Kubernetes

AWS Load Balancer Controller

<https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.6/guide/ingress/annotations/>

Test

[https://uklabs.kodekloud.com/courses/labs-certified-kubernetes-administrator-with-practice tests/](https://uklabs.kodekloud.com/courses/labs-certified-kubernetes-administrator-with-practice%20tests/)

Apply the coupon code **udemystudent151113**

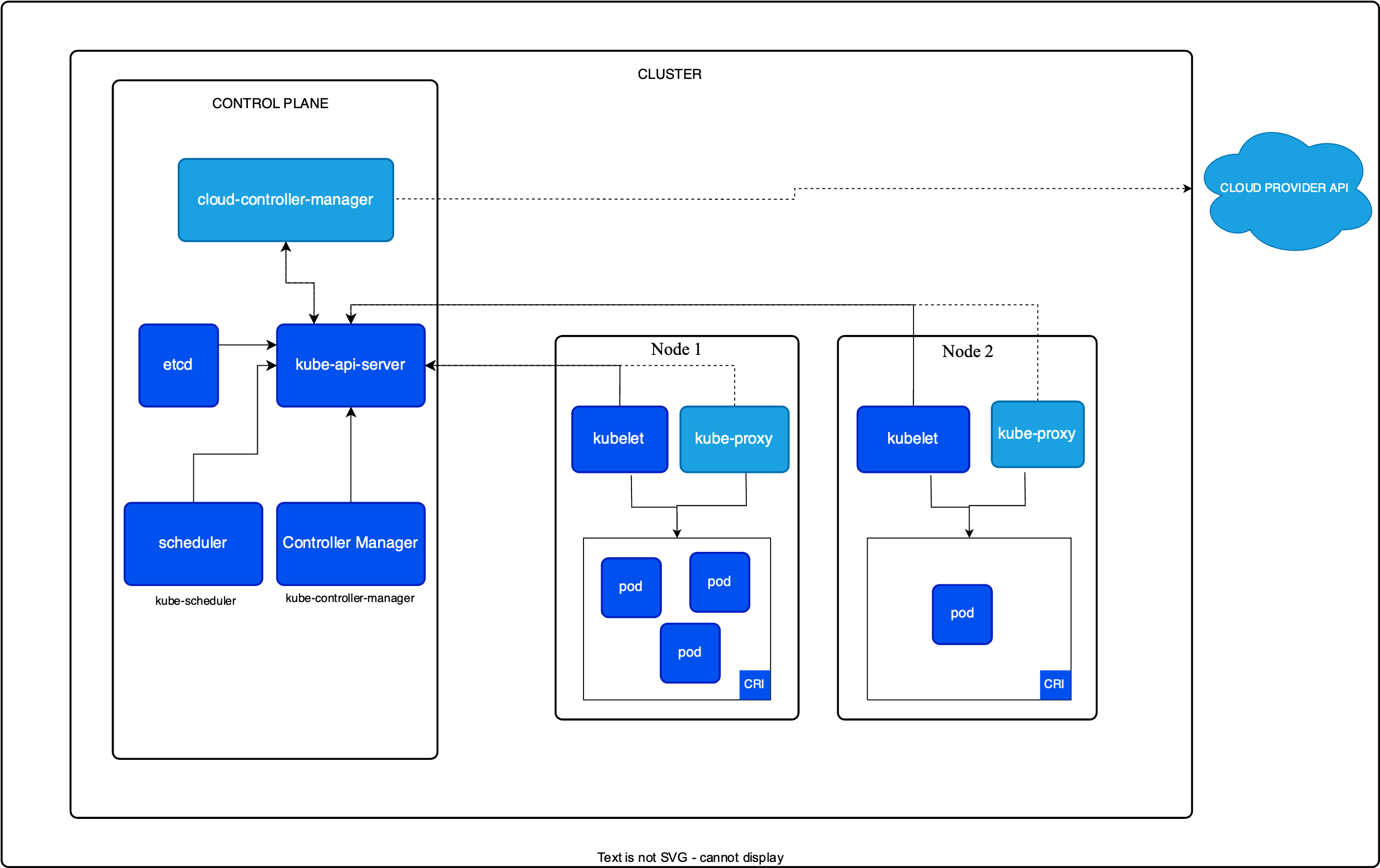
Access EKS From Laptop Need to Create Access Entries with User Ashish

A screenshot of a computer

Description automatically generated



[Cluster Architecture](https://kubernetes.io/docs/concepts/architecture/)

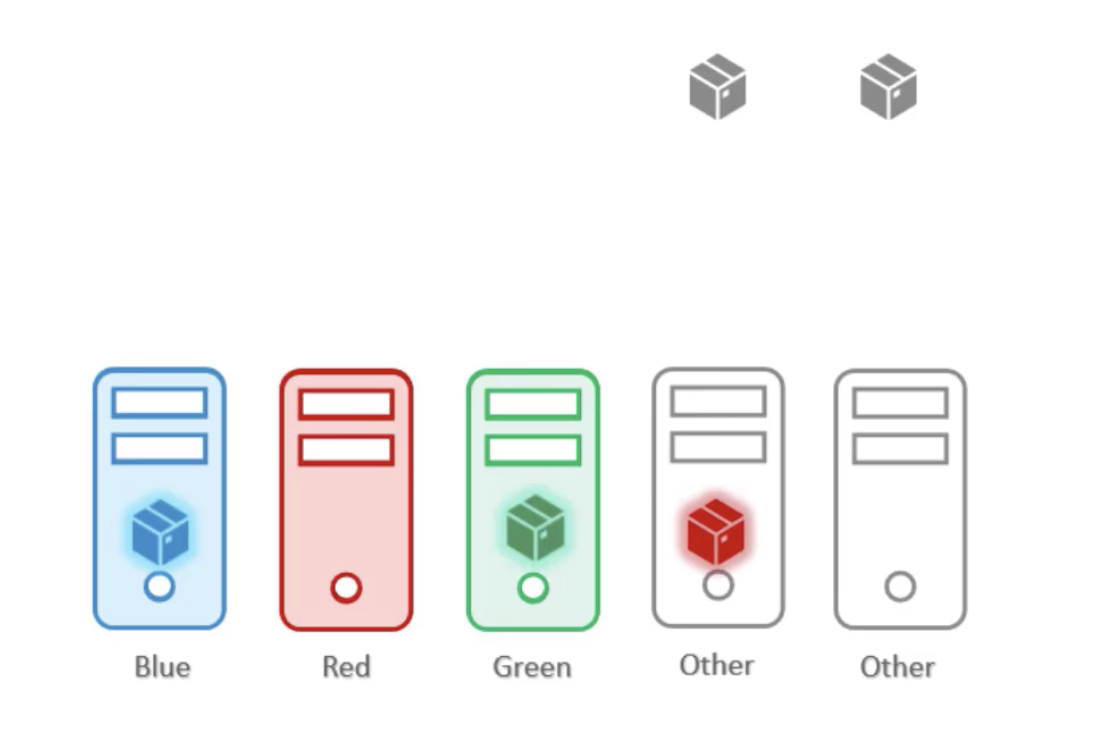


Kubernetes cluster architecture

[Kubernetes Components](https://kubernetes.io/docs/concepts/overview/components/)

* kube-apiserver
* Etcd
* kube-scheduler
* kube-controller-manager
* cloud-controller-manager

[Taints and Tolerations](https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)



[Node affinity](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/" \l "affinity-and-anti-affinity) is a property of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) that attracts them to a set of [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/) (either as a preference or a hard requirement). Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also [evaluates other parameters](https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/) as part of its function.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

## **Concepts**

You add a taint to a node using [kubectl taint](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#taint). For example,

kubectl taint nodes node1 key1=value1:NoSchedule

places a taint on node node1. The taint has key key1, value value1, and taint effect NoSchedule. This means that no pod will be able to schedule onto node1 unless it has a matching toleration.

To remove the taint added by the command above, you can run:

kubectl taint nodes node1 key1=value1:NoSchedule-

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the kubectl taint line above, and thus a pod with either toleration would be able to schedule onto node1:

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoSchedule"

**tolerations**:

- **key**: "key1"

**operator**: "Exists"

**effect**: "NoSchedule"

Here's an example of a pod that uses tolerations:

[pods/pod-with-toleration.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/pods/pod-with-toleration.yaml)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: nginx

**labels**:

**env**: test

**spec**:

**containers**:

- **name**: nginx

**image**: nginx

**imagePullPolicy**: IfNotPresent

**tolerations**:

- **key**: "example-key"

**operator**: "Exists"

**effect**: "NoSchedule"

The default value for operator is Equal.

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

* The operator is Exists (in which case no value should be specified), or
* The operator is Equal and the values should be equal.

**Note:**

There are two special cases:

An empty key with operator Exists matches all keys, values and effects which means this will tolerate everything.

An empty effect matches all effects with key key1.

The above example used effect of NoSchedule. Alternatively, you can use effect of PreferNoSchedule.

The allowed values for the effect field are:

**NoExecute**

This affects pods that are already running on the node as follows:

* Pods that do not tolerate the taint are evicted immediately
* Pods that tolerate the taint without specifying tolerationSeconds in their toleration specification remain bound forever
* Pods that tolerate the taint with a specified tolerationSeconds remain bound for the specified amount of time. After that time elapses, the node lifecycle controller evicts the Pods from the node.

**NoSchedule**

No new Pods will be scheduled on the tainted node unless they have a matching toleration. Pods currently running on the node are **not** evicted.

**PreferNoSchedule**

PreferNoSchedule is a "preference" or "soft" version of NoSchedule. The control plane will try to avoid placing a Pod that does not tolerate the taint on the node, but it is not guaranteed.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

* if there is at least one un-ignored taint with effect NoSchedule then Kubernetes will not schedule the pod onto that node
* if there is no un-ignored taint with effect NoSchedule but there is at least one un-ignored taint with effect PreferNoSchedule then Kubernetes will try to not schedule the pod onto the node
* if there is at least one un-ignored taint with effect NoExecute then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

kubectl taint nodes node1 key1=value1:NoSchedule

kubectl taint nodes node1 key1=value1:NoExecute

kubectl taint nodes node1 key2=value2:NoSchedule

And a pod has two tolerations:

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoSchedule"

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoExecute"

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect NoExecute is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted. However, a toleration with NoExecute effect can specify an optional tolerationSeconds field that dictates how long the pod will stay bound to the node after the taint is added. For example,

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoExecute"

**tolerationSeconds**: 3600

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

## **Example Use Cases**

Taints and tolerations are a flexible way to steer pods away from nodes or evict pods that shouldn't be running. A few of the use cases are

* **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, kubectl taint nodes nodename dedicated=groupName:NoSchedule) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/)). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them and ensure they only use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. dedicated=groupName), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with dedicated=groupName.
* **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. kubectl taint nodes nodename special=true:NoSchedule or kubectl taint nodes nodename special=true:PreferNoSchedule) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/). For example, it is recommended to use [Extended Resources](https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#extended-resources) to represent the special hardware, taint your special hardware nodes with the extended resource name and run the [ExtendedResourceToleration](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#extendedresourcetoleration) admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the ExtendedResourceToleration admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.
* **Taint based Evictions**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

## **Taint based Evictions**

**FEATURE STATE:** Kubernetes v1.18 [stable]

The node controller automatically taints a Node when certain conditions are true. The following taints are built in:

* node.kubernetes.io/not-ready: Node is not ready. This corresponds to the NodeCondition Ready being "False".
* node.kubernetes.io/unreachable: Node is unreachable from the node controller. This corresponds to the NodeCondition Ready being "Unknown".
* node.kubernetes.io/memory-pressure: Node has memory pressure.
* node.kubernetes.io/disk-pressure: Node has disk pressure.
* node.kubernetes.io/pid-pressure: Node has PID pressure.
* node.kubernetes.io/network-unavailable: Node's network is unavailable.
* node.kubernetes.io/unschedulable: Node is unschedulable.
* node.cloudprovider.kubernetes.io/uninitialized: When the kubelet is started with "external" cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

In case a node is to be drained, the node controller or the kubelet adds relevant taints with NoExecute effect. This effect is added by default for the node.kubernetes.io/not-ready and node.kubernetes.io/unreachable taints. If the fault condition returns to normal, the kubelet or node controller can remove the relevant taint(s).

In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the

‘node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

**Note:** The control plane limits the rate of adding new taints to nodes. This rate limiting manages the number of evictions that are triggered when many nodes become unreachable at once (for example: if there is a network disruption).

You can specify tolerationSeconds for a Pod to define how long that Pod stays bound to a failing or unresponsive Node.

For example, you might want to keep an application with a lot of local state bound to node for a long time in the event of network partition, hoping that the partition will recover and thus the pod eviction can be avoided. The toleration you set for that Pod might look like:

**tolerations**:

- **key**: "node.kubernetes.io/unreachable"

**operator**: "Exists"

**effect**: "NoExecute"

**tolerationSeconds**: 6000

**Note:**

Kubernetes automatically adds a toleration for node.kubernetes.io/not-ready and node.kubernetes.io/unreachable with tolerationSeconds=300, unless you, or a controller, set those tolerations explicitly.

These automatically-added tolerations mean that Pods remain bound to Nodes for 5 minutes after one of these problems is detected.

[DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) pods are created with NoExecute tolerations for the following taints with no tolerationSeconds:

* node.kubernetes.io/unreachable
* node.kubernetes.io/not-ready

This ensures that DaemonSet pods are never evicted due to these problems.

## **Taint Nodes by Condition**

The control plane, using the node [controller](https://kubernetes.io/docs/concepts/architecture/controller/), automatically creates taints with a NoSchedule effect for [node conditions](https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/#node-conditions).

The scheduler checks taints, not node conditions, when it makes scheduling decisions. This ensures that node conditions don't directly affect scheduling. For example, if the DiskPressure node condition is active, the control plane adds the node.kubernetes.io/disk-pressure taint and does not schedule new pods onto the affected node. If the MemoryPressure node condition is active, the control plane adds the node.kubernetes.io/memory-pressure taint.

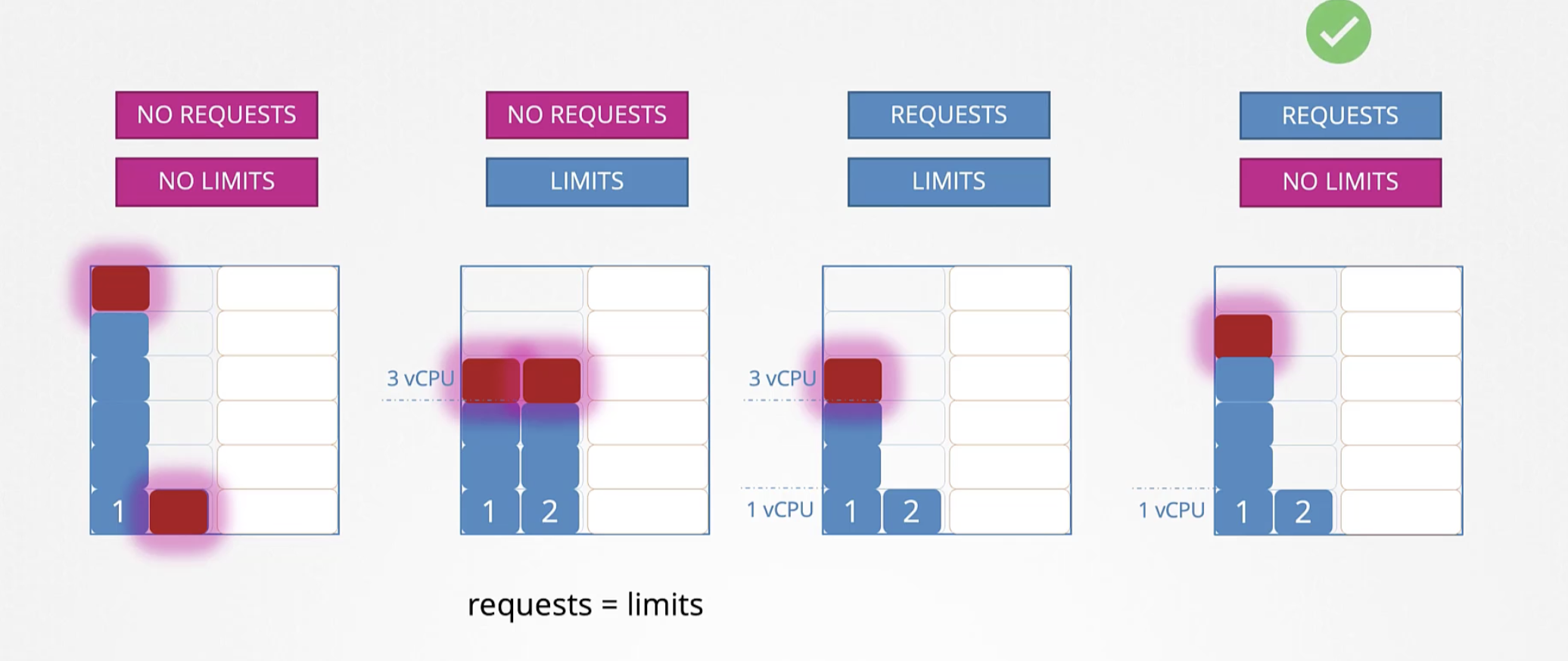
You can ignore node conditions for newly created pods by adding the corresponding Pod tolerations. The control plane also adds the node.kubernetes.io/memory-pressure toleration on pods that have a [QoS class](https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/) other than BestEffort. This is because Kubernetes treats pods in the Guaranteed or Burstable QoS classes (even pods with no memory request set) as if they are able to cope with memory pressure, while new BestEffort pods are not scheduled onto the affected node.

The DaemonSet controller automatically adds the following NoSchedule tolerations to all daemons, to prevent DaemonSets from breaking.

* node.kubernetes.io/memory-pressure
* node.kubernetes.io/disk-pressure
* node.kubernetes.io/pid-pressure (1.14 or later)
* node.kubernetes.io/unschedulable (1.10 or later)
* node.kubernetes.io/network-unavailable (host network only)

Adding these tolerations ensures backward compatibility. You can also add arbitrary tolerations to DaemonSets.

[Resource Management for Pods and Containers](https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/)



Replicaset

<https://uklabs.kodekloud.com/topic/practice-test-replicasets-2/>

kubectl create -f replicaset-definition.yml

kubectl get replicaset

kubectl delete replicaset myapp-replicaset

kubectl replace -f replicaset-definition.yml

kubectl scale --replicas=6 -f replicaset-definition.yml

kubectl scale --replicas=6 replicaset myapp-replicaset

Type Name

Answer 14: kubectl edit rs new-replica-set

kubectl get pods

kubectl delete pod

Deployments:

kubectl create -f deployment-definition.yml

kubectl get all

kubectl get deployments

kubectl get replicaset

kubectl get pods

kubectl delete replicaset myapp-replicaset

kubectl replace -f replicaset-definition.yml

kubectl scale --replicas=6 -f replicaset-definition.yml

kubectl scale --replicas=6 replicaset myapp-replicaset

Type Name

**Create an NGINX Pod**

kubectl run nginx --image=nginx

**Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run nginx --image=nginx --dry-run=client -o yaml

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)**

kubectl create deployment --image=nginx nginx --dry-run=client -o yaml

**Generate Deployment YAML file (-o yaml). Don’t create it(–dry-run) and save it to a file.**

kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml

**Make necessary changes to the file (for example, adding more replicas) and then create the deployment.**

kubectl create -f nginx-deployment.yaml

**OR**

**In k8s version 1.19+, we can specify the --replicas option to create a deployment with 4 replicas.**

kubectl create deployment --image=nginx nginx --replicas=4 --dry-run=client -o yaml > nginx-deployment.yaml

kubectl set image deployment nginx nginx=nginx:1.18

**Service type**

For some parts of your application (for example, frontends) you may want to expose a Service onto

an external IP address, one that's accessible from outside of your cluster.

Kubernetes Service types allow you to specify what kind of Service you want.

The available type values and their behaviours are:

[**ClusterIP**](https://kubernetes.io/docs/concepts/services-networking/service/#type-clusterip)

Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the

cluster. This is the default that is used if you don't explicitly specify a type for a Service. You can expose the Service to

the public internet using an [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) or a [Gateway](https://gateway-api.sigs.k8s.io/).

# Services Cluster IP

apiVersion: v1

kind: Service

metadata:

name: backend

spec:

type: ClusterIP

selector:

app: myapp

type: backend

ports:

- port: 80 # Port of Service Maps with Cluster IP of Service

targetPort: 80 # Target Pod Port

[**NodePort**](https://kubernetes.io/docs/concepts/services-networking/service/#type-nodeport)

Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: ClusterIP.

kubectl expose deployment nginx --port 80

# Services NodePort

apiVersion: v1

kind: Service

metadata:

name: frontend

spec:

type: NodePort

selector:

app: myapp

type: frontend

ports:

- port: 80 # Port of Service Maps with Cluster IP of Service

targetPort: 80 # Target Pod Port

nodePort: 30008 # Used to Access Web Server Externally Range From 30000-32767

[**LoadBalancer**](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer)

Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

# Service Load Balancer Work With Only Cloud Platform

apiVersion: v1

kind: Service

metadata:

name: myapp-service

spec:

selector:

app: Selector Label

type: LoadBalancer

ports:

- port: 80

targetPort: 80

nodePort: 30008

[**ExternalName**](https://kubernetes.io/docs/concepts/services-networking/service/#externalname)

Maps the Service to the contents of the externalName field (for example, to the hostname api.foo.bar.example). The mapping configures your cluster's DNS server to return a CNAME record with that external hostname value. No proxying of any kind is set up.

The type field in the Service API is designed as nested functionality - each level adds to the previous. However there is an exception to this nested design. You can define a LoadBalancer Service by [disabling the load balancer NodePort allocation](https://kubernetes.io/docs/concepts/services-networking/service/#load-balancer-nodeport-allocation).

type: ClusterIP

This default Service type assigns an IP address from a pool of IP addresses that your cluster has reserved for that purpose.

Several of the other types for Service build on the ClusterIP type as a foundation.

If you define a Service that has the .spec.clusterIP set to "None" then Kubernetes does not assign an IP address. See [headless Services](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) for more information.

Choosing your own IP address

You can specify your own cluster IP address as part of a Service creation request. To do this, set the .spec.clusterIP field. For example, if you already have an existing DNS entry that you wish to reuse, or legacy systems that are configured for a specific IP address and difficult to re-configure.

The IP address that you choose must be a valid IPv4 or IPv6 address from within the service-cluster-ip-range CIDR range that is configured for the API server. If you try to create a Service with an invalid clusterIP address value, the API server will return a 422 HTTP status code to indicate that there's a problem.

Read [avoiding collisions](https://kubernetes.io/docs/reference/networking/virtual-ips/#avoiding-collisions) to learn how Kubernetes helps reduce the risk and impact of two different Services both trying to use the same IP address.

type: NodePort

If you set the type field to NodePort, the Kubernetes control plane allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service. Your Service reports the allocated port in its .spec.ports[\*].nodePort field.

Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes' IP addresses directly.

For a node port Service, Kubernetes additionally allocates a port (TCP, UDP or SCTP to match the protocol of the Service). Every node in the cluster configures itself to listen on that assigned port and to forward traffic to one of the ready endpoints associated with that Service. You'll be able to contact the type: NodePort Service, from outside the cluster, by connecting to any node using the appropriate protocol (for example: TCP), and the appropriate port (as assigned to that Service).

Choosing your own port

If you want a specific port number, you can specify a value in the nodePort field. The control plane will either allocate you that port or report that the API transaction failed. This means that you need to take care of possible port collisions yourself. You also have to use a valid port number, one that's inside the range configured for NodePort use.

Here is an example manifest for a Service of type: NodePort that specifies a NodePort value (30007, in this example):

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**type**: NodePort

**selector**:

**app.kubernetes.io/name**: MyApp

**ports**:

- **port**: 80

*# By default and for convenience, the `targetPort` is set to*

*# the same value as the `port` field.*

**targetPort**: 80

*# Optional field*

*# By default and for convenience, the Kubernetes control plane*

*# will allocate a port from a range (default: 30000-32767)*

**nodePort**: 30007

Reserve Nodeport ranges to avoid collisions

**FEATURE STATE:** Kubernetes v1.29 [stable]

The policy for assigning ports to NodePort services applies to both the auto-assignment and the manual assignment scenarios. When a user wants to create a NodePort service that uses a specific port, the target port may conflict with another port that has already been assigned.

To avoid this problem, the port range for NodePort services is divided into two bands. Dynamic port assignment uses the upper band by default, and it may use the lower band once the upper band has been exhausted. Users can then allocate from the lower band with a lower risk of port collision.

Custom IP address configuration for type: NodePort Services

You can set up nodes in your cluster to use a particular IP address for serving node port services. You might want to do this if each node is connected to multiple networks (for example: one network for application traffic, and another network for traffic between nodes and the control plane).

If you want to specify particular IP address(es) to proxy the port, you can set the --nodeport-addresses flag for kube-proxy or the equivalent nodePortAddresses field of the [kube-proxy configuration file](https://kubernetes.io/docs/reference/config-api/kube-proxy-config.v1alpha1/) to particular IP block(s).

This flag takes a comma-delimited list of IP blocks (e.g. 10.0.0.0/8, 192.0.2.0/25) to specify IP address ranges that kube-proxy should consider as local to this node.

For example, if you start kube-proxy with the --nodeport-addresses=127.0.0.0/8 flag, kube-proxy only selects the loopback interface for NodePort Services. The default for --nodeport-addresses is an empty list. This means that kube-proxy should consider all available network interfaces for NodePort. (That's also compatible with earlier Kubernetes releases.)

**Note:** This Service is visible as <NodeIP>:spec.ports[\*].nodePort and .spec.clusterIP:spec.ports[\*].port. If the --nodeport-addresses flag for kube-proxy or the equivalent field in the kube-proxy configuration file is set, <NodeIP> would be a filtered node IP address (or possibly IP addresses).

type: LoadBalancer

On cloud providers which support external load balancers, setting the type field to LoadBalancer provisions a load balancer for your Service. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's .status.loadBalancer field. For example:

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app.kubernetes.io/name**: MyApp

**ports**:

- **protocol**: TCP

**port**: 80

**targetPort**: 9376

**clusterIP**: 10.0.171.239

**type**: LoadBalancer

**status**:

**loadBalancer**:

**ingress**:

- **ip**: 192.0.2.127

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

To implement a Service of type: LoadBalancer, Kubernetes typically starts off by making the changes that are equivalent to you requesting a Service of type: NodePort. The cloud-controller-manager component then configures the external load balancer to forward traffic to that assigned node port.

You can configure a load balanced Service to [omit](https://kubernetes.io/docs/concepts/services-networking/service/#load-balancer-nodeport-allocation) assigning a node port, provided that the cloud provider implementation supports this.

Some cloud providers allow you to specify the loadBalancerIP. In those cases, the load-balancer is created with the user-specified loadBalancerIP. If the loadBalancerIP field is not specified, the load balancer is set up with an ephemeral IP address. If you specify a loadBalancerIP but your cloud provider does not support the feature, the loadbalancerIP field that you set is ignored.

**Note:**

The.spec.loadBalancerIP field for a Service was deprecated in Kubernetes v1.24.

This field was under-specified and its meaning varies across implementations. It also cannot support dual-stack networking. This field may be removed in a future API version.

If you're integrating with a provider that supports specifying the load balancer IP address(es) for a Service via a (provider specific) annotation, you should switch to doing that.

If you are writing code for a load balancer integration with Kubernetes, avoid using this field. You can integrate with [Gateway](https://gateway-api.sigs.k8s.io/) rather than Service, or you can define your own (provider specific) annotations on the Service that specify the equivalent detail.

Load balancers with mixed protocol types

**FEATURE STATE:** Kubernetes v1.26 [stable]

By default, for LoadBalancer type of Services, when there is more than one port defined, all ports must have the same protocol, and the protocol must be one which is supported by the cloud provider.

The feature gate MixedProtocolLBService (enabled by default for the kube-apiserver as of v1.24) allows the use of different protocols for LoadBalancer type of Services, when there is more than one port defined.

**Note:** The set of protocols that can be used for load balanced Services is defined by your cloud provider; they may impose restrictions beyond what the Kubernetes API enforces.

Namespaces:

kubectl get namespace

kubectl create namespace dev

kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>

kubectl get pods --namespace=<insert-namespace-name-here>

kubectl config set-context $(kubectl config current-context) --namespace=<insert-namespace-name-here>

kubectl config set-context --current --namespace=<insert-namespace-name-here>

*# Validate it*

kubectl config view --minify | grep namespace:

kubectl get pods --all-namespaces

kubectl run pod redis --image=redis --namespace=finance

Create Objects

> kubectl apply –f nginx.yaml

Update Objects

> kubectl apply –f nginx.yaml

> kubectl run --image=nginx nginx

> kubectl create deployment --image=nginx nginx

> kubectl expose deployment nginx --port 80

> kubectl edit deployment nginx

> kubectl scale deployment nginx --replicas=5

> kubectl set image deployment nginx nginx=nginx:1.18

[Pods](https://kubernetes.io/docs/concepts/workloads/pods/)

**Create an NGINX Pod**

kubectl run nginx --image=nginx

**Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run nginx --image=nginx --dry-run=client -o yaml

[Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)

**Create a deployment**  
kubectl create deployment --image=nginx nginx

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)**

kubectl create deployment --image=nginx nginx --dry-run=client -o yaml

**Generate Deployment with 4 Replicas**

kubectl create deployment nginx --image=nginx --replicas=4

You can also scale a deployment using the kubectl scale command.

kubectl scale deployment nginx --replicas=4

**Another way to do this is to save the YAML definition to a file and modify**

kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > nginx-deployment.yaml

You can then update the YAML file with the replicas or any other field before creating the deployment.

[Service](https://kubernetes.io/docs/concepts/services-networking/service/)

**Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379**

kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors)

Or

kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml (This will not use the pods labels as selectors, instead it will assume selectors as **app=redis.**[You cannot pass in selectors as an option.](https://github.com/kubernetes/kubernetes/issues/46191) So it does not work very well if your pod has a different label set.

So generate the file and modify the selectors before creating the service)

**Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:**

kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors, [but you cannot specify the node port](https://github.com/kubernetes/kubernetes/issues/25478). You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

Or

kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml

(This will not use the pods labels as selectors)

Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the kubectl expose command. If you need to specify a node port, generate a definition file using the same command and manually input the nodeport before creating the service.

**Reference:**

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>

<https://kubernetes.io/docs/reference/kubectl/conventions/>

[Labels and Selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)

Labels on the matchLabels in replicaset and the pods must be same.

Labels on the service selector and the pods must be same.

[Assigning Pods to Nodes](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/)

You can constrain a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/) so that it is *restricted* to run on particular [node(s)](https://kubernetes.io/docs/concepts/architecture/nodes/), or to *prefer* to run on particular nodes. There are several ways to do this and the recommended approaches all use [label selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) to facilitate the selection. Often, you do not need to set any such constraints; the [scheduler](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector) field matching against [node labels](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#built-in-node-labels)
* [Affinity and anti-affinity](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity)
* [nodeName](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodename) field
* [Pod topology spread constraints](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#pod-topology-spread-constraints)

Node labels

Like many other Kubernetes objects, nodes have [labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/). You can [attach labels manually](https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/#add-a-label-to-a-node). Kubernetes also populates a [standard set of labels](https://kubernetes.io/docs/reference/node/node-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](https://kubernetes.io/docs/reference/generated/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](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction)  admission plugin](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction) 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](https://kubernetes.io/docs/reference/access-authn-authz/node/) 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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector). 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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#built-in-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](https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-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](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#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](https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/) 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](https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-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](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/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](https://kubernetes.io/docs/reference/scheduling/config/#multiple-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](https://kubernetes.io/docs/reference/scheduling/config/" \l "scheduling-plugins) in the [scheduler configuration](https://kubernetes.io/docs/reference/scheduling/config/). 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](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#how-daemon-pods-are-scheduled), 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](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#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](https://kubernetes.io/docs/reference/labels-annotations-taints/).

**Note:** Inter-pod affinity and anti-affinity require substantial amount 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 labelled, 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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#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](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/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](https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/) 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](https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/) 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](https://git.k8s.io/design-proposals-archive/scheduling/podaffinity.md).

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

Read [Operators](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#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 a 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](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/).

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**:

**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)*

**matchLabelKeys**:

- pod-template-hash

mismatchLabelKeys

**FEATURE STATE:** Kubernetes v1.29 [alpha]

**Note:**

The mismatchLabelKeys field is a 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](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/).

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](https://kubernetes.io/docs/tutorials/stateful-application/zookeeper/" \l "tolerating-node-failure) 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.

**Note:** 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](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity) or a the [nodeselector field](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/" \l "nodeselector) 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](https://kubernetes.io/docs/concepts/workloads/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](https://kubernetes.io/docs/concepts/scheduling-eviction/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.