Nodes

Kubernetes runs your workload by placing containers into Pods to run on *Nodes*. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.

The components on a node include the kubelet, a container runtime, and the kube-proxy.

Management

There are two main ways to have Nodes added to the API server:

1. The kubelet on a node self-registers to the control plane
2. You (or another human user) manually add a Node object

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For example, if you try to create a Node from the following JSON manifest:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the `metadata.name` field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

> **Note:**
> Kubernetes keeps the object for the invalid Node and continues checking to see whether it becomes healthy.
>
> You, or a controller, must explicitly delete the Node object to stop that health checking.

The name of a Node object must be a valid DNS subdomain name.

Node name uniqueness

The name identifies a Node. Two Nodes cannot have the same name at the same time. Kubernetes also assumes that a resource with the same name is the same object. In case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents) and attributes like node labels. This may lead to

inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from API server first and re-added after the update.

Self-registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the API server.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`).

  No-op if `register-node` is false.
- `--node-ip` - Optional comma-separated list of the IP addresses for the node. You can only specify a single address for each address family. For example, in a single-stack IPv4 cluster, you set this value to be the IPv4 address that the kubelet should use for the node. See configure IPv4/IPv6 dual stack for details of running a dual-stack cluster.

  If you don't provide this argument, the kubelet uses the node's default IPv4 address, if any; if the node has no IPv4 addresses then the kubelet uses the node's default IPv6 address.
- `--node-labels` - Labels to add when registering the node in the cluster (see label restrictions enforced by the NodeRestriction admission plugin).
- `--node-status-update-frequency` - Specifies how often kubelet posts its node status to the API server.

When the Node authorization mode and NodeRestriction admission plugin are enabled, kubelets are only authorized to create/modify their own Node resource.

> **Note:**
> As mentioned in the Node name uniqueness section, when Node configuration needs to be updated, it is a good practice to re-register the node with the API server. For example, if the kubelet being restarted with the new set of `--node-labels`, but the same Node name is used, the change will not take an effect, as labels are being set on the Node registration.
>
> Pods already scheduled on the Node may misbehave or cause issues if the Node configuration will be changed on kubelet restart. For example, already running Pod may be tainted against the new labels assigned to the Node, while other Pods, that are incompatible with that Pod will be scheduled based on this new label. Node re-registration ensures all Pods will be drained and properly re-scheduled.

Manual Node administration

You can create and modify Node objects using kubectl.

When you want to create Node objects manually, set the kubelet flag `--register-node=false`.

You can modify Node objects regardless of the setting of `--register-node`. For example, you can set labels on an existing Node or mark it unschedulable.

You can use labels on Nodes in conjunction with node selectors on Pods to control scheduling. For example, you can constrain a Pod to only be eligible to run on a subset of the available nodes.

Marking a node as unschedulable prevents the scheduler from placing new pods onto that Node but does not affect existing Pods on the Node. This is useful as a preparatory step before a node reboot or other maintenance.

To mark a Node unschedulable, run:

```
kubectl cordon $NODENAME
```

See Safely Drain a Node for more details.

> **Note:** Pods that are part of a DaemonSet tolerate being run on an unschedulable Node. DaemonSets typically provide node-local services that should run on the Node even if it is being drained of workload applications.

Node status

A Node's status contains the following information:

- Addresses
- Conditions
- Capacity and Allocatable
- Info

You can use `kubectl` to view a Node's status and other details:

```
kubectl describe node <insert-node-name-here>
```

Each section of the output is described below.

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet `--hostname-override` parameter.
- ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
- InternalIP: Typically the IP address of the node that is routable only within the cluster.

Conditions

The `conditions` field describes the status of all `Running` nodes. Examples of conditions include:

Node Condition	Description
Ready	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy and is not accepting pods, and `Unknown` if the node controller has not heard from the node in the last `node-monitor-grace-period` (default is 40 seconds)

Node Condition	Description
`DiskPressure`	`True` if pressure exists on the disk size—that is, if the disk capacity is low; otherwise `False`
`MemoryPressure`	`True` if pressure exists on the node memory—that is, if the node memory is low; otherwise `False`
`PIDPressure`	`True` if pressure exists on the processes—that is, if there are too many processes on the node; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

> **Note:** If you use command-line tools to print details of a cordoned Node, the Condition includes `SchedulingDisabled`. `SchedulingDisabled` is not a Condition in the Kubernetes API; instead, cordoned nodes are marked Unschedulable in their spec.

In the Kubernetes API, a node's condition is represented as part of the `.status` of the Node resource. For example, the following JSON structure describes a healthy node:

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

When problems occur on nodes, the Kubernetes control plane automatically creates taints that match the conditions affecting the node. An example of this is when the `status` of the Ready condition remains `Unknown` or `False` for longer than the kube-controller-manager's `NodeMonitorGracePeriod`, which defaults to 40 seconds. This will cause either an `node.kubernetes.io/unreachable` taint, for an `Unknown` status, or a `node.kubernetes.io/not-ready` taint, for a `False` status, to be added to the Node.

These taints affect pending pods as the scheduler takes the Node's taints into consideration when assigning a pod to a Node. Existing pods scheduled to the node may be evicted due to the application of `NoExecute` taints. Pods may also have tolerations that let them schedule to and continue running on a Node even though it has a specific taint.

See Taint Based Evictions and Taint Nodes by Condition for more details.

Capacity and Allocatable

Describes the resources available on the node: CPU, memory, and the maximum number of pods that can be scheduled onto the node.

The fields in the capacity block indicate the total amount of resources that a Node has. The allocatable block indicates the amount of resources on a Node that is available to be consumed by normal Pods.

You may read more about capacity and allocatable resources while learning how to reserve compute resources on a Node.

Info

Describes general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), container runtime details, and which operating system the node uses. The kubelet gathers this information from the node and publishes it into the Kubernetes API.

Heartbeats

Heartbeats, sent by Kubernetes nodes, help your cluster determine the availability of each node, and to take action when failures are detected.

For nodes there are two forms of heartbeats:

- updates to the `.status` of a Node
- Lease objects within the `kube-node-lease` namespace. Each Node has an associated Lease object.

Compared to updates to `.status` of a Node, a Lease is a lightweight resource. Using Leases for heartbeats reduces the performance impact of these updates for large clusters.

The kubelet is responsible for creating and updating the `.status` of Nodes, and for updating their related Leases.

- The kubelet updates the node's `.status` either when there is change in status or if there has been no update for a configured interval. The default interval for `.status` updates to Nodes is 5 minutes, which is much longer than the 40 second default timeout for unreachable nodes.
- The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from updates to the Node's `.status`. If the Lease update fails, the kubelet retries, using exponential backoff that starts at 200 milliseconds and capped at 7 seconds.

Node controller

The node controller is a Kubernetes control plane component that manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment and whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health. The node controller is responsible for:

- In the case that a node becomes unreachable, updating the `Ready` condition in the Node's `.status` field. In this case the node controller sets the `Ready` condition to `Unknown`.
- If a node remains unreachable: triggering API-initiated eviction for all of the Pods on the unreachable node. By default, the node controller waits 5 minutes between marking the node as `Unknown` and submitting the first eviction request.

By default, the node controller checks the state of each node every 5 seconds. This period can be configured using the `--node-monitor-period` flag on the `kube-controller-manager` component.

Rate limits on eviction

In most cases, the node controller limits the eviction rate to `--node-eviction-rate` (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.

The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (the `Ready` condition is `Unknown` or `False`) at the same time:

- If the fraction of unhealthy nodes is at least `--unhealthy-zone-threshold` (default 0.55), then the eviction rate is reduced.
- If the cluster is small (i.e. has less than or equal to `--large-cluster-size-threshold` nodes - default 50), then evictions are stopped.
- Otherwise, the eviction rate is reduced to `--secondary-node-eviction-rate` (default 0.01) per second.

The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the control plane while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then the eviction mechanism does not take per-zone unavailability into account.

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy, then the node controller evicts at the normal rate of `--node-eviction-rate`. The corner case is when all zones are completely unhealthy (none of the nodes in the cluster are healthy). In such a case, the node controller assumes that there is some problem with connectivity between the control plane and the nodes, and doesn't perform any evictions. (If there has been an outage and some nodes reappear, the node controller does evict pods from the remaining nodes that are unhealthy or unreachable).

The node controller is also responsible for evicting pods running on nodes with `NoExecute` taints, unless those pods tolerate that taint. The node controller also adds taints corresponding to node problems like node unreachable or not ready. This means that the scheduler won't place Pods onto unhealthy nodes.

Resource capacity tracking

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs. Nodes that self register report their capacity during registration. If you manually add a Node, then you need to set the node's capacity information when you add it.

The Kubernetes scheduler ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet's control.

> **Note:** If you want to explicitly reserve resources for non-Pod processes, see reserve resources for system daemons.

Node topology

FEATURE STATE: `Kubernetes v1.18 [beta]`

If you have enabled the `TopologyManager` feature gate, then the kubelet can use topology hints when making resource assignment decisions. See Control Topology Management Policies on a Node for more information.

Graceful node shutdown

FEATURE STATE: `Kubernetes v1.21 [beta]`

The kubelet attempts to detect node system shutdown and terminates pods running on the node.

Kubelet ensures that pods follow the normal pod termination process during the node shutdown. During node shutdown, the kubelet does not accept new Pods (even if those Pods are already bound to the node).

The Graceful node shutdown feature depends on systemd since it takes advantage of systemd inhibitor locks to delay the node shutdown with a given duration.

Graceful node shutdown is controlled with the `GracefulNodeShutdown` feature gate which is enabled by default in 1.21.

Note that by default, both configuration options described below, `shutdownGracePeriod` and `shutdownGracePeriodCriticalPods` are set to zero, thus not activating the graceful node shutdown functionality. To activate the feature, the two kubelet config settings should be configured appropriately and set to non-zero values.

Once systemd detects or notifies node shutdown, the kubelet sets a `NotReady` condition on the Node, with the `reason` set to `"node is shutting down"`. The kube-scheduler honors this condition and does not schedule any Pods onto the affected node; other third-party schedulers are expected to follow the same logic. This means that new Pods won't be scheduled onto that node and therefore none will start.

The kubelet **also** rejects Pods during the `PodAdmission` phase if an ongoing node shutdown has been detected, so that even Pods with a toleration for `node.kubernetes.io/not-ready:NoSchedule` do not start there.

At the same time when kubelet is setting that condition on its Node via the API, the kubelet also begins terminating any Pods that are running locally.

During a graceful shutdown, kubelet terminates pods in two phases:

1. Terminate regular pods running on the node.
2. Terminate critical pods running on the node.

Graceful node shutdown feature is configured with two `KubeletConfiguration` options:

- `shutdownGracePeriod`：
  - Specifies the total duration that the node should delay the shutdown by. This is the total grace period for pod termination for both regular and critical pods.
- `shutdownGracePeriodCriticalPods`：
  - Specifies the duration used to terminate critical pods during a node shutdown. This value should be less than `shutdownGracePeriod`.

> **Note:** There are cases when Node termination was cancelled by the system (or perhaps manually by an administrator). In either of those situations the Node will return to the `Ready` state. However Pods which already started the process of termination will not be restored by kubelet and will need to be re-scheduled.

For example, if `shutdownGracePeriod=30s`, and `shutdownGracePeriodCriticalPods=10s`, kubelet will delay the node shutdown by 30 seconds. During the shutdown, the first 20 (30-10) seconds would be reserved for gracefully terminating normal pods, and the last 10 seconds would be reserved for terminating critical pods.

> **Note:**
> When pods were evicted during the graceful node shutdown, they are marked as shutdown. Running `kubectl get pods` shows the status of the evicted pods as `Terminated`. And `kubectl describe pod` indicates that the pod was evicted because of node shutdown:

```
Reason:         Terminated
Message:        Pod was terminated in response to imminent node shutdown.
```

Pod Priority based graceful node shutdown

FEATURE STATE: `Kubernetes v1.24 [beta]`

To provide more flexibility during graceful node shutdown around the ordering of pods during shutdown, graceful node shutdown honors the PriorityClass for Pods, provided that you enabled this feature in your cluster. The feature allows cluster administers to explicitly define the ordering of pods during graceful node shutdown based on priority classes.

The Graceful Node Shutdown feature, as described above, shuts down pods in two phases, non-critical pods, followed by critical pods. If additional flexibility is needed to explicitly define the ordering of pods during shutdown in a more granular way, pod priority based graceful shutdown can be used.

When graceful node shutdown honors pod priorities, this makes it possible to do graceful node shutdown in multiple phases, each phase shutting down a particular priority class of pods. The kubelet can be configured with the exact phases and shutdown time per phase.

Assuming the following custom pod priority classes in a cluster,

Pod priority class name	Pod priority class value
`custom-class-a`	100000
`custom-class-b`	10000
`custom-class-c`	1000
`regular/unset`	0

Within the kubelet configuration the settings for `shutdownGracePeriodByPodPriority` could look like:

Pod priority class value	Shutdown period
100000	10 seconds
10000	180 seconds
1000	120 seconds
0	60 seconds

The corresponding kubelet config YAML configuration would be:

```yaml
shutdownGracePeriodByPodPriority:
  - priority: 100000
    shutdownGracePeriodSeconds: 10
  - priority: 10000
    shutdownGracePeriodSeconds: 180
  - priority: 1000
    shutdownGracePeriodSeconds: 120
  - priority: 0
    shutdownGracePeriodSeconds: 60
```

The above table implies that any pod with `priority` value >= 100000 will get just 10 seconds to stop, any pod with value >= 10000 and < 100000 will get 180 seconds to stop, any pod with value >= 1000 and < 10000 will get 120 seconds to stop. Finally, all other pods will get 60 seconds to stop.

One doesn't have to specify values corresponding to all of the classes. For example, you could instead use these settings:

Pod priority class value	Shutdown period
100000	300 seconds
1000	120 seconds
0	60 seconds

In the above case, the pods with `custom-class-b` will go into the same bucket as `custom-class-c` for shutdown.

If there are no pods in a particular range, then the kubelet does not wait for pods in that priority range. Instead, the kubelet immediately skips to the next priority class value range.

If this feature is enabled and no configuration is provided, then no ordering action will be taken.

Using this feature requires enabling the `GracefulNodeShutdownBasedOnPodPriority` feature gate , and setting `ShutdownGracePeriodByPodPriority` in the kubelet config to the desired configuration containing the pod priority class values and their respective shutdown periods.

> **Note:** The ability to take Pod priority into account during graceful node shutdown was introduced as an Alpha feature in Kubernetes v1.23. In Kubernetes 1.27 the feature is Beta and is enabled by default.

Metrics `graceful_shutdown_start_time_seconds` and `graceful_shutdown_end_time_seconds` are emitted under the kubelet subsystem to monitor node shutdowns.

Non Graceful node shutdown

FEATURE STATE: `Kubernetes v1.26 [beta]`

A node shutdown action may not be detected by kubelet's Node Shutdown Manager, either because the command does not trigger the inhibitor locks mechanism used by kubelet or because of a user error, i.e., the ShutdownGracePeriod and ShutdownGracePeriodCriticalPods are not configured properly. Please refer to above section Graceful Node Shutdown for more details.

When a node is shutdown but not detected by kubelet's Node Shutdown Manager, the pods that are part of a StatefulSet will be stuck in terminating status on the shutdown node and cannot move to a new running node. This is because kubelet on the shutdown node is not available to delete the pods so the StatefulSet cannot create a new pod with the same name. If there are volumes used by the pods, the VolumeAttachments will not be deleted from the original shutdown node so the volumes used by these pods cannot be attached to a new running node. As a result, the application running on the StatefulSet cannot function properly. If the original shutdown node comes up, the pods will be deleted by kubelet and new pods will be created on a different running node. If the original shutdown node does not come up, these pods will be stuck in terminating status on the shutdown node forever.

To mitigate the above situation, a user can manually add the taint `node.kubernetes.io/out-of-service` with either `NoExecute` or `NoSchedule` effect to a Node marking it out-of-service. If the `NodeOutOfServiceVolumeDetach` feature gate is enabled on kube-controller-manager, and a Node is marked out-of-service with this taint, the pods on the node will be forcefully

deleted if there are no matching tolerations on it and volume detach operations for the pods terminating on the node will happen immediately. This allows the Pods on the out-of-service node to recover quickly on a different node.

During a non-graceful shutdown, Pods are terminated in the two phases:

1. Force delete the Pods that do not have matching `out-of-service` tolerations.
2. Immediately perform detach volume operation for such pods.

> **Note:**
> - Before adding the taint `node.kubernetes.io/out-of-service` , it should be verified that the node is already in shutdown or power off state (not in the middle of restarting).
> - The user is required to manually remove the out-of-service taint after the pods are moved to a new node and the user has checked that the shutdown node has been recovered since the user was the one who originally added the taint.

Swap memory management

FEATURE STATE: `Kubernetes v1.22 [alpha]`

Prior to Kubernetes 1.22, nodes did not support the use of swap memory, and a kubelet would by default fail to start if swap was detected on a node. In 1.22 onwards, swap memory support can be enabled on a per-node basis.

To enable swap on a node, the `NodeSwap` feature gate must be enabled on the kubelet, and the `--fail-swap-on` command line flag or `failSwapOn` configuration setting must be set to false.

> **Warning:** When the memory swap feature is turned on, Kubernetes data such as the content of Secret objects that were written to tmpfs now could be swapped to disk.

A user can also optionally configure `memorySwap.swapBehavior` in order to specify how a node will use swap memory. For example,

```
memorySwap:
  swapBehavior: LimitedSwap
```

The available configuration options for `swapBehavior` are:

- `LimitedSwap` : Kubernetes workloads are limited in how much swap they can use. Workloads on the node not managed by Kubernetes can still swap.
- `UnlimitedSwap` : Kubernetes workloads can use as much swap memory as they request, up to the system limit.

If configuration for `memorySwap` is not specified and the feature gate is enabled, by default the kubelet will apply the same behaviour as the `LimitedSwap` setting.

The behaviour of the `LimitedSwap` setting depends if the node is running with v1 or v2 of control groups (also known as "cgroups"):

- **cgroupsv1:** Kubernetes workloads can use any combination of memory and swap, up to the pod's memory limit, if set.
- **cgroupsv2:** Kubernetes workloads cannot use swap memory.

For more information, and to assist with testing and provide feedback, please see KEP-2400 and its design proposal.

What's next

Learn more about the following:

- Components that make up a node.
- API definition for Node.
- Node section of the architecture design document.
- Taints and Tolerations.
- Node Resource Managers.
- Resource Management for Windows nodes.

Feedback

Was this page helpful?

Yes No

Last modified July 09, 2023 at 3:45 PM PST: Fix wrong feature state (#41927) (23aea946b5)