**Chapter 6. Automated Placement**

*Automated Placement* is the core function of the Kubernetes scheduler for assigning new Pods to nodes satisfying container resource requests and honoring scheduling policies. This pattern describes the principles of Kubernetes’ scheduling algorithm and the way to influence the placement decisions from the outside.

**Problem**

A reasonably sized microservices-based system consists of tens or even hundreds of isolated processes. Containers and Pods do provide nice abstractions for packaging and deployment but do not solve the problem of placing these processes on suitable nodes. With a large and ever-growing number of microservices, assigning and placing them individually to nodes is not a manageable activity.

Containers have dependencies among themselves, dependencies to nodes, and resource demands, and all of that changes over time too. The resources available on a cluster also vary over time, through shrinking or extending the cluster, or by having it consumed by already placed containers. The way we place containers impacts the availability, performance, and capacity of the distributed systems as well. All of that makes scheduling containers to nodes a moving target that has to be shot on the move.

**Solution**

In Kubernetes, assigning Pods to nodes is done by the scheduler. It is an area that is highly configurable, still evolving, and changing rapidly as of this writing. In this chapter, we cover the main scheduling control mechanisms, driving forces that affect the placement, why to choose one or the other option, and the resulting consequences. The Kubernetes scheduler is a potent and time-saving tool. It plays a fundamental role in the Kubernetes platform as a whole, but similarly to other Kubernetes components (API Server, Kubelet), it can be run in isolation or not used at all.

At a very high level, the main operation the Kubernetes scheduler performs is to retrieve each newly created Pod definition from the API Server and assign it to a node. It finds a suitable node for every Pod (as long as there is such a node), whether that is for the initial application placement, scaling up, or when moving an application from an unhealthy node to a healthier one. It does this by considering runtime dependencies, resource requirements, and guiding policies for high availability, by spreading Pods horizontally, and also by colocating Pods nearby for performance and low-latency interactions. However, for the scheduler to do its job correctly and allow declarative placement, it needs nodes with available capacity, and containers with declared resource profiles and guiding policies in place. Let’s look at each of these in more detail.

**Available Node Resources**

First of all, the Kubernetes cluster needs to have nodes with enough resource capacity to run new Pods. Every node has capacity available for running Pods, and the scheduler ensures that the sum of the resources requested for a Pod is less than the available allocatable node capacity. Considering a node dedicated only to Kubernetes, its capacity is calculated using the formula in [Example 6-1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-node-resources).

*Example 6-1. Node capacity*

*Allocatable* [capacity for application pods] =

*Node Capacity* [available capacity on a node]

- *Kube-Reserved* [Kubernetes daemons like kubelet, container runtime]

- *System-Reserved* [OS system daemons like sshd, udev]

If you don’t reserve resources for system daemons that power the OS and Kubernetes itself, the Pods can be scheduled up to the full capacity of the node, which may cause Pods and system daemons to compete for resources, leading to resource starvation issues on the node. Also keep in mind that if containers are running on a node that is not managed by Kubernetes, reflected in the node capacity calculations by Kubernetes.

A workaround for this limitation is to run a placeholder Pod that doesn’t do anything, but has only resource requests for CPU and memory corresponding to the untracked containers’ resource use amount. Such a Pod is created only to represent and reserve the resource consumption of the untracked containers and helps the scheduler build a better resource model of the node.

**Container Resource Demands**

Another important requirement for an efficient Pod placement is that containers have their runtime dependencies and resource demands defined. We covered that in more detail in [Chapter 2, *Predictable Demands*](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch02.html#PredictableDemands). It boils down to having containers that declare their resource profiles (with request and limit) and environment dependencies such as storage or ports. Only then are Pods sensibly assigned to nodes and can run without affecting each other during peak times.

**Placement Policies**

The last piece of the puzzle is having the right filtering or priority policies for your specific application needs. The scheduler has a default set of predicate and priority policies configured that is good enough for most use cases. It can be overridden during scheduler startup with a different set of policies, as shown in [Example 6-2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-scheduler-policy).

