Cluster Architecture

The architectural concepts behind Kubernetes.

1 - 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)
`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.

2 - Communication between Nodes and the Control Plane

This document catalogs the communication paths between the API server and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

Node to Control Plane

Kubernetes has a "hub-and-spoke" API pattern. All API usage from nodes (or the pods they run) terminates at the API server. None of the other control plane components are designed to expose remote services. The API server is configured to listen for remote connections on a secure HTTPS port (typically 443) with one or more forms of client authentication enabled. One or more forms of authorization should be enabled, especially if anonymous requests or service account tokens are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the API server along with valid client credentials. A good approach is that the client credentials provided to the kubelet are in the form of a client certificate. See kubelet TLS bootstrapping for automated provisioning of kubelet client certificates.

Pods that wish to connect to the API server can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The `kubernetes` service (in `default` namespace) is configured with a virtual IP address that is redirected (via `kube-proxy`) to the HTTPS endpoint on the API server.

The control plane components also communicate with the API server over the secure port.

As a result, the default operating mode for connections from the nodes and pod running on the nodes to the control plane is secured by default and can run over untrusted and/or public networks.

Control plane to node

There are two primary communication paths from the control plane (the API server) to the nodes. The first is from the API server to the kubelet process which runs on each node in the cluster. The second is from the API server to any node, pod, or service through the API server's *proxy* functionality.

API server to kubelet

The connections from the API server to the kubelet are used for:

- Fetching logs for pods.
- Attaching (usually through `kubectl`) to running pods.
- Providing the kubelet's port-forwarding functionality.

These connections terminate at the kubelet's HTTPS endpoint. By default, the API server does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks and **unsafe** to run over untrusted and/or public networks.

To verify this connection, use the `--kubelet-certificate-authority` flag to provide the API server with a root certificate bundle to use to verify the kubelet's serving certificate.

If that is not possible, use SSH tunneling between the API server and kubelet if required to avoid connecting over an untrusted or public network.

Finally, Kubelet authentication and/or authorization should be enabled to secure the kubelet API.

API server to nodes, pods, and services

The connections from the API server to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing `https:` to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials. So while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted or public networks.

SSH tunnels

Kubernetes supports SSH tunnels to protect the control plane to nodes communication paths. In this configuration, the API server initiates an SSH tunnel to each node in the cluster (connecting to the SSH server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

> **Note:** SSH tunnels are currently deprecated, so you shouldn't opt to use them unless you know what you are doing. The Konnectivity service is a replacement for this communication channel.

Konnectivity service

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

As a replacement to the SSH tunnels, the Konnectivity service provides TCP level proxy for the control plane to cluster communication. The Konnectivity service consists of two parts: the Konnectivity server in the control plane network and the Konnectivity agents in the nodes network. The Konnectivity agents initiate connections to the Konnectivity server and maintain the network connections. After enabling the Konnectivity service, all control plane to nodes traffic goes through these connections.

Follow the Konnectivity service task to set up the Konnectivity service in your cluster.

What's next

- Read about the Kubernetes control plane components
- Learn more about Hubs and Spoke model
- Learn how to Secure a Cluster
- Learn more about the Kubernetes API
- Set up Konnectivity service
- Use Port Forwarding to Access Applications in a Cluster
- Learn how to Fetch logs for Pods, use kubectl port-forward

3 - Controllers

In robotics and automation, a *control loop* is a non-terminating loop that regulates the state of a system.

Here is one example of a control loop: a thermostat in a room.

When you set the temperature, that's telling the thermostat about your *desired state*. The actual room temperature is the *current state*. The thermostat acts to bring the current state closer to the desired state, by turning equipment on or off.

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

Controller pattern

A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

The controller might carry the action out itself; more commonly, in Kubernetes, a controller will send messages to the API server that have useful side effects. You'll see examples of this below.

Control via API server

The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a Pod, or perhaps several Pods, to carry out a task and then stop.

(Once scheduled, Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the control plane act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it `Finished`.

(This is a bit like how some thermostats turn a light off to indicate that your room is now at the temperature you set).

Direct control

In contrast with Job, some controllers need to make changes to things outside of your cluster.

For example, if you use a control loop to make sure there are enough Nodes in your cluster, then that controller needs something outside the current cluster to set up new Nodes when needed.

Controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

(There actually is a controller that horizontally scales the nodes in your cluster.)

The important point here is that the controller makes some changes to bring about your desired state, and then reports the current state back to your cluster's API server. Other control loops can observe that reported data and take their own actions.

In the thermostat example, if the room is very cold then a different controller might also turn on a frost protection heater. With Kubernetes clusters, the control plane indirectly works with IP address management tools, storage services, cloud provider APIs, and other services by extending Kubernetes to implement that.

Desired versus current state

Kubernetes takes a cloud-native view of systems, and is able to handle constant change.

Your cluster could be changing at any point as work happens and control loops automatically fix failures. This means that, potentially, your cluster never reaches a stable state.

As long as the controllers for your cluster are running and able to make useful changes, it doesn't matter if the overall state is stable or not.

Design

As a tenet of its design, Kubernetes uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state, and has a different kind of resource that it manages to make that desired state happen. For example, a controller for Jobs tracks Job objects (to discover new work) and Pod objects (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

It's useful to have simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so Kubernetes is designed to allow for that.

> **Note:**
> There can be several controllers that create or update the same kind of object. Behind the scenes, Kubernetes controllers make sure that they only pay attention to the resources linked to their controlling resource.
>
> For example, you can have Deployments and Jobs; these both create Pods. The Job controller does not delete the Pods that your Deployment created, because there is information (labels) the controllers can use to tell those Pods apart.

Ways of running controllers

Kubernetes comes with a set of built-in controllers that run inside the kube-controller-manager. These built-in controllers provide important core behaviors.

The Deployment controller and Job controller are examples of controllers that come as part of Kubernetes itself ("built-in" controllers). Kubernetes lets you run a resilient control plane, so that if any of the built-in controllers were to fail, another part of the control plane will take over the work.

You can find controllers that run outside the control plane, to extend Kubernetes. Or, if you want, you can write a new controller yourself. You can run your own controller as a set of Pods, or externally to Kubernetes. What fits best will depend on what that particular controller does.

What's next

- Read about the Kubernetes control plane

- Discover some of the basic Kubernetes objects
- Learn more about the Kubernetes API
- If you want to write your own controller, see Extension Patterns in Extending Kubernetes.

4 - Leases

Distributed systems often have a need for *leases*, which provide a mechanism to lock shared resources and coordinate activity between members of a set. In Kubernetes, the lease concept is represented by Lease objects in the `coordination.k8s.io` API Group, which are used for system-critical capabilities such as node heartbeats and component-level leader election.

Node heartbeats

Kubernetes uses the Lease API to communicate kubelet node heartbeats to the Kubernetes API server. For every `Node`, there is a `Lease` object with a matching name in the `kube-node-lease` namespace. Under the hood, every kubelet heartbeat is an **update** request to this `Lease` object, updating the `spec.renewTime` field for the Lease. The Kubernetes control plane uses the time stamp of this field to determine the availability of this `Node`.

See Node Lease objects for more details.

Leader election

Kubernetes also uses Leases to ensure only one instance of a component is running at any given time. This is used by control plane components like `kube-controller-manager` and `kube-scheduler` in HA configurations, where only one instance of the component should be actively running while the other instances are on stand-by.

API server identity

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

Starting in Kubernetes v1.26, each `kube-apiserver` uses the Lease API to publish its identity to the rest of the system. While not particularly useful on its own, this provides a mechanism for clients to discover how many instances of `kube-apiserver` are operating the Kubernetes control plane. Existence of kube-apiserver leases enables future capabilities that may require coordination between each kube-apiserver.

You can inspect Leases owned by each kube-apiserver by checking for lease objects in the `kube-system` namespace with the name `kube-apiserver-<sha256-hash>`. Alternatively you can use the label selector `k8s.io/component=kube-apiserver`:

```
kubectl -n kube-system get lease -l k8s.io/component=kube-apiserver
```

```
NAME                                        HOLDER
kube-apiserver-c4vwjftbvpc5os2vvzle4qg27a   kube-apiserver-c4vwjftbvpc5os2vvzle4qg27a_9cb
kube-apiserver-dz2dqprdpsgnm756t5rnov7yka   kube-apiserver-dz2dqprdpsgnm756t5rnov7yka_84f
kube-apiserver-fyloo45sdenffw2ugwaz3likua   kube-apiserver-fyloo45sdenffw2ugwaz3likua_c5f
```

The SHA256 hash used in the lease name is based on the OS hostname as seen by that API server. Each kube-apiserver should be configured to use a hostname that is unique within the cluster. New instances of kube-apiserver that use the same hostname will take over existing Leases using a new holder identity, as opposed to instantiating new Lease objects. You can check the hostname used by kube-apisever by checking the value of the `kubernetes.io/hostname` label:

```
kubectl -n kube-system get lease kube-apiserver-c4vwjftbvpc5os2vvzle4qg27a -o yaml
```

```
apiVersion: coordination.k8s.io/v1
kind: Lease
metadata:
  creationTimestamp: "2022-11-30T15:37:15Z"
  labels:
    k8s.io/component: kube-apiserver
    kubernetes.io/hostname: kind-control-plane
  name: kube-apiserver-c4vwjftbvpc5os2vvzle4qg27a
  namespace: kube-system
  resourceVersion: "18171"
  uid: d6c68901-4ec5-4385-b1ef-2d783738da6c
spec:
  holderIdentity: kube-apiserver-c4vwjftbvpc5os2vvzle4qg27a_9cbf54e5-1136-44bd-8f9a-1dcd1
  leaseDurationSeconds: 3600
  renewTime: "2022-11-30T18:04:27.912073Z"
```

Expired leases from kube-apiservers that no longer exist are garbage collected by new kube-apiservers after 1 hour.

You can disable API server identity leases by disabling the `APIServerIdentity` feature gate.

Workloads

Your own workload can define its own use of Leases. For example, you might run a custom controller where a primary or leader member performs operations that its peers do not. You define a Lease so that the controller replicas can select or elect a leader, using the Kubernetes API for coordination. If you do use a Lease, it's a good practice to define a name for the Lease that is obviously linked to the product or component. For example, if you have a component named Example Foo, use a Lease named `example-foo`.

If a cluster operator or another end user could deploy multiple instances of a component, select a name prefix and pick a mechanism (such as hash of the name of the Deployment) to avoid name collisions for the Leases.

You can use another approach so long as it achieves the same outcome: different software products do not conflict with one another.

5 - Cloud Controller Manager

FEATURE STATE: `Kubernetes v1.11 [beta]`

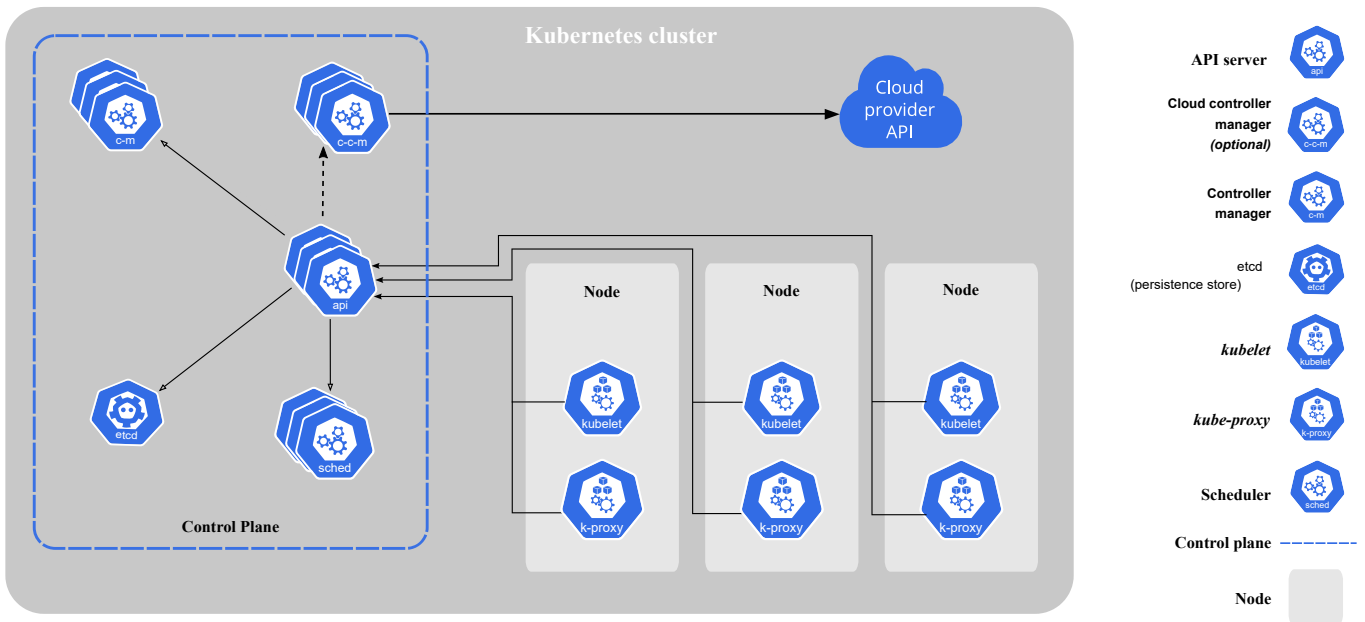
Cloud infrastructure technologies let you run Kubernetes on public, private, and hybrid clouds. Kubernetes believes in automated, API-driven infrastructure without tