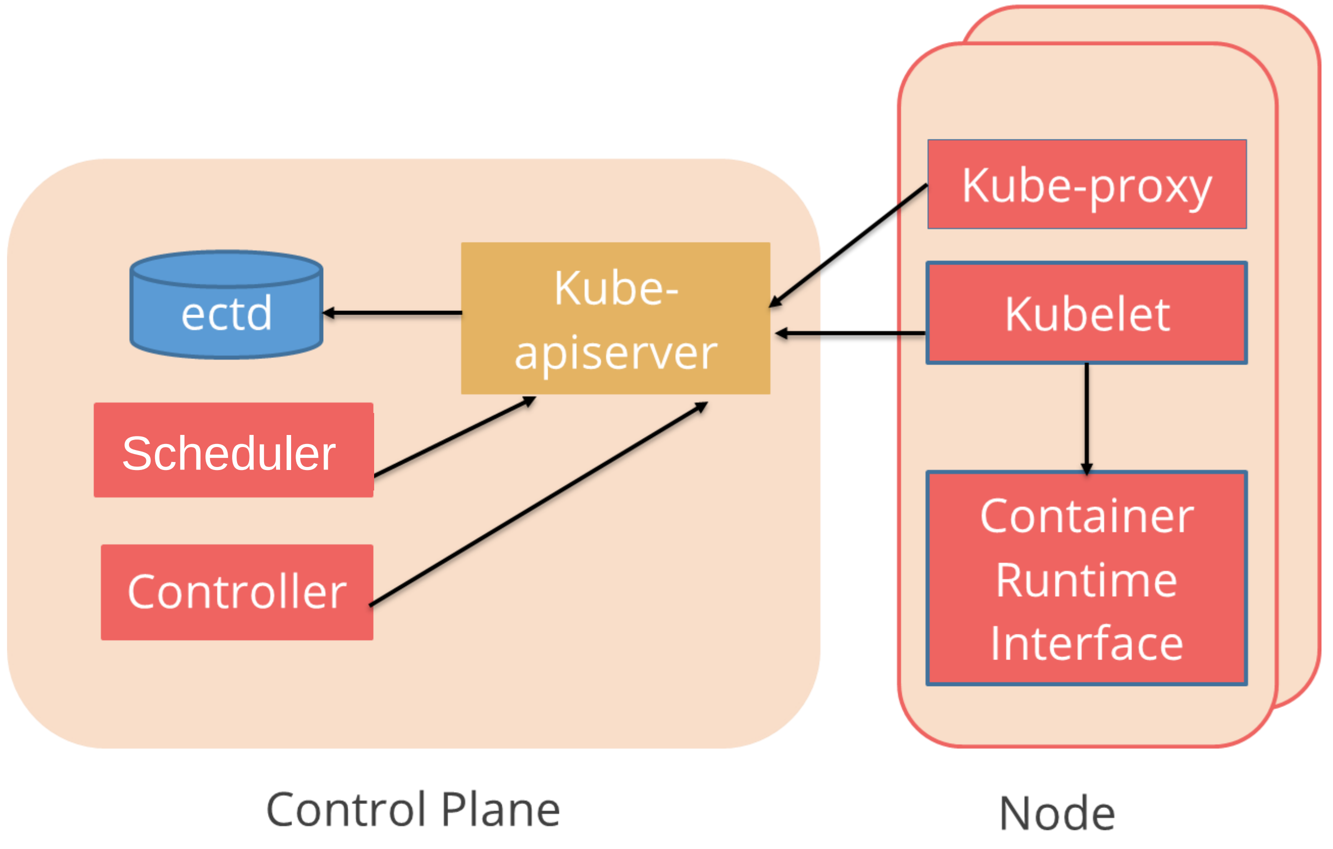
- ✓ Most scheduling plug-ins are utilized and enabled by default
- ✓ New scheduling plug-ins can be configured and implemented along with default plug-ins
- ✓ A set of plug-ins can be configured as a scheduler profile and multiple profiles may be defined to fit different types of workloads

Kube-Scheduler

Kubernetes Scheduler

The Kubernetes scheduler is a control plane process that assigns Pods to Nodes.

High-level view of Kubernetes scheduler:



Configure Kubernetes Scheduler

The behavior of the kube-scheduler can be configured by creating a configuration file and giving its path as a command line option.

In the kube-scheduler, a scheduling profile allows one to customize the different stages of scheduling. An extension point exposes each step. Plug-ins implement one or more of these extension points to offer scheduling behaviors.

The **kube-scheduler --config <filename>** command can be run using the KubeSchedulerConfiguration (v1beta2 or v1beta3) struct to specify scheduling profiles.

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

When new Pods are created, they are added to the queue.

Step 03

The scheduler takes Pods off the queue continually and schedules them.

Working of Kubernetes Scheduler

The scheduler's code is in **scheduler.go.**

There are three important facets that need to be considered while scheduling.

1. The code that watches or waits for Pod creation:

Demo

```
 // Run begins watching and scheduling. It waits for cache to be
synced and then starts a go routine.

func (sched *Scheduler) Run() {
        if !sched.config.WaitForCacheSync() {
                return
        }

        go wait.Until(sched.scheduleOne, 0,
sched.config.StopEverything)
```

Working of Kubernetes Scheduler

2. The code that is responsible for queuing the Pod:

Demo

```
    // Queue for Pod that needs scheduling
    podQueue *cache.FIFO
```

Demo

```
// Event handler triggering

func (f *ConfigFactory) getNextpod() *v1.pod {
for {
       Pod := cache.Pop(f.podQueue).(*v1.pod)
              if f.ResponsibleForpod(pod) {
                     glog.V(4).Infof("About to try and schedule
Pod %v", pod.Name)
                        return Pod
               }
        }
}
```

Working of Kubernetes Scheduler

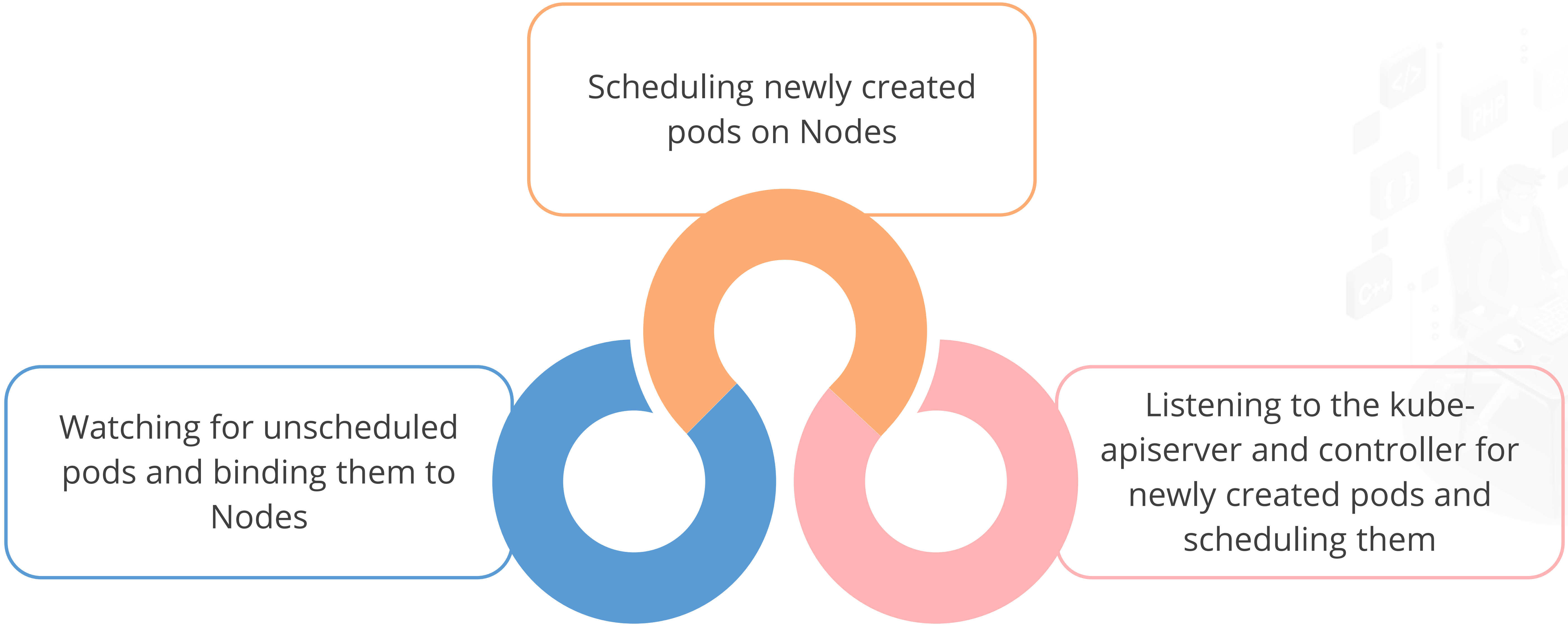
3. The code that handles the errors:

Demo

```
//Error handling
//Scheduled Pod cache
        podInformer.Informer().AddEventHandler(
                cache.FilteringResourceEventHandler{
                        FilterFunc: func(obj interface{}) bool {
                                switch t := obj.(type) {
                                case *v1.pod:
                                        return
assignedNonTerminatedpod(t)
                                default:
                                        runtime.HandleError(fmt.Er
rorf("unable to handle object in %T: %T", c, obj))
                                        return false
                                }
                        },
```

Working of Kubernetes Scheduler

The Kubernetes scheduler is responsible for three tasks.


Scheduling newly created pods on Nodes
Watching for unscheduled pods and binding them to Nodes
Listening to the kube-apiserver and controller for newly created pods and scheduling them

Running Multiple Schedulers

● As the default scheduler runs on a default algorithm, in some specific use cases it might not function accordingly. Hence, the need for a custom scheduler arises.

● Furthermore, multiple schedulers, besides 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.

Running Multiple Schedulers

The three prerequisites that need to be considered before configuring multiple schedulers are:

- A Kubernetes cluster should be created
- The **kubectl** command line tool should be configured to communicate with the cluster.
- A minimum of two nodes that are not the control plane hosts should be created

Running Multiple Schedulers

The steps to run multiple schedulers simultaneously are:

01

- Create a container image for the user’s scheduler binary
- Use the default scheduler (kube-scheduler) as the second scheduler

02

- Create a Kubernetes deployment for the scheduler
- Use a Deployment which monitors a ReplicaSet, that oversees the pods, and makes the scheduler failure resistant

Running Multiple Schedulers

The steps to run multiple schedulers simultaneously are:

03

- Run the second scheduler in the cluster
- Allow the leader election

04

- Create pods and specify the schedulers for each
- Validate if the pods were scheduled with the appropriate schedulers

Kubernetes Scheduler Commands and Options

Some of the nondeprecated options that can be used with **kube-scheduler [flags]** include:

--add_dir_header

Adds the file directory to the header of the log messages, if true

--allow-metric-labels stringToString

The map from metric-label to value allow-list of this label

--alsologtostderr

Log to standard error as well as files

TECHNOLOGY

Node Selection in Kubernetes Scheduler

Node Selection in Kubernetes Scheduler

kube-scheduler selects a Node for the Pod in a two-step operation.

Filtering

kube-scheduler 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**

● **Scheduling profiles**

Understanding Node Selection in Kube-scheduler

Duration: 15 mins

Problem Statement:

You’ve been asked to create a custom Kubernetes scheduler for the Pods.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a custom scheduler
2. Creating a Pod using lab-scheduler

TECHNOLOGY

Perform Rolling Updates on a DaemonSet

DaemonSet Update Strategy

A DaemonSet has two types of update strategies:

OnDelete

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

RollingUpdate

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

Performing a Rolling Update

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

Demo

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

Performing a Rolling Update

Demo

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

Performing a Rolling Update

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

Demo

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

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

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

To check the update strategy, use the following command:

Demo

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

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

Performing a Rolling Update

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

Demo

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

The output will be as shown below:

Demo

```
RollingUpdate

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

Updating a DaemonSet Template

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

Demo

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

Demo

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

Commands

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

Demo

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

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

Demo

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

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

Demo

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

Commands

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

Demo

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

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

Demo

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

Troubleshooting

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

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

Demo

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

Troubleshooting

Broken rollout

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

Clock skew

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

Clean up

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

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

Rollout

Duration: 10 mins

Problem Statement:

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

Assisted Practice: Guidelines

Steps to be followed:

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

TECHNOLOGY

Rollbacks

Performing a Rollback on a DaemonSet

Rolling back a DaemonSet is a three-step process:

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

Demo

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

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

Performing a Rollback on a DaemonSet

The details of a specific revision are given below:

Demo

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

// This returns the details of that revision:

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

Performing a Rollback on a DaemonSet

Step 2: Roll back to a specific revision

Demo

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

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

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

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

Performing a Rollback on a DaemonSet

Step 3: Watch the progress of the DaemonSet rollback

Demo

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

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

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

Demo

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

DaemonSet Revisions

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

Demo

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

// This returns a list of controller revisions:

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

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

TECHNOLOGY

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.

Run the command shown below to check the health of kube-scheduler:

Demo

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

Node Scoring Threshold

After it has found enough Nodes, kube-scheduler stops looking for viable Nodes; this will help in improving performance of the scheduling process.

Threshold scoring will be done as shown below:

- 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 on 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%:

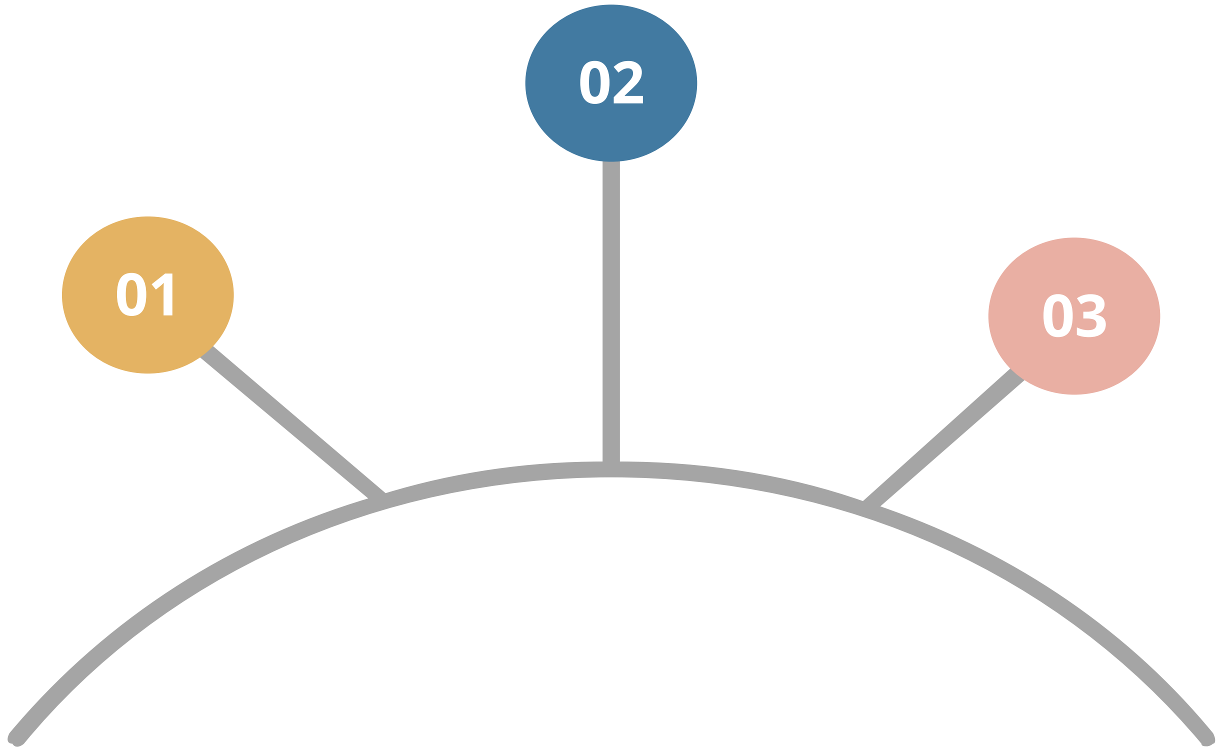
Example

```
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.

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



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

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:

Example

```
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.
```

TECHNOLOGY

Scheduling Policies

Scheduling Policies

A scheduling policy can be used to set the predicates and priorities that the kube-scheduler runs to filter and score Nodes respectively.

Setting a scheduling policy

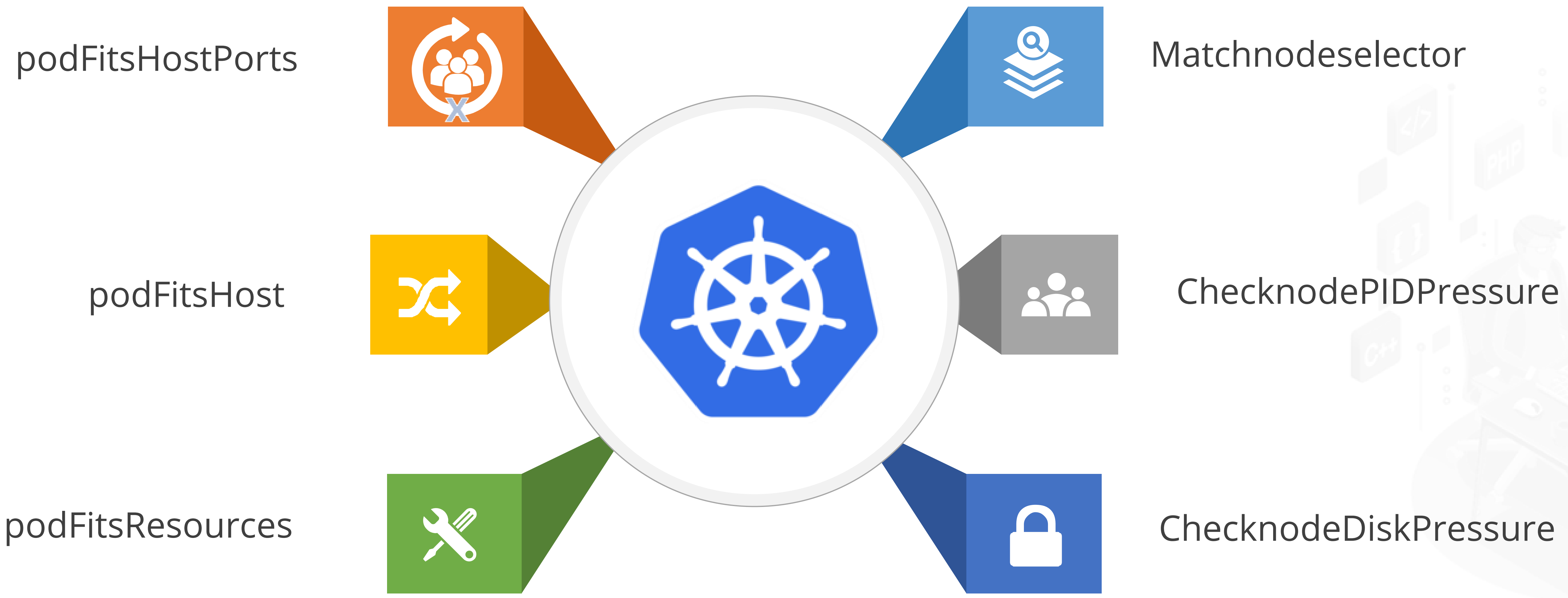
Running: kube-scheduler --policy-config-file <filename> or kube-scheduler --policy-configmap <ConfigMap>
and
Using: Policy type

Note

The scheduling policy is not supported since Kubernetes version 1.23.

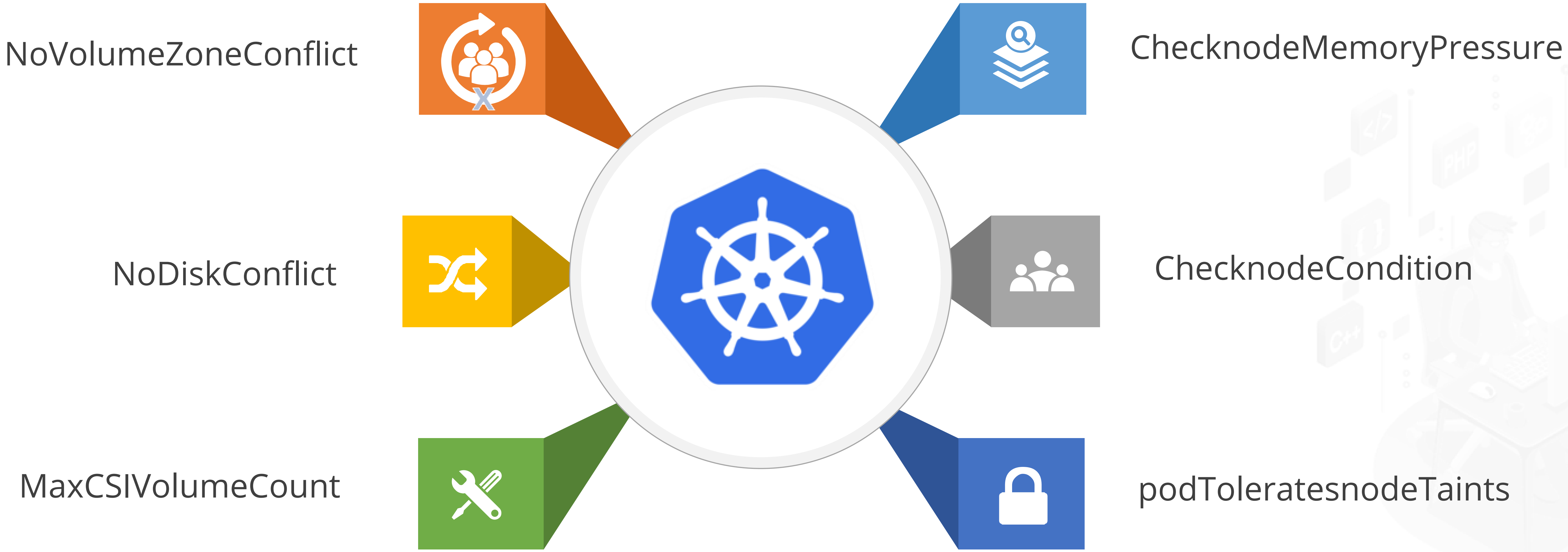
Predicates

The predicates that implement filtering are:



Predicates

The predicates that implement filtering are:



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

TECHNOLOGY

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.
- Plug-ins provide scheduling behaviors via the implementation of one or more extension points.

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

Extension Points

Scheduling happens in stages. These are exposed through the following extension points:

- QueueSort
- PreFilter
- Filter
- PreScore
- Score
- Reserve
- Permit
- PreBind
- Bind
- PostBind

Extension Points

For each extension point, a user may disable specific default plug-ins or enable a specific plug-in.

An example of this is given below:

Example

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:
        disabled:
        - name: nodeResourcesLeastAllocated
        enabled:
        - name: MyCustomPluginA
          weight: 2
        - name: MyCustomPluginB
          weight: 1
```

Scheduling Plug-ins

The following plug-ins, which implement one or more extension points, are enabled by default:

SelectorSpread	podTopologySpread	nodePreferAvoidpod
ImageLocality	nodeName	nodeAffinity
TaintToleration	nodePorts	nodeResourcesBalancedAllocation
points	nodeUnschedulable	nodeResourcesFit

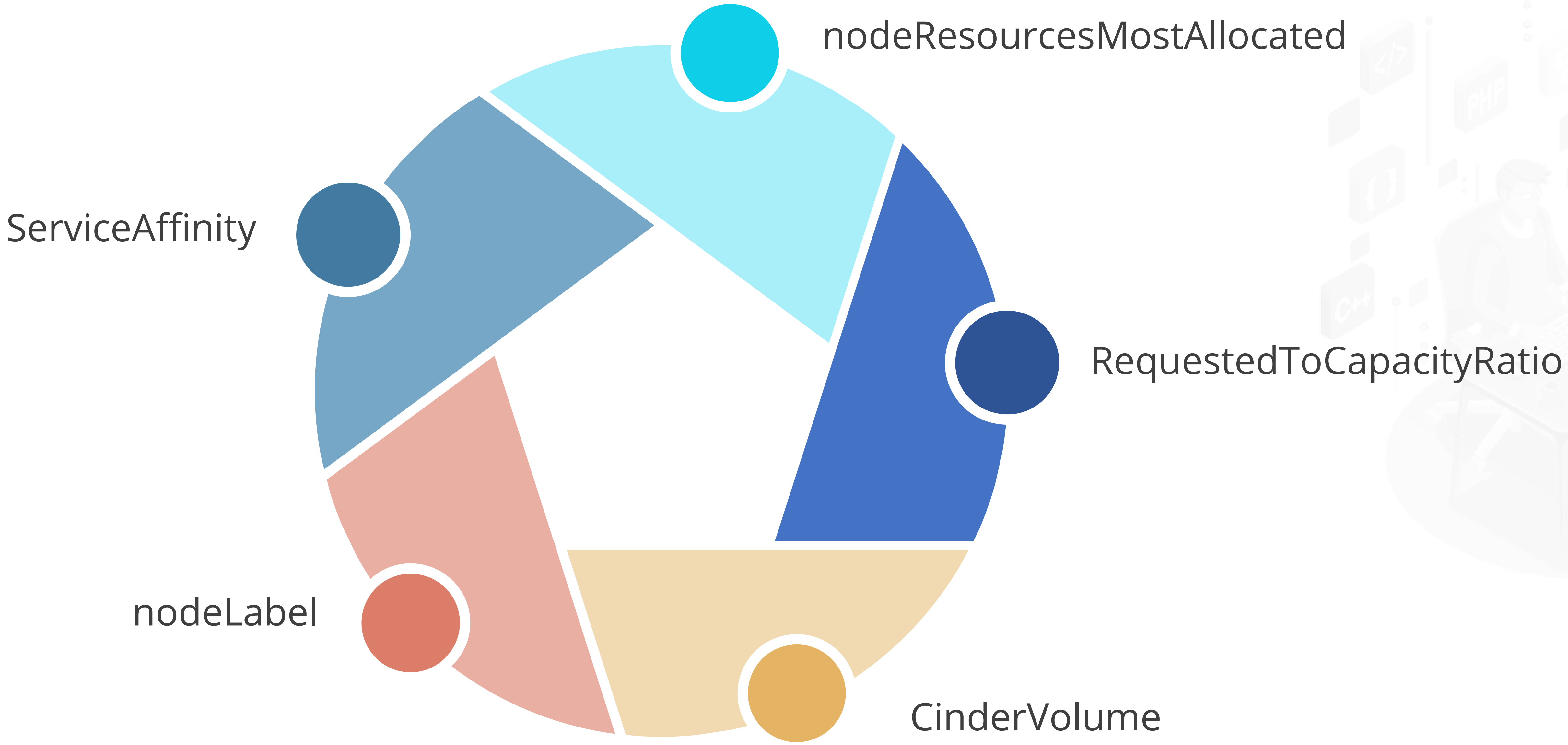
Scheduling Plug-ins

More plug-ins which are enabled by default are:

nodeResourcesLeastAllocated	EBSLimits	DefaultBinder
nodeVolumeLimits	PrioritySort	VolumeZone
InterpodAffinity	VolumeRestrictions	AzureDiskLimits
VolumeBinding	GCEPDLimits	NDefaultPreemptionocation

Scheduling Plug-ins

Some plug-ins are not enabled by default and will need to be enabled through the component config APIs.



Multiple Profiles

With a configuration like the one shown below, the scheduler will run with two profiles, one with default plug-ins enabled and the other with all scoring plug-ins disabled.

Demo

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

Note

A Pod that must be scheduled according to a specific profile must include the corresponding scheduler name under its .spec.schedulerName.

TECHNOLOGY

Topology Management Policies

simplilearn

Topology Manager: Overview

- An increasing number of systems are leveraging a combination of CPUs and hardware accelerators to support latency-critical execution and high-throughput parallel computation.
- Optimizations in CPU isolation, device locality, and memory are required to bring out the best performance in scheduling.

Topology Manager is a Kubelet component that coordinates the set of components responsible for optimizations.

How Topology Manager Works

The Topology Manager acts as a source of truth, supporting other Kubelet components, in making Topology-aligned choices for resource allocation.

The Topology Manager provides an interface for components, which are called Hint Providers, to send and receive topology information.

It receives topology information from the Hint Providers as a bitmask denoting the available NUMA Nodes and an indication of preferred allocation.

Scopes and Policies

The Topology Manager aligns:

- Pods of all QoS classes
- Requested resources that the Hint Provider provides topology hints for

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

Pod Interactions with Policies

In the specs shown below, the Pod runs in the Burstable QoS class. This is due to the requests being less than the limits.

Review the Containers in the following Pod specs:

Demo

```
spec:
  containers:
  - name: nginx
    image: nginx

This Pod runs in the BestEffort QoS class because no
resource requests or limits are specified.

spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

Pod Interactions with Policies

If the selected policy is anything other than *none*, the Topology Manager will pick up the Pod specifications as shown below:

Demo

```
  spec:
containers:
- name: nginx
  image: nginx
  resources:
    limits:
      memory: "200Mi"
      cpu: "2"
      example.com/device: "1"
    requests:
      memory: "200Mi"
      cpu: "2"
      example.com/device: "1"
```

Pod Interactions with Policies

The Pod with integral CPU requests runs in the Guaranteed QoS class as the limits and requests are equal.

Demo

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
```

Pod Interactions with Policies

This Pod runs in the Guaranteed QoS class by sharing CPU request. This is because the requests are equal to the limits.

Demo

```
  spec:
containers:
- name: nginx
  image: nginx
  resources:
    limits:
      example.com/deviceA: "1"
      example.com/deviceB: "1"
    requests:
      example.com/deviceA: "1"
      example.com/deviceB: "1"
```

This Pod runs in the BestEffort QoS class because there are no CPU and memory requests.

Known Limitations

1

The maximum number of NUMA Nodes allowed by the Topology Manager is eight.

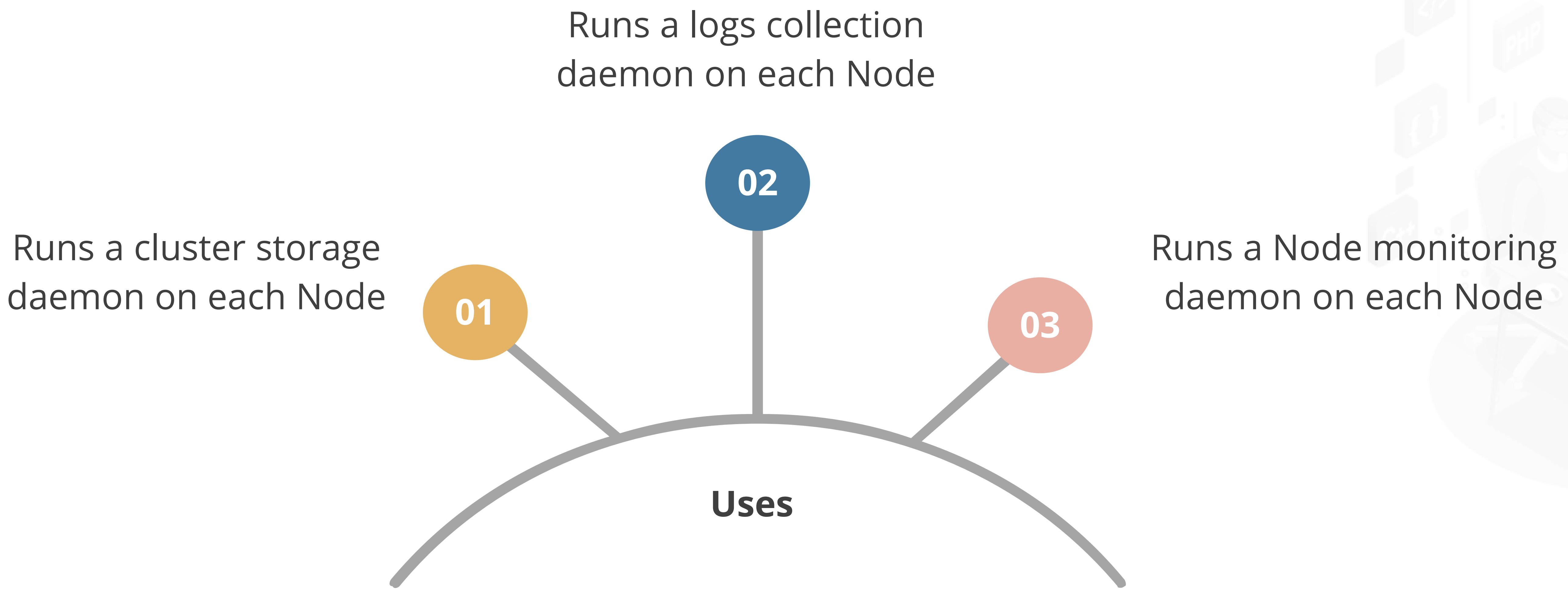
2

The scheduler is not topology aware, and it is scheduled on a Node. However, its failure is dependent on the Topology Manager.

DaemonSet

DaemonSet: Introduction

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



Runs a logs collection daemon on each Node

Runs a cluster storage daemon on each Node

Runs a Node monitoring daemon on each Node

Writing a DaemonSet Spec

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

Demo

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

Writing a DaemonSet Spec

Demo

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

Demo

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

Required Fields

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

Note

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

Pod Template

The **.spec.template** is a Pod template.

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

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

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

Pod Selector

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

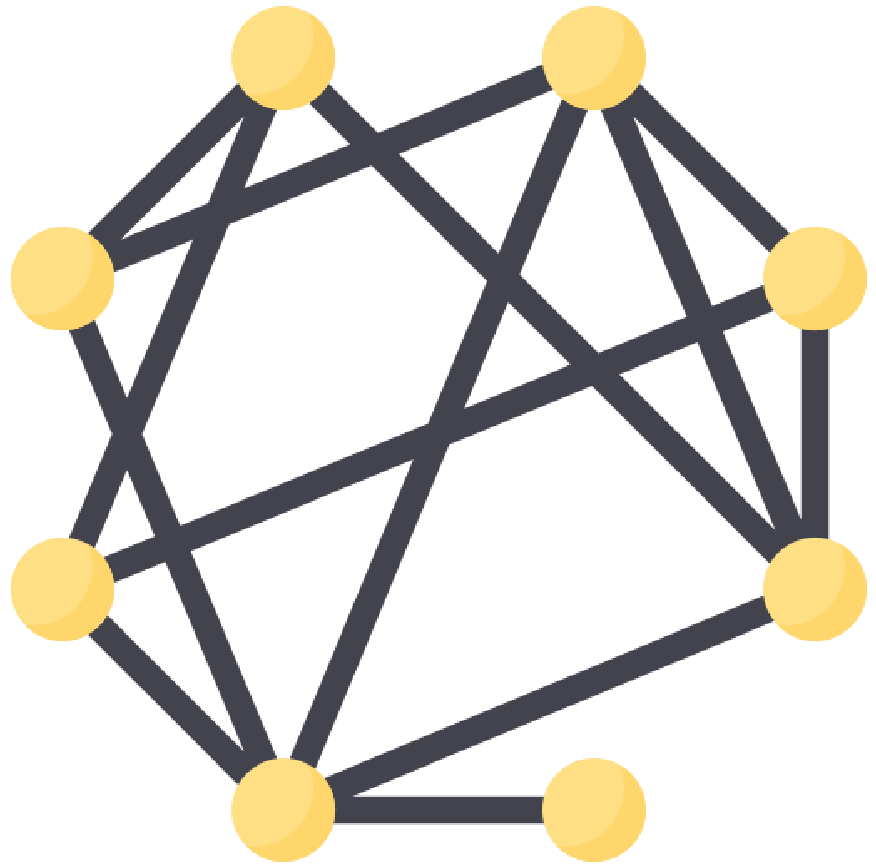
1 matchLabels

2 matchExpressions

The result is ANDed when both the fields are specified.

Running Pods on Select Nodes

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



How Daemon Pods Are Scheduled

The DaemonSet Controller creates and schedules a DaemonSet.

It results in:

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

Running Pods on Select Nodes

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

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

Demo

```
nodeAffinity:

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

Taints and Tolerations

The following toleration keys are added to DaemonSet Pods automatically depending on the related features:

Toleration key	Effect	Version	Description
node.kubernetes.io/not-ready	NoExecute	1.13+	DaemonSet Pods will not be evicted when there are Node-related problems, such as a network partition.
node.kubernetes.io/unreachable	NoExecute	1.13+	DaemonSet Pods will not be evicted when there are Node-related problems, such as a network partition.

Taints and Tolerations

Toleration key	Effect	Version	Description
node.kubernetes.io/disk-pressure	NoSchedule	1.8+	DaemonSet Pods tolerate disk-pressure attributes provided by the default scheduler.
node.kubernetes.io/memory-pressure	NoSchedule	1.8+	DaemonSet Pods tolerate memory-pressure attributes set by the default scheduler.
node.kubernetes.io/unschedulable	NoSchedule	1.12+	DaemonSet Pods tolerate unschedulable attributes provided by the default scheduler.
node.kubernetes.io/network-unavailable	NoSchedule	1.12+	DaemonSet Pods, which use host network, tolerate network-unavailable attributes set by the default scheduler.

Communicating with the Daemon Pod

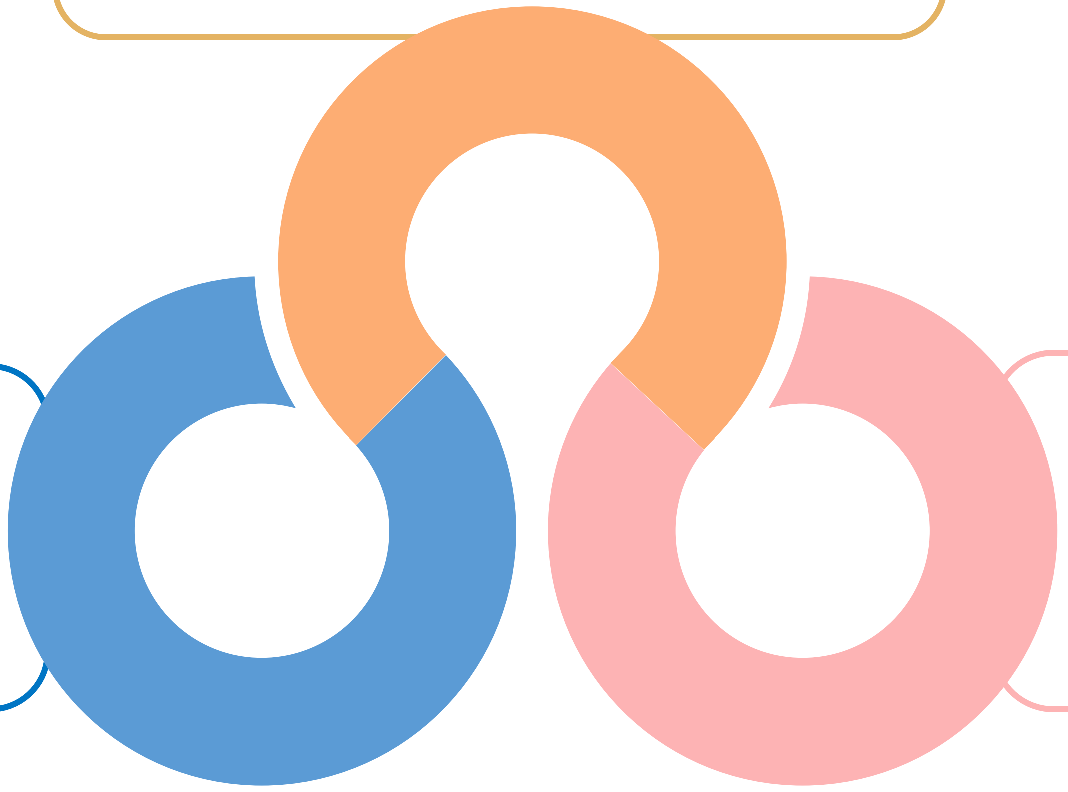
Here are some patterns for communicating with a Pod in a DaemonSet:

- Push
- DNS
- NodeIP and known port
- Service

Updating a DaemonSet

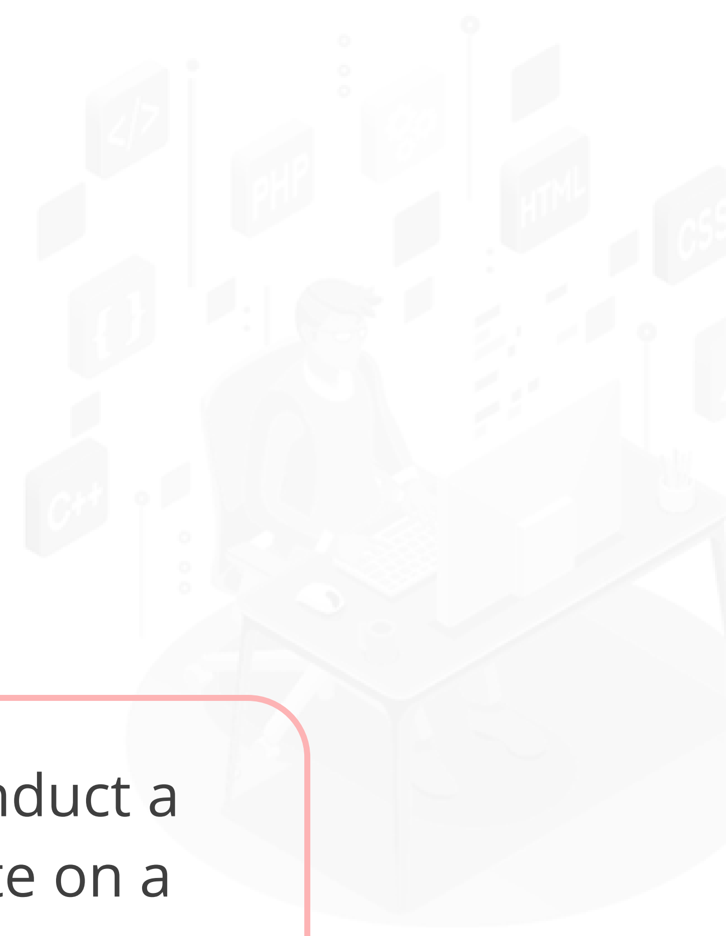
The DaemonSet will instantly add Pods to newly matching Nodes if the Node labels are changed. It will also delete Pods from newly not-matching Nodes.

Users can modify the Pod that a DaemonSet creates.



Users can delete a DaemonSet.

Users can conduct a Rolling Update on a DaemonSet.



Alternatives to DaemonSet

There are four alternatives to a DaemonSet:

- Init scripts
- Bare Pods
- Static Pods
- Deployments

Understanding the DaemonSet

Duration: 10 mins

Problem Statement:

You’ve been assigned a task to create and understand the DaemonSet.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating the DaemonSet
2. Verifying the DaemonSet

Understanding the Security Context

Duration: 10 mins

Problem Statement:

You’ve been assigned a task to set the security context for a Container.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating the security context
2. Verifying the security context
3. Getting a shell into the running Container

TECHNOLOGY

Pod Overhead

Introduction

Pod overhead accounts for resources consumed by the Pod infrastructure.

(beyond Container requests and limits)

The Pod overhead is set at admission time
(based on the overhead associated with a Pod's RuntimeClass).

When enabled, it is considered along with the total of all Container resource requests made when scheduling a Pod.

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.

It uses around 120MiB per Pod for the VM and the guest OS.

Demo

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

Using Pod Overhead

Workloads that set the kata-fc RuntimeClass handler factor will take the CPU and memory overheads while calculating resource quotas, scheduling Nodes, as well as sizing Pod cgroup.

Here is an example workload test-pod:

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  runtimeClassName: kata-fc
  containers:
  - name: busybox-ctr
    image: busybox
    stdin: true
    tty: true
    resources:
```

Example

```
limits:
        cpu: 500m
        memory: 100Mi
  - name: nginx-ctr
    image: nginx
    resources:
      limits:
        cpu: 1500m
        memory: 100MiB
```

The workload's PodSpec is updated by the RuntimeClass admission controller to include the overhead as described in the RuntimeClass.

Using Pod Overhead

After the RuntimeClass admission controller, the user can check the updated PodSpec.

Demo

```
kubectl get Pod test-pod
 -o jsonpath='{.spec.overhead}'


The output is:

map[cpu:250m memory:120Mi]
```

If a ResourceQuota is defined, the sum of container requests and overhead field is calculated.

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 check this against what is being observed by the Node, use:

Example

```
kubectl describe Node | grep test-pod -B2

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
 ---------                    ----                ------------  ----------   ---------------  -------------  ---
 default                      test-pod            2250m (56%)   2250m (56%)  320Mi (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.

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

Here’s how crictl is used:

Demo

```
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

Here’s how you can use crictl:

Demo

```
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
```

Observability

A kube_pod_overhead metric aids in corroborating if the Pod overhead is being utilized. It also helps to observe the stability of workloads that run with a defined overhead.

Observability is not available in the 1.9 release of kube-state-metrics.

TECHNOLOGY

Performance Tuning

Introduction

Performance tuning maximizes infrastructure utilization and reduces costs instead of merely scaling up in reaction to the demands of the environment.

Follow these ten steps to tune the performance of your application for best 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 keep the app small
- Have health and readiness check endpoints

Define Resource Profile to Match Application Requirements

Requests must be defined to support Kubernetes as it efficiently schedules fine-tuned images on appropriate Nodes. Limits must also be set on the memory, CPU, and other resources.

For instance, to use a Go-based microservice as an email server, designate the resource profile given below:

Demo

```
resources:
   requests:
     memory: 1Gi
     cpu: 250m
   limits:
     memory: 2.5Gi
     cpu: 750m
```

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 Taints and Tolerations

The behavior of taints is perpendicular to that of Affinity rules. Taints provides certain rules to prevent events like the nondeployment of some containers to specific Nodes. The taint option can be applied to a certain Node using kubectl.

Demo

```
$ kubectl taint Nodes backup1=backups-only:NoSchedule
```

An exception for a certain Pod can be provided by including toleration in the PodSpec.

Configure Taints and Tolerations

Schedule backups on tainted Nodes by adding these fields in the PodSpec:

Demo

```
spec:
   tolerations:
     - key: "backup1"
         operator: "Equal"
         value: "backups-only"
         effect: "NoSchedule"
```

Configure Pod Priorities

Kubernetes PriorityClass defines and enforces a certain order.

Here's how a priority class can be created:

Demo

```
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."
```

- A high priority Pod is assigned a higher value number.
- A priorityClassName may be added under the PodSpec priorityClassName: highPriority

Configure Kubernetes Features

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

Here are the features that boost scaling performance:

CPU manager
Pod overhead
Accelerators

Optimize Etcd Cluster

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

Note

Deploying Nodes with solid state disks (SSDs) with low I/O latency and high throughput will optimize database performance.

Deploy Kubernetes Clusters

Deploy Kubernetes clusters in geographical zones closer to end users to ensure low latency.

Note

Kubernetes clusters can be deployed locally.

Feedback from Metrics

Quantitative feedback on system performance helps tune the performance of Kubernetes-managed systems.

Understanding Pod Priority and Preemption

Duration: 10 mins

Problem Statement:

You have been asked to use a Pod with priority and preemption.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a priority class object
2. Describing the created priority classes
3. Creating a Pod priority file
4. Describing the Pod for a verification

TECHNOLOGY

Managing Resources

Organizing Resource Configurations

Simplify management of multiple resources by grouping them in the same file. In a YAML file, this can be separated by **---** as shown here:

Demo

```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
```

Demo

```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

Use this code snippet to create multiple resources:

Demo

```
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/
```

Organizing Resource Configurations

A URL can also be set as a configuration source. This is useful for directly deploying from configuration files checked into GitHub.

Demo

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/website/master/conte
nt/en/examples/application/nginx/nginx-deployment.yaml

deployment.apps/my-nginx created
```

Bulk Operations in kubectl

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

Demo

```
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
```

Bulk Operations in kubectl

kubectl outputs resource names in the same syntax as the input. These chain operations can be executed using $() or xargs.

Demo

```
kubectl get $(kubectl create -f docs/concepts/cluster-
administration/nginx/ -o name | grep service)
kubectl create -f docs/concepts/cluster-administration/nginx/ -o name |
grep service | xargs -i kubectl get {}

NAME           TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)      AGE
my-nginx-svc   LoadBalancer   10.0.0.208    <pending>     80/TCP       0s
```

Bulk Operations in kubectl

If resources are organized across several subdirectories within a particular directory, one can perform recursive operations on subdirectories. One can do this using --recursive or -R alongside --filename, -f flag.

Demo

```
project/k8s/development
├── configmap
│   └── my-configmap.yaml
├── deployment
│   └── my-deployment.yaml
└── pvc
    └── my-pvc.yaml
```

Demo

```
// Creating the resources in this directory using the
following command will lead to an error:
kubectl apply -f project/k8s/development

Error: you must provide one or more resources by
argument or filename (.json|.yaml|.yml|stdin)

// Instead, set  --recursive or -R flag with the --
filename,-f flag as such:
kubectl apply -f project/k8s/development --recursive

configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

Bulk Operations in kubectl

The --recursive flag works with any operation that accepts the --filename,-f flag, such as kubectl {create,get,delete,describe,rollout}.

The --recursive flag also works when multiple -f arguments are provided.

Demo

```
kubectl apply -f project/k8s/namespaces -f
project/k8s/development --recursive

namespace/development created
namespace/staging created
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

Using Labels

A few scenarios that use many labels are shown in the example below.

This example shows a guestbook that requires each tier to be distinguished.

Demo

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

Demo

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

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

kubectl get Pod -lapp=guestbook,role=slave

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

Canary Deployments

Multiple labels may be necessary to identify Deployments of different releases or configurations of the same component. Then, the canary of a new application release is deployed along with the previous release.

For example, a track label can be used to differentiate different releases. The primary, stable release would have a track label with the value stable as shown below:

Demo

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

Canary Deployments

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

Demo

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

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

Demo

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

Updating Labels

The labels are updated whenever there is a need to relabel an existing Pod and other resources before new resources are created. This can be done by using the kubectl label.

If all the NGINX Pods must be labeled, run the command given below:

Demo

```
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 arbitrary, nonidentifying metadata that is retrieved by API clients, such as tools and libraries. This can be accomplished by using kubectl annotate, as shown below:

Demo

```
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
```

Scaling Applications

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

An example of this is shown below:

Example

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

deployment.apps/my-nginx scaled

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

kubectl get Pod -l app=nginx

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

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

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

horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

kubectl Apply

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

Demo

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

kubectl Edit

Resources can be updated using kubectl edit.

An example of this is shown below:

Example

```
kubectl edit deployment/my-nginx

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

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

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

rm /tmp/nginx.yaml
```

Disruptive Updates

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

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

Demo

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

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

Deployment of Flask Application with Redis

Duration: 15 mins

Problem Statement:

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

Assisted Practice: Guidelines

Steps to be followed:

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

Key Takeaways

- Scheduling ensures that Pods are matched to Nodes so that Kubelet can run them.
- Kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.
- A DaemonSet ensures that all Nodes run a copy of Pod. Deleting a DaemonSet will clean up the Pods it created.
- The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of "Plugin" APIs to the existing scheduler.

Rollout Test of Httpd Docker Images in Kubernetes Cluster

Duration: 15 Min

Project agenda: To test different Docker images of Httpd web server using the rollout process.

Description:
Your organization has the following images:
httpd:2.4, httpd:2.2, httpd:2.0
Verify if your organization's images given above are working using the rollout scenarios.

Steps to perform:
1. Setting up the cluster
2. Creating an httpd Deployment
3. Updating the image version from httpd:2.0 to httpd:2.2
4. Updating the image version from httpd:2.2 to httpd:2.4
5. Verifying the rollout status

LESSON-END PROJECT