**CAUTION**

Scheduler policies and custom schedulers can be defined only by an administrator as part of the cluster configuration. As a regular user you just can refer to predefined schedulers.

*Example 6-2. An example scheduler policy*

{

**"kind"** : "Policy",

**"apiVersion"** : "v1",

**"predicates"** : [ [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO1-1)

{**"name"** : "PodFitsHostPorts"},

{**"name"** : "PodFitsResources"},

{**"name"** : "NoDiskConflict"},

{**"name"** : "NoVolumeZoneConflict"},

{**"name"** : "MatchNodeSelector"},

{**"name"** : "HostName"}

],

**"priorities"** : [ [2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO1-2)

{**"name"** : "LeastRequestedPriority", **"weight"** : 2},

{**"name"** : "BalancedResourceAllocation", **"weight"** : 1},

{**"name"** : "ServiceSpreadingPriority", **"weight"** : 2},

{**"name"** : "EqualPriority", **"weight"** : 1}

]

}

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO1-1)

Predicates are rules that filter out unqualified nodes. For example, *PodFitsHostsPorts* schedules Pods to request certain fixed host ports only on those nodes that have this port still available.

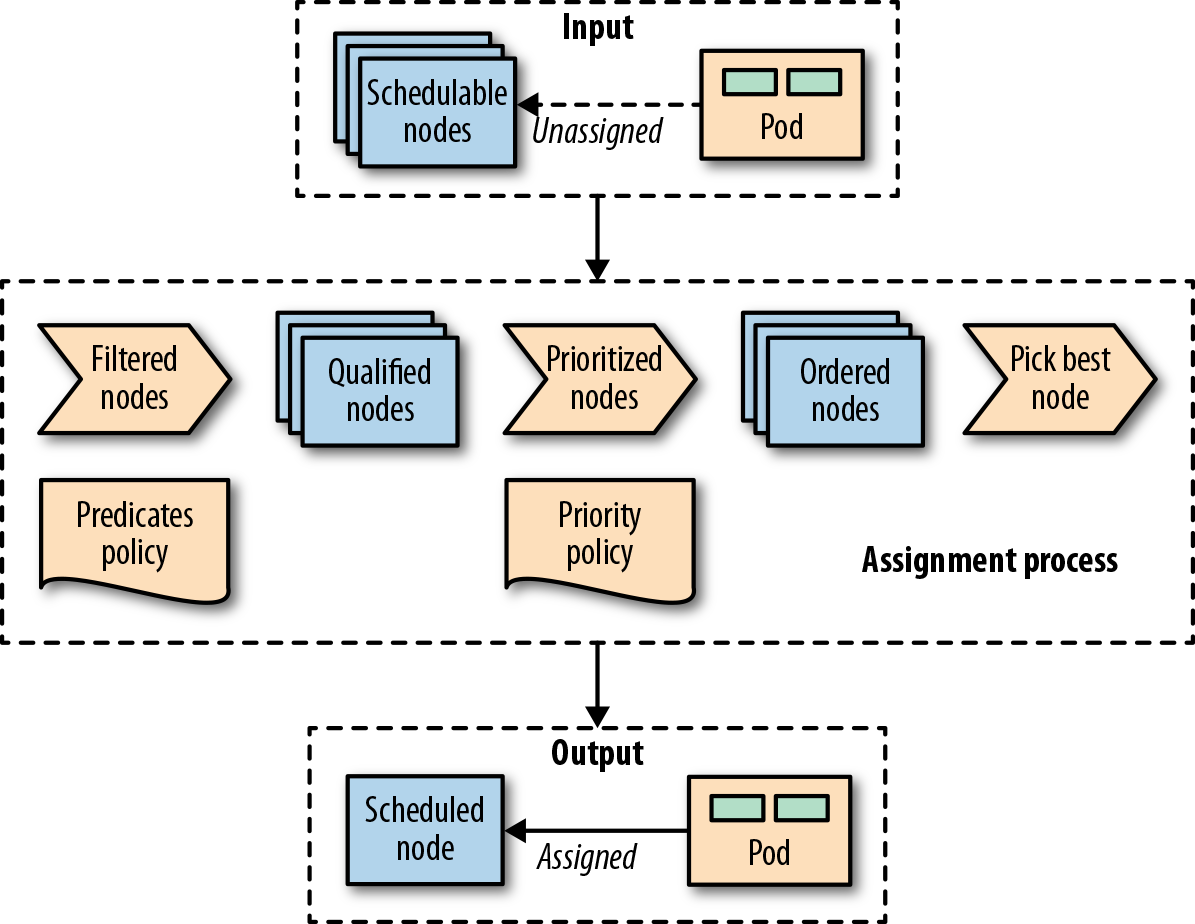
[2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO1-2)

