

Assigning Pods to Nodes

You can constrain a `Pod` so that it is *restricted* to run on particular `node(s)`, or to *prefer* to run on particular nodes. There are several ways to do this and the recommended approaches all use [label selectors](#) to facilitate the selection. Often, you do not need to set any such constraints; the `scheduler` will automatically do a reasonable placement (for example, spreading your Pods across nodes so as not place Pods on a node with insufficient free resources). However, there are some circumstances where you may want to control which node the Pod deploys to, for example, to ensure that a Pod ends up on a node with an SSD attached to it, or to co-locate Pods from two different services that communicate a lot into the same availability zone.

You can use any of the following methods to choose where Kubernetes schedules specific Pods:

- [nodeSelector](#) field matching against [node labels](#)
- [Affinity and anti-affinity](#)
- [nodeName](#) field
- [Pod topology spread constraints](#)

Node labels

Like many other Kubernetes objects, nodes have [labels](#). You can [attach labels manually](#). Kubernetes also populates a [standard set of labels](#) on all nodes in a cluster.

Note: The value of these labels is cloud provider specific and is not guaranteed to be reliable. For example, the value of `kubernetes.io/hostname` may be the same as the node name in some environments and a different value in other environments.

Node isolation/restriction

Adding labels to nodes allows you to target Pods for scheduling on specific nodes or groups of nodes. You can use this functionality to ensure that specific Pods only run on nodes with certain isolation, security, or regulatory properties.

If you use labels for node isolation, choose label keys that the `kubelet` cannot modify. This prevents a compromised node from setting those labels on itself so that the scheduler schedules workloads onto the compromised node.

The [NodeRestriction admission plugin](#) prevents the kubelet from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix.

To make use of that label prefix for node isolation:

1. Ensure you are using the [Node authorizer](#) and have *enabled* the `NodeRestriction` admission plugin.
2. Add labels with the `node-restriction.kubernetes.io/` prefix to your nodes, and use those labels in your [node selectors](#). For example, `example.com.node-restriction.kubernetes.io/fips=true` or `example.com.node-restriction.kubernetes.io/pci-dss=true`.

nodeSelector

`nodeSelector` is the simplest recommended form of node selection constraint. You can add the `nodeSelector` field to your Pod specification and specify the [node labels](#) you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

See [Assign Pods to Nodes](#) for more information.

Affinity and anti-affinity

`nodeSelector` is the simplest way to constrain Pods to nodes with specific labels. Affinity and anti-affinity expands the types of constraints you can define. Some of the benefits of affinity and anti-affinity include:

- The affinity/anti-affinity language is more expressive. `nodeSelector` only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
- You can indicate that a rule is *soft* or *preferred*, so that the scheduler still schedules the Pod even if it can't find a matching node.
- You can constrain a Pod using labels on other Pods running on the node (or other topological domain), instead of just node labels, which allows you to define rules for which Pods can be co-located on a node.

The affinity feature consists of two types of affinity:

- *Node affinity* functions like the `nodeSelector` field but is more expressive and allows you to specify soft rules.
- *Inter-pod affinity/anti-affinity* allows you to constrain Pods against labels on other Pods.

Node affinity

Node affinity is conceptually similar to `nodeSelector`, allowing you to constrain which nodes your Pod can be scheduled on based on node labels. There are two types of node affinity:

- `requiredDuringSchedulingIgnoredDuringExecution`: The scheduler can't schedule the Pod unless the rule is met. This functions like `nodeSelector`, but with a more expressive syntax.
- `preferredDuringSchedulingIgnoredDuringExecution`: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.

Note: In the preceding types, `IgnoredDuringExecution` means that if the node labels change after Kubernetes schedules the Pod, the Pod continues to run.

You can specify node affinities using the `.spec.affinity.nodeAffinity` field in your Pod spec.

For example, consider the following Pod spec:

[pods/pod-with-node-affinity.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: topology.kubernetes.io/zone
                operator: In
                values:
                  - antarctica-east1
                  - antarctica-west1
        preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            preference:
              matchExpressions:
                - key: another-node-label-key
                  operator: In
                  values:
                    - another-node-label-value
  containers:
    - name: with-node-affinity
      image: registry.k8s.io/pause:2.0
```

In this example, the following rules apply:

- The node *must* have a label with the key `topology.kubernetes.io/zone` and the value of that label *must* be either `antarctica-east1` or `antarctica-west1`.
- The node *preferably* has a label with the key `another-node-label-key` and the value `another-node-label-value`.

You can use the `operator` field to specify a logical operator for Kubernetes to use when interpreting the rules. You can use `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt` and `Lt`.

Read [Operators](#) to learn more about how these work.

`NotIn` and `DoesNotExist` allow you to define node anti-affinity behavior. Alternatively, you can use [node taints](#) to repel Pods from specific nodes.

Note:

If you specify both `nodeSelector` and `nodeAffinity`, *both* must be satisfied for the Pod to be scheduled onto a node.

If you specify multiple terms in `nodeSelectorTerms` associated with `nodeAffinity` types, then the Pod can be scheduled onto a node if one of the specified terms can be satisfied (terms are ORed).

If you specify multiple expressions in a single `matchExpressions` field associated with a term in `nodeSelectorTerms`, then the Pod can be scheduled onto a node only if all the expressions are satisfied (expressions are ANDed).

See [Assign Pods to Nodes using Node Affinity](#) for more information.

Node affinity weight

You can specify a `weight` between 1 and 100 for each instance of the `preferredDuringSchedulingIgnoredDuringExecution` affinity type. When the scheduler finds nodes that meet all the other scheduling requirements of the Pod, the scheduler iterates through every preferred rule that the node satisfies and adds the value of the `weight` for that expression to a sum.

The final sum is added to the score of other priority functions for the node. Nodes with the highest total score are prioritized when the scheduler makes a scheduling decision for the Pod.

For example, consider the following Pod spec:

[pods/pod-with-affinity-anti-affinity.yaml](#) 

```

apiVersion: v1
kind: Pod
metadata:
  name: with-affinity-anti-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
            preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                preference:
                  matchExpressions:
                    - key: label-1
                      operator: In
                      values:
                        - key-1
              - weight: 50
                preference:
                  matchExpressions:
                    - key: label-2
                      operator: In
                      values:
                        - key-2
      containers:
        - name: with-node-affinity
          image: registry.k8s.io/pause:2.0

```

If there are two possible nodes that match the `preferredDuringSchedulingIgnoredDuringExecution` rule, one with the `label-1:key-1` label and another with the `label-2:key-2` label, the scheduler considers the `weight` of each node and adds the weight to the other scores for that node, and schedules the Pod onto the node with the highest final score.

Note: If you want Kubernetes to successfully schedule the Pods in this example, you must have existing nodes with the `kubernetes.io/os=linux` label.

Node affinity per scheduling profile

FEATURE STATE: [Kubernetes v1.20](#) [beta]

When configuring multiple [scheduling profiles](#), you can associate a profile with a node affinity, which is useful if a profile only applies to a specific set of nodes. To do so, add an `addedAffinity` to the `args` field of the [NodeAffinity plugin](#) in the [scheduler configuration](#). For example:

```

apiVersion: kubescheduler.config.k8s.io/v1beta3
kind: KubeSchedulerConfiguration

profiles:
- schedulerName: default-scheduler
- schedulerName: foo-scheduler
  pluginConfig:
    - name: NodeAffinity
      args:
        addedAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: scheduler-profile
                operator: In
                values:
              - foo

```

The `addedAffinity` is applied to all Pods that set `.spec.schedulerName` to `foo-scheduler`, in addition to the NodeAffinity specified in the PodSpec. That is, in order to match the Pod, nodes need to satisfy `addedAffinity` and the Pod's `.spec.NodeAffinity`.

Since the `addedAffinity` is not visible to end users, its behavior might be unexpected to them. Use node labels that have a clear correlation to the scheduler profile name.

Note: The DaemonSet controller, which [creates Pods for DaemonSets](#), does not support scheduling profiles. When the DaemonSet controller creates Pods, the default Kubernetes scheduler places those Pods and honors any `nodeAffinity` rules in the DaemonSet controller.

Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your Pods can be scheduled on based on the labels of **Pods** already running on that node, instead of the node labels.

Inter-pod affinity and anti-affinity rules take the form "this Pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more Pods that meet rule Y", where X is a topology domain like node, rack, cloud provider zone or region, or similar and Y is the rule Kubernetes tries to satisfy.

You express these rules (Y) as [label selectors](#) with an optional associated list of namespaces. Pods are namespaced objects in Kubernetes, so Pod labels also implicitly have namespaces. Any label selectors for Pod labels should specify the namespaces in which Kubernetes should look for those labels.

You express the topology domain (X) using a `topologyKey`, which is the key for the node label that the system uses to denote the domain. For examples, see [Well-Known Labels, Annotations and Taints](#).

Note: Inter-pod affinity and anti-affinity require substantial amounts of processing which can slow down scheduling in large clusters significantly. We do not recommend using them in clusters larger than several hundred nodes.

Note: Pod anti-affinity requires nodes to be consistently labeled, in other words, every node in the cluster must have an appropriate label matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

Types of inter-pod affinity and anti-affinity

Similar to [node affinity](#) are two types of Pod affinity and anti-affinity as follows:

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

For example, you could use `requiredDuringSchedulingIgnoredDuringExecution` affinity to tell the scheduler to co-locate Pods of two services in the same cloud provider zone because they communicate with each other a lot. Similarly, you could use `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity to spread Pods from a service across multiple cloud provider

zones.


To use inter-pod affinity, use the `affinity.podAffinity` field in the Pod spec. For inter-pod anti-affinity, use the `affinity.podAntiAffinity` field in the Pod spec.

Scheduling a group of pods with inter-pod affinity to themselves

If the current Pod being scheduled is the first in a series that have affinity to themselves, it is allowed to be scheduled if it passes all other affinity checks. This is determined by verifying that no other pod in the cluster matches the namespace and selector of this pod, that the pod matches its own terms, and the chosen node matches all requested topologies. This ensures that there will not be a deadlock even if all the pods have inter-pod affinity specified.

Pod affinity example

Consider the following Pod spec:

[pods/pod-with-pod-affinity.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: topology.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: registry.k8s.io/pause:2.0
```

This example defines one Pod affinity rule and one Pod anti-affinity rule. The Pod affinity rule uses the "hard" `requiredDuringSchedulingIgnoredDuringExecution`, while the anti-affinity rule uses the "soft" `preferredDuringSchedulingIgnoredDuringExecution`.

The affinity rule specifies that the scheduler is allowed to place the example Pod on a node only if that node belongs to a specific [zone](#) where other Pods have been labeled with `security=S1`. For instance, if we have a cluster with a designated zone, let's call it "Zone V," consisting of nodes labeled with `topology.kubernetes.io/zone=V`, the scheduler can assign the Pod to any node within Zone V, as long as there is at least one Pod within Zone V already labeled with `security=S1`. Conversely, if there are no Pods with `security=S1` labels in Zone V, the scheduler will not assign the example Pod to any node in that zone.

The anti-affinity rule specifies that the scheduler should try to avoid scheduling the Pod on a node if that node belongs to a specific [zone](#) where other Pods have been labeled with `security=S2`. For instance, if we have a cluster with a designated zone, let's call it "Zone R," consisting of nodes labeled with `topology.kubernetes.io/zone=R`, the scheduler should avoid assigning the Pod to any

node within Zone R, as long as there is at least one Pod within Zone R already labeled with `security=S2` . Conversely, the anti-affinity rule does not impact scheduling into Zone R if there are no Pods with `security=S2` labels.

To get yourself more familiar with the examples of Pod affinity and anti-affinity, refer to the [design proposal](#).

You can use the `In` , `NotIn` , `Exists` and `DoesNotExist` values in the `operator` field for Pod affinity and anti-affinity.

Read [Operators](#) to learn more about how these work.

In principle, the `topologyKey` can be any allowed label key with the following exceptions for performance and security reasons:

- For Pod affinity and anti-affinity, an empty `topologyKey` field is not allowed in both `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution` .
- For `requiredDuringSchedulingIgnoredDuringExecution` Pod anti-affinity rules, the admission controller `LimitPodHardAntiAffinityTopology` limits `topologyKey` to `kubernetes.io/hostname` . You can modify or disable the admission controller if you want to allow custom topologies.

In addition to `labelSelector` and `topologyKey` , you can optionally specify a list of namespaces which the `labelSelector` should match against using the `namespaces` field at the same level as `labelSelector` and `topologyKey` . If omitted or empty, `namespaces` defaults to the namespace of the Pod where the affinity/anti-affinity definition appears.

Namespace selector

FEATURE STATE: [Kubernetes v1.24](#) [\[stable\]](#)

You can also select matching namespaces using `namespaceSelector` , which is a label query over the set of namespaces. The affinity term is applied to namespaces selected by both `namespaceSelector` and the `namespaces` field. Note that an empty `namespaceSelector` ({}) matches all namespaces, while a null or empty `namespaces` list and null `namespaceSelector` matches the namespace of the Pod where the rule is defined.

matchLabelKeys

FEATURE STATE: [Kubernetes v1.29](#) [\[alpha\]](#)

Note:

The `matchLabelKeys` field is an alpha-level field and is disabled by default in Kubernetes 1.29. When you want to use it, you have to enable it via the `MatchLabelKeysInPodAffinity` [feature gate](#).

Kubernetes includes an optional `matchLabelKeys` field for Pod affinity or anti-affinity. The field specifies keys for the labels that should match with the incoming Pod's labels, when satisfying the Pod (anti)affinity.

The keys are used to look up values from the pod labels; those key-value labels are combined (using `AND`) with the match restrictions defined using the `labelSelector` field. The combined filtering selects the set of existing pods that will be taken into Pod (anti)affinity calculation.

A common use case is to use `matchLabelKeys` with `pod-template-hash` (set on Pods managed as part of a Deployment, where the value is unique for each revision). Using `pod-template-hash` in `matchLabelKeys` allows you to target the Pods that belong to the same revision as the incoming Pod, so that a rolling upgrade won't break affinity.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: application-server
...
spec:
  template:
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - database
              topologyKey: topology.kubernetes.io/zone
            # Only Pods from a given rollout are taken into consideration when calculating pod affinity.
            # If you update the Deployment, the replacement Pods follow their own affinity rules
            # (if there are any defined in the new Pod template)
            mismatchLabelKeys:
              - pod-template-hash
```

mismatchLabelKeys

FEATURE STATE: [Kubernetes v1.29](#) [alpha]

Note:

The `mismatchLabelKeys` field is an alpha-level field and is disabled by default in Kubernetes 1.29. When you want to use it, you have to enable it via the `MatchLabelKeysInPodAffinity` [feature gate](#).

Kubernetes includes an optional `mismatchLabelKeys` field for Pod affinity or anti-affinity. The field specifies keys for the labels that should **not** match with the incoming Pod's labels, when satisfying the Pod (anti)affinity.

One example use case is to ensure Pods go to the topology domain (node, zone, etc) where only Pods from the same tenant or team are scheduled in. In other words, you want to avoid running Pods from two different tenants on the same topology domain at the same time.


```

apiVersion: v1
kind: Pod
metadata:
  labels:
    # Assume that all relevant Pods have a "tenant" label set
    tenant: tenant-a
  ...
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        # ensure that pods associated with this tenant land on the correct node pool
        - matchLabelKeys:
            - tenant
          topologyKey: node-pool
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        # ensure that pods associated with this tenant can't schedule to nodes used for another tenant
        - mismatchLabelKeys:
            - tenant # whatever the value of the "tenant" label for this Pod, prevent
              # scheduling to nodes in any pool where any Pod from a different
              # tenant is running.
          labelSelector:
            # We have to have the labelSelector which selects only Pods with the tenant label,
            # otherwise this Pod would hate Pods from daemonsets as well, for example,
            # which aren't supposed to have the tenant label.
          matchExpressions:
            - key: tenant
              operator: Exists
          topologyKey: node-pool

```

More practical use-cases

Inter-pod affinity and anti-affinity can be even more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc. These rules allow you to configure that a set of workloads should be co-located in the same defined topology; for example, preferring to place two related Pods onto the same node.

For example: imagine a three-node cluster. You use the cluster to run a web application and also an in-memory cache (such as Redis). For this example, also assume that latency between the web application and the memory cache should be as low as is practical. You could use inter-pod affinity and anti-affinity to co-locate the web servers with the cache as much as possible.

In the following example Deployment for the Redis cache, the replicas get the label `app=store`. The `podAntiAffinity` rule tells the scheduler to avoid placing multiple replicas with the `app=store` label on a single node. This creates each cache in a separate node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: redis-server
          image: redis:3.2-alpine
```

The following example Deployment for the web servers creates replicas with the label `app=web-store` . The Pod affinity rule tells the scheduler to place each replica on a node that has a Pod with the label `app=store` . The Pod anti-affinity rule tells the scheduler never to place multiple `app=web-store` servers on a single node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - web-store
              topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: web-app
          image: nginx:1.16-alpine
```

Creating the two preceding Deployments results in the following cluster layout, where each web server is co-located with a cache, on three separate nodes.

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

The overall effect is that each cache instance is likely to be accessed by a single client that is running on the same node. This approach aims to minimize both skew (imbalanced load) and latency.

You might have other reasons to use Pod anti-affinity. See the [ZooKeeper tutorial](#) for an example of a StatefulSet configured with anti-affinity for high availability, using the same technique as this example.

nodeName

`nodeName` is a more direct form of node selection than affinity or `nodeSelector` . `nodeName` is a field in the Pod spec. If the `nodeName` field is not empty, the scheduler ignores the Pod and the kubelet on the named node tries to place the Pod on that node. Using `nodeName` overrules using `nodeSelector` or affinity and anti-affinity rules.

Some of the limitations of using `nodeName` to select nodes are:

- If the named node does not exist, the Pod will not run, and in some cases may be automatically deleted.

- If the named node does not have the resources to accommodate the Pod, the Pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.

Warning: `nodeName` is intended for use by custom schedulers or advanced use cases where you need to bypass any configured schedulers. Bypassing the schedulers might lead to failed Pods if the assigned Nodes get oversubscribed. You can use [node affinity](#) or the [nodeSelector field](#) to assign a Pod to a specific Node without bypassing the schedulers.

Here is an example of a Pod spec using the `nodeName` field:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

The above Pod will only run on the node `kube-01` .

Pod topology spread constraints

You can use *topology spread constraints* to control how Pods are spread across your cluster among failure-domains such as regions, zones, nodes, or among any other topology domains that you define. You might do this to improve performance, expected availability, or overall utilization.

Read [Pod topology spread constraints](#) to learn more about how these work.

Operators

The following are all the logical operators that you can use in the `operator` field for `nodeAffinity` and `podAffinity` mentioned above.

Operator	Behavior
In	The label value is present in the supplied set of strings
NotIn	The label value is not contained in the supplied set of strings
Exists	A label with this key exists on the object
DoesNotExist	No label with this key exists on the object

The following operators can only be used with `nodeAffinity` .

Operator	Behaviour
Gt	The supplied value will be parsed as an integer, and that integer is less than the integer that results from parsing the value of a label named by this selector
Lt	The supplied value will be parsed as an integer, and that integer is greater than the integer that results from parsing the value of a label named by this selector

Note: `Gt` and `Lt` operators will not work with non-integer values. If the given value doesn't parse as an integer, the pod will fail to get scheduled. Also, `Gt` and `Lt` are not available for `podAffinity`.

What's next

- Read more about [taints and tolerations](#) .
- Read the design docs for [node affinity](#) and for [inter-pod affinity/anti-affinity](#).
- Learn about how the [topology manager](#) takes part in node-level resource allocation decisions.
- Learn how to use [nodeSelector](#).
- Learn how to use [affinity and anti-affinity](#).

Feedback

Was this page helpful?

Yes

No

Last modified March 26, 2024 at 3:26 PM PST: [Upgrade note about nodeName to warning.\(0b15eccf6d\)](#)