priority class & Preemption

priority class is a way to assign a priority value to a Pod, which determines its relative importance compared to other Pods. The priority value can be any integer between 0 and 1000000, with higher values indicating higher priority.
Preemption policies determine whether a higher priority Pod can preempt(evict) a lower priority Pod to be scheduled on a node. There are three preemption policies:

"PreemptLowerPriority" (default):
Pods with this priority class are allowed to preempt lower priority Pods if there are no nodes with available resources to schedule them without preemption

"IfNoOtherPods":
Pods with this priority class are allowed to preempt lower priority Pods only if there are no other Pods in the cluster that can be evicted to make room for the higher priority Pods

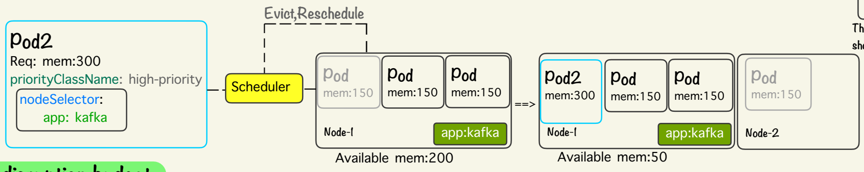
"Never":
Pods with this priority class are never allowed to preempt lower priority Pods

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
preemptionPolicy: PreemptLowerPriority
```

The globalDefault field indicates whether this PriorityClass should be used for pods without a PriorityClass

```
apiVersion: v1
kind: Pod
metadata:
  name: db-1
spec:
  containers:
  - name: db-1
    image: mysql:latest
  priorityClassName: high-priority
```

Once you have defined the priority class, you can assign it to a Pod by specifying its spec.priorityClassName field.



Pod disruption budget

Pod Disruption Budget (PDB) in Kubernetes is a way to ensure that a certain number or percentage of pods with an application are not voluntarily evicted at the same time. This can help to maintain high availability during voluntary disruptions like upgrades and maintenance.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: my-app
```

In this example, the Pod Disruption Budget named my-pdb specifies that at least two pods with the label app=my-app should be available at all times

Bin packing

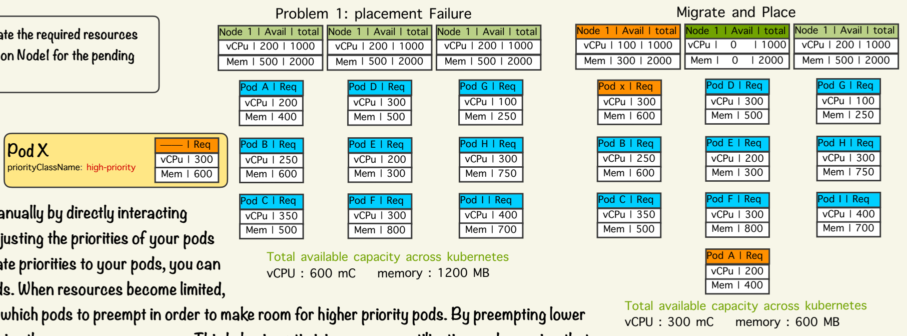
Bin packing in k8s refers to the process of efficiently utilizing resources by scheduling pods on nodes in a way that maximizes resource usage and minimizes wasted resources. Kubernetes achieves bin packing through its scheduler, which considers factors such as resource requests, limits, and available resources on nodes to make optimal scheduling decisions. Kubernetes scheduler follows two strategies to decide the scheduling of Pods:

BestFit: In this approach, the scheduler places the incoming Pod in the node with the least amount of free resources after placement. This strategy aims to leave as much space free as possible on every other node.
WorstFit: In this approach, the scheduler places the incoming Pod in the node with the most amount of free resources after placement. This strategy aims to fill up nodes as much as possible.

Placement failures can occur in bin packing scenarios in Kubernetes when the scheduler is unable to find a suitable node to schedule a pod due to resource constraints or other constraints defined in the cluster.

scenario: there are three nodes in the cluster, each with 1000m of CPU and 2GB RAM. Currently, there are nine running pods (blue) with their allocated resource requests. However, a new pod (orange) with a request of 300m CPU and 600MB RAM cannot be scheduled. This is due to the unavailability of any node that satisfies both the CPU and RAM requirements of the new pod. Surprisingly, even though the entire cluster has a total of 600m CPU and 1200MB RAM available, the scheduler is unable to find a suitable node.

you can consider moving Pod A from Node1 to Node2. By doing so, you would consolidate the required resources (400m CPU, 900MB RAM) on Node1. This would create enough available resources on Node1 for the pending pod X to be comfortably placed by the scheduler.



The operation of moving Pod A from Node1 to Node2 can be performed manually by directly interacting with the Kubernetes API.this can be done using command.Additionally, adjusting the priorities of your pods can help in scenarios where resources are scarce. By assigning appropriate priorities to your pods, you can ensure that critical pods have higher priorities compared to less critical pods. When resources become limited, the Kubernetes scheduler can use these priorities to make decisions about which pods to preempt in order to make room for higher priority pods. By preempting lower priority pods, Kubernetes ensures that critical pods get scheduled and receive the necessary resources. This helps in optimizing resource utilization and ensuring that important workloads are given priority even in resource-constrained environments.

Imbalanced placement in Kubernetes refers to a situation where the distribution of pods or workloads across the nodes in a Kubernetes cluster is uneven or skewed. This can lead to certain nodes being overloaded while others are underutilized, resulting in inefficient resource allocation and potential performance issues. There are a few common causes of imbalanced placement in Kubernetes:

There are a few common causes of imbalanced placement in Kubernetes:

Node labels and pod affinity/anti-affinity: Kubernetes provides mechanisms like node labels and pod affinity/anti-affinity rules to influence the placement of pods. If these rules are not properly configured or if there are inconsistencies in the labels, pods may not be distributed evenly across nodes.

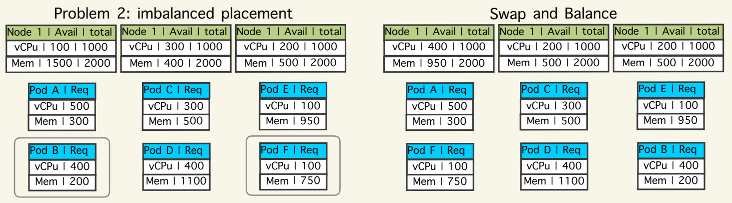
Resource requests and limits: Kubernetes allows you to specify resource requests and limits for pods, indicating the minimum and maximum amount of resources (CPU, memory) they require. If these values are set incorrectly or if there is a wide variation in the resource requirements of pods, it can lead to imbalanced placement.

Node capacity and utilization: If the nodes in a Kubernetes cluster have different capacities in terms of CPU, memory, or other resources, it can result in imbalanced placement. Nodes with higher capacity may end up hosting more pods, while nodes with lower capacity may remain underutilized.

scenario: Node1 has high CPU usage (90%) but relatively low memory usage (25%). On the other hand, Node3 has low CPU usage (20%) but high memory usage (85%). This imbalance in resource utilization across the nodes can have the following impacts:

Pod B on Node1: Since Pod B is a CPU-intensive process, the high CPU usage on Node1 indicates that there might be limited CPU resources available for Pod B during peak load situations. This can result in Pod B experiencing CPU starvation, leading to degraded performance or even failures if it requires more CPU resources than what is available.

Pod E on Node3: As Node3 has high memory usage (85%), Pod E, which is running on Node3, might face memory starvation during peak load scenarios. If Pod E requires additional memory resources that are not available due to high memory usage on Node3, it can lead to out-of-memory errors or performance degradation



If we swap Pod B and Pod F between Node1 and Node3, the observation and impact remain the same. Node1 still has 40% CPU usage and 48% memory usage, while Node3 has 50% CPU usage and 55% memory usage. With these resource utilization levels, any pods on these two nodes should still be able to handle any kind of peak load without experiencing resource starvation or performance degradation

Daemonset

A DaemonSet is a type of controller that ensures that all (or some) nodes in a cluster run a copy of a specific pod. It is often used for system-level tasks that should be run on every node, such as log collection, monitoring, or other types of background tasks

When you create a DaemonSet, Kubernetes automatically creates a pod on each node that matches the specified label selector. If a new node is added to the cluster, Kubernetes automatically creates a new pod on that node as well

By using labels and node selectors, you can specify which nodes in the Kubernetes cluster should run a particular DaemonSet. This allows you to restrict the execution of the DaemonSet to specific nodes

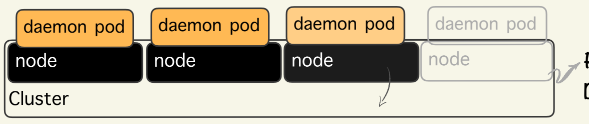
```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-deamon
  namespace: dev-drive-monitor
spec:
  template:
    metadata:
      labels:
        app: monitoring
    spec:
      containers:
        - name: monitoring-agent
          image: monitoring-agent
      nodeSelector:
       Drive: ssd

  selector:
    matchLabels:
      app: monitoring
```

Only on nodes that have this label, a pod of the DaemonSet type is automatically created

The selector section specifies the label selector used to identify which pods are managed by the DaemonSet

what is the best way to test a DaemonSet on a limited number of nodes without consuming too many resources from the customer's service?

One approach could be to create a separate namespace with a ResourceQuota that limits the amount of resources that can be used by the DaemonSet. This will ensure that the DaemonSet does not consume too many resources from the customer's service



Remove a node:
DaemonSet automatically terminate the corresponding pod

Add a new node:
DaemonSet automatically create a new pod on the node

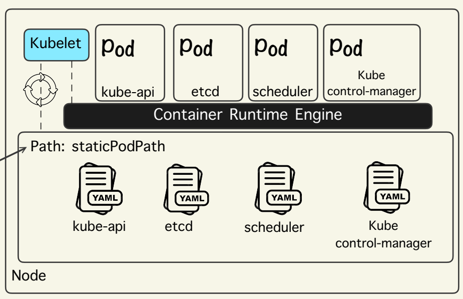
Static Pod

A static pod is a pod that is managed directly by the kubelet on a specific node, rather than by the Kubernetes API server. A static pod is defined by a YAML manifest file that is placed in a specific directory on the node, and the kubelet monitors that directory for changes to the manifest file

The containers come up statically, and their manifest file is located in the directory /etc/kubernetes/manifests. This means that the Kubernetes components, such as the API server, controller manager, and scheduler, are started as containers using pre-defined manifests located in the /etc/kubernetes/manifests directory

StaticPodPath is a directory path where static pod manifests are stored on a node
/var/lib/kubelet/<configuration>.yaml

```
...
staticPodPath: /etc/kubernetes/manifests
```



```
$ kubectl --namespace kube-system get pods -o wide
NAME                                  READY   STATUS    RESTARTS   AGE    IP               NODE
etcd-kubemaster-1                      1/1    Running      4       6d2h   192.168.100.11   kubemaster-1
kube-apiserver-kubemaster-1            1/1    Running      3       6d2h   192.168.100.11   kubemaster-1
kube-controller-manager-kubemaster-1   1/1    Running      3       6d2h   192.168.100.11   kubemaster-1
kube-scheduler-kubemaster-1            1/1    Running      3       6d2h   192.168.100.11   kubemaster-1
```

To delete a static pod in Kubernetes, you can either delete its corresponding manifest file from StaticPodPath or move the manifest file file to another path. Use the following command to remove the manifest file:

```
sudo rm /etc/kubernetes/manifests/<static-pod-manifest.yaml>
```

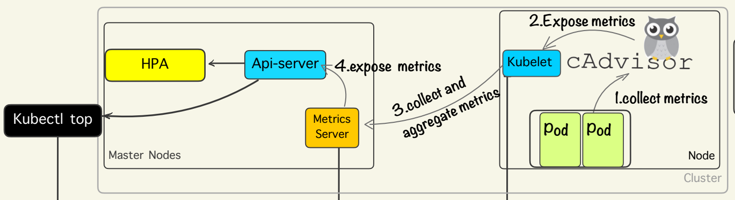
Replace <static-pod-manifest.yaml> with the filename of the manifest associated with the static pod you want to delete.

After performing either of these operations , the kubelet running on the node will detect the change in the static pod directory. It will stop managing the static pod associated with the deleted or moved YAML file, and Kubernetes will initiate the termination process for that pod.

Static PODs	DaemonSets
Created by the kubelet	Created by Kube-API server
Deploy Control Plane components as static pods	Deploy MonitoringAgents, logging Agents on nodes
ignored by the kube-scheduler	

Metrics Server

The Metrics Server is a component of Kubernetes that provides container resource metrics for built-in autoscaling pipelines. It collects resource metrics from Kubelets and exposes them through the Metrics API in the Kubernetes API server. These metrics can be used by the Horizontal Pod Autoscaler and Vertical Pod Autoscaler for autoscaling purposes



cAdvisor (short for "Container Advisor") is a component of the kubelet that is responsible for collecting and monitoring performance metrics for containers and pods running on a node in a Kubernetes cluster

API provided by the kubelet for discovering and retrieving per-node summarized stats available through the /metrics/resource endpoint

The Metrics Server aggregates metrics such as CPU and memory usage and stores in memory

```
kubectl top nodes
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
node1      50m          5%     983Mi           49%
node2      47m          4%     1043Mi          52%
```

```
$ kubectl top pod
NAME                CPU(cores)   MEMORY(bytes)
pod1                0m           10Mi
pod2                1m           100Mi
```