Priorities are rules that sort available nodes according to preferences. For example, *LeastRequestedPriority*gives nodes with fewer requested resources a higher priority.

Consider that in addition to configuring the policies of the default scheduler, it is also possible to run multiple schedulers and allow Pods to specify which scheduler to place them. You can start another scheduler instance that is configured differently by giving it a unique name. Then when defining a Pod, just add the field .spec.schedulerName with the name of your custom scheduler to the Pod specification and the Pod will be picked up by the custom scheduler only.

**Scheduling Process**

Pods get assigned to nodes with certain capacities based on placement policies. For completeness, [Figure 6-1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#img-scheduler) visualizes at a high level how these elements get together and the main steps a Pod goes through when being scheduled.



***Figure 6-1. A Pod-to-node assignment process***

As soon as a Pod is created that is not assigned to a node yet, it gets picked by the scheduler together with all the available nodes and the set of filtering and priority policies. In the first stage, the scheduler applies the filtering policies and removes all nodes that do not qualify based on the Pod’s criteria. In the second stage, the remaining nodes get ordered by weight. In the last stage the Pod gets a node assigned, which is the primary outcome of the scheduling process.

In most cases, it is better to let the scheduler do the Pod-to-node assignment and not micromanage the placement logic. However, on some occasions, you may want to force the assignment of a Pod to a specific node or a group of nodes. This assignment can be done using a node selector. .spec.nodeSelector is Pod field and specifies a map of key-value pairs that must be present as labels on the node for the node to be eligible to run the Pod. For example, say you want to force a Pod to run on a specific node where you have SSD storage or GPU acceleration hardware. With the Pod definition in [Example 6-3](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-node-selector) that has nodeSelector matching disktype: ssd, only nodes that are labeled with disktype=ssd will be eligible to run the Pod.

*Example 6-3. Node selector based on type of disk available*

apiVersion: v1

kind: Pod

metadata:

name: random-generator

spec:

containers:

- image: k8spatterns/random-generator:1.0

name: random-generator

nodeSelector:

disktype: ssd [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO2-1)

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO2-1)

Set of node labels a node must match to be considered to be the node of this Pod

In addition to specifying custom labels to your nodes, you can use some of the default labels that are present on every node. Every node has a unique kubernetes.io/hostname label that can be used to place a Pod on a node by its hostname. Other default labels that indicate the OS, architecture, and instance-type can be useful for placement too.

**Node Affinity**

Kubernetes supports many more flexible ways to configure the scheduling processes. One such a feature is node affinity, which is a generalization of the node selector approach described previously that allows specifying rules as either required or preferred. Required rules must be met for a Pod to be scheduled to a node, whereas preferred rules only imply preference by increasing the weight for the matching nodes without making them mandatory. Besides, the node affinity feature greatly expands the types of constraints you can express by making the language more expressive with operators such as In, NotIn, Exists, DoesNotExist, Gt, or Lt. [Example 6-4](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-node-affinity)demonstrates how node affinity is declared.

*Example 6-4. Pod with node affinity*

apiVersion: v1

kind: Pod

metadata:

name: random-generator

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution: [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO3-1)

nodeSelectorTerms:

- matchExpressions: [2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO3-2)

- key: numberCores

operator: Gt

values: [ "3" ]

preferredDuringSchedulingIgnoredDuringExecution: [3](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO3-3)

- weight: 1

preference:

matchFields: [4](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO3-4)

- key: metadata.name

operator: NotIn

values: [ "master" ]

containers:

- image: k8spatterns/random-generator:1.0

name: random-generator

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO3-1)

Hard requirement that the node must have more than three cores (indicated by a node label) to be considered in the scheduling process. The rule is not reevaluated during execution if the conditions on the node change.

[2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO3-2)

Match on labels.

[3](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO3-3)

Soft requirements, which is a list of selectors with weights. For every node, the sum of all weights for matching selectors is calculated, and the highest-valued node is chosen, as long as it matches the hard requirement.

[4](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO3-4)

Match on a field (specified as jsonpath). Note that only In and NotIn are allowed as operators, and only one value is allowed to be given in the list of values.

**Pod Affinity and Antiaffinity**

Node affinity is a more powerful way of scheduling and should be preferred when nodeSelector is not enough. This mechanism allows constraining which nodes a Pod can run based on label or field matching. It doesn’t allow expressing dependencies among Pods to dictate where a Pod should be placed relative to other Pods. To express how Pods should be spread to achieve high availability, or be packed and co-located together to improve latency, Pod affinity and antiaffinity can be used.

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at multiple topology levels. Using the topologyKey field, and the matching labels, it is possible to enforce more fine-grained rules, which combine rules on domains like node, rack, cloud provider zone, and region, as demonstrated in [Example 6-5](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-pod-affinity).

*Example 6-5. Pod with Pod affinity*

apiVersion: v1

kind: Pod

metadata:

name: random-generator

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution: [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO4-1)

- labelSelector: [2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO4-2)

matchLabels:

confidential: high

topologyKey: security-zone [3](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO4-3)

podAntiAffinity: [4](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO4-4)

preferredDuringSchedulingIgnoredDuringExecution: [5](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO4-5)

- weight: 100

podAffinityTerm:

labelSelector:

matchLabels:

confidential: none

topologyKey: kubernetes.io/hostname

containers:

- image: k8spatterns/random-generator:1.0

name: random-generator

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO4-1)

Required rules for the Pod placement concerning other Pods running on the target node.

[2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO4-2)

Label selector to find the Pods to be colocated with.

[3](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO4-3)

The nodes on which Pods with labels confidential=high are running are supposed to carry a label security-zone. The Pod defined here is scheduled to a node with the same label and value.

[4](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO4-4)

Antiaffinity rules to find nodes where a Pod would *not* be placed.

[5](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO4-5)

Rule describing that the Pod should not (but could) be placed on any node where a Pod with the label confidential=none is running.

Similar to node affinity, there are hard and soft requirements for Pod affinity and antiaffinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution, respectively. Again, as with node affinity, there is the IgnoredDuringExecution suffix in the field name, which exists for future extensibility reasons. At the moment, if the labels on the node change and affinity rules are no longer valid, the Pods continue running,[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#idm46299074601848) but in the future runtime changes may also be taken into account.

**Taints and Tolerations**

A more advanced feature that controls where Pods can be scheduled and are allowed to run is based on taints and tolerations. While node affinity is a property of Pods that allows them to choose nodes, taints and tolerations are the opposite. They allow the nodes to control which Pods should or should not be scheduled on them. A *taint* is a characteristic of the node, and when it is present, it prevents Pods from scheduling onto the node unless the Pod has toleration for the taint. In that sense, taints and tolerations can be considered as an *opt-in* to allow scheduling on nodes, which by default are not available for scheduling, whereas affinity rules are an *opt-out* by explicitly selecting on which nodes to run and thus exclude all the nonselected nodes.

A taint is added to a node by using kubectl: kubectl taint nodes master node-role.kubernetes.io/master="true":NoSchedule, which has the effect shown in [Example 6-6](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-taint). A matching toleration is added to a Pod as shown in [Example 6-7](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-toleration). Notice that the values for key and effect in the taints section of [Example 6-6](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-taint) and the tolerations: section in [Example 6-7](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#ex-toleration) have the same values.

*Example 6-6. Tainted node*

apiVersion: v1

kind: Node

metadata:

name: master

spec:

taints: [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO5-1)

- effect: NoSchedule

key: node-role.kubernetes.io/master

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO5-1)

Taint on a node’s spec to mark this node as not available for scheduling except when a Pod tolerates this taint

*Example 6-7. Pod tolerating node taints*

apiVersion: v1

kind: Pod

metadata:

name: random-generator

spec:

containers:

- image: k8spatterns/random-generator:1.0

name: random-generator

tolerations:

- key: node-role.kubernetes.io/master [1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO6-1)

operator: Exists

effect: NoSchedule [2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#callout_automated_placement_CO6-2)

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO6-1)

Tolerate (i.e., consider for scheduling) nodes, which have a taint with key node-role.kubernetes.io/master. On production clusters, this taint is set on the master node to prevent scheduling of Pods on the master. A toleration like this allows this Pod to be installed on the master nevertheless.

[2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#co_automated_placement_CO6-2)

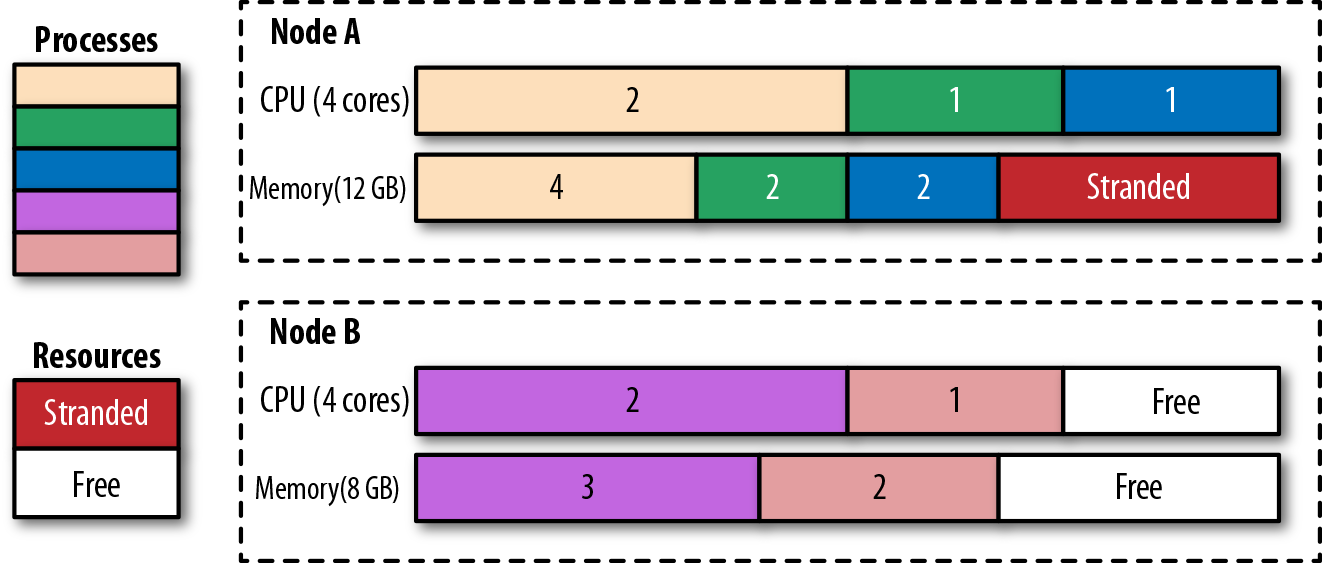
Tolerate only when the taint specifies a NoSchedule effect. This field can be empty here, in which case the toleration applies to every effect.

There are hard taints that prevent scheduling on a node (effect=NoSchedule), soft taints that try to avoid scheduling on a node (effect=PreferNoSchedule), and taints that can evict already running Pods from a node (effect=NoExecute).

Taints and tolerations allow for complex use cases like having dedicated nodes for an exclusive set of Pods, or force eviction of Pods from problematic nodes by tainting those nodes.

You can influence the placement based on the application’s high availability and performance needs, but try not to limit the scheduler much and back yourself into a corner where no more Pods can be scheduled, and there are too many stranded resources. For example, if your containers’ resource requirements are too coarse-grained, or nodes are too small, you may end up with stranded resources in nodes that are not utilized.

In [Figure 6-2](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#img-resources), we can see node A has 4 GB of memory that cannot be utilized as there is no CPU left to place other containers. Creating containers with smaller resource requirements may help improve this situation. Another solution is to use the Kubernetes *descheduler*, which helps defragment nodes and improve their utilization.



***Figure 6-2. Processes scheduled to nodes and stranded resources***

Once a Pod is assigned to a node, the job of the scheduler is done, and it does not change the placement of the Pod unless the Pod is deleted and recreated without a node assignment. As you have seen, with time, this can lead to resource fragmentation and poor utilization of cluster resources. Another potential issue is that the scheduler decisions are based on its cluster view at the point in time when a new Pod is scheduled. If a cluster is dynamic and the resource profile of the nodes changes or new nodes are added, the scheduler will not rectify its previous Pod placements. Apart from changing the node capacity, you may also alter the labels on the nodes that affect placement, but past placements are not rectified either.

All these are scenarios that can be addressed by the descheduler. The Kubernetes descheduler is an optional feature that typically is run as a Job whenever a cluster administrator decides it is a good time to tidy up and defragment a cluster by rescheduling the Pods. The descheduler comes with some predefined policies that can be enabled and tuned or disabled. The policies are passed as a file to the descheduler Pod, and currently, they are the following:

RemoveDuplicates

This strategy ensures that only a single Pod associated with a ReplicaSet or Deployment is running on a single node. If there are more Pods than one, these excess Pods are evicted. This strategy is useful in scenarios where a node has become unhealthy, and the managing controllers started new Pods on other healthy nodes. When the unhealthy node is recovered and joins the cluster, the number of running Pods is more than desired, and the descheduler can help bring the numbers back to the desired replicas count. Removing duplicates on nodes can also help with the spread of Pods evenly on more nodes when scheduling policies and cluster topology have changed after the initial placement.

LowNodeUtilization

This strategy finds nodes that are underutilized and evicts Pods from other over-utilized nodes, hoping these Pods will be placed on the underutilized nodes, leading to better spread and use of resources. The underutilized nodes are identified as nodes with CPU, memory, or Pod count below the configured thresholds values. Similarly, overutilized nodes are those with values greater than the configured targetThresholdsvalues. Any node between these values is appropriately utilized and not affected by this strategy.

RemovePodsViolatingInterPodAntiAffinity

This strategy evicts Pods violating interpod antiaffinity rules, which could happen when the antiaffinity rules are added after the Pods have been placed on the nodes.

RemovePodsViolatingNodeAffinity

This strategy is for evicting Pods violating node affinity rules.

Regardless of the policy used, the descheduler avoids evicting the following:

* Critical Pods that are marked with scheduler.alpha.kubernetes.io/critical-podannotation.
* Pods not managed by a ReplicaSet, Deployment, or Job.
* Pods managed by a DaemonSet.
* Pods that have local storage.
* Pods with PodDisruptionBudget where eviction would violate its rules.
* Deschedule Pod itself (achieved by marking itself as a critical Pod).

Of course, all evictions respect Pods’ QoS levels by choosing *Best-Efforts* Pods first, then *Burstable* Pods, and finally *Guaranteed* Pods as candidates for eviction. See [Chapter 2, *Predictable Demands*](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch02.html#PredictableDemands) for a detailed explanation of these QoS levels.

**Discussion**

Placement is an area where you want to have as minimal intervention as possible. If you follow the guidelines from [Chapter 2, *Predictable Demands*](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch02.html#PredictableDemands) and declare all the resource needs of a container, the scheduler will do its job and place the Pod on the most suitable node possible. However, when that is not enough, there are multiple ways to steer the scheduler toward the desired deployment topology. To sum up, from simpler to more complex, the following approaches control Pod scheduling (keep in mind, as of this writing, this list changes with every other release of Kubernetes):

nodeName

The simplest form of hardwiring a Pod to a node. This field should ideally be populated by the scheduler, which is driven by policies rather than manual node assignment. Assigning a Pod to a node limits greatly where a Pod can be scheduled. This throws us back in to the pre-Kubernetes era when we explicitly specified the nodes to run our applications.

nodeSelector

Specification of a map of key-value pairs. For the Pod to be eligible to run on a node, the Pod must have the indicated key-value pairs as the label on the node. Having put some meaningful labels on the Pod and the node (which you should do anyway), a node selector is one of the simplest acceptable mechanisms for controlling the scheduler choices.

Default scheduling alteration

The default scheduler is responsible for the placement of new Pods onto nodes within the cluster, and it does it reasonably. However, it is possible to alter the filtering and priority policies list, order, and weight of this scheduler if necessary.

Pod affinity and antiaffinity

These rules allow a Pod to express dependencies on other Pods. For example, for an application’s latency requirements, high availability, security constraints, and so forth.

Node affinity

This rule allows a Pod to express dependency toward nodes. For example, considering nodes’ hardware, location, and so forth.

Taints and tolerations

Taints and tolerations allow the node to control which Pods should or should not be scheduled on them. For example, to dedicate a node for a group of Pods, or even evict Pods at runtime. Another advantage of Taints and Tolerations is that if you expand the Kubernetes cluster by adding new nodes with new labels, you don’t need to add the new labels on all the Pods, but only on the Pods that should be placed on the new nodes.

Custom scheduler

If none of the preceding approaches is good enough, or maybe you have complex scheduling requirements, you can also write your custom scheduler. A custom scheduler can run instead of, or alongside, the standard Kubernetes scheduler. A hybrid approach is to have a “scheduler extender” process that the standard Kubernetes scheduler calls out to as a final pass when making scheduling decisions. This way you don’t have to implement a full scheduler, but only provide HTTP APIs to filter and prioritize nodes. The advantage of having your scheduler is that you can consider factors outside of the Kubernetes cluster like hardware cost, network latency, and better utilization while assigning Pods to nodes. You can also use multiple custom schedulers alongside the default scheduler and configure which scheduler to use for each Pod. Each scheduler could have a different set of policies dedicated to a subset of the Pods.

As you can see, there are lots of ways to control the Pod placement and choosing the right approach or combining multiple approaches can be challenging. The takeaway from this chapter is this: size and declare container resource profiles, label Pods and nodes accordingly, and finally, do only a minimal intervention to the Kubernetes scheduler.

**More Information**

* [Automated Placement Example](http://bit.ly/2TTJUMh)
* [Assigning Pods to Nodes](https://kubernetes.io/docs/user-guide/node-selection/)
* [Node Placement and Scheduling Explained](https://red.ht/2TP1ceB)
* [Pod Disruption Budget](https://kubernetes.io/docs/admin/disruptions/)
* [Guaranteed Scheduling for Critical Add-On Pods](https://kubernetes.io/docs/admin/rescheduler/)
* [The Kubernetes Scheduler](http://bit.ly/2Hrq8lJ)
* [Scheduler Algorithm](http://bit.ly/2F9Vfi2)
* [Configuring Multiple Schedulers](http://bit.ly/2HLv5Fk)
* [Descheduler for Kubernetes](http://bit.ly/2YMQzYn)
* [Keep Your Kubernetes Cluster Balanced: The Secret to High Availability](http://bit.ly/2zuecKk)
* [Everything You Ever Wanted to Know About Resource Scheduling, but Were Afraid to Ask](http://bit.ly/2FNkBT9)

[1](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html#idm46299074601848-marker) However, if node labels change and allow for unscheduled Pods to match their node affinity selector, these Pods are scheduled on this node.

* [**Add Highlight**](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html)
* [**Add Note**](https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch06.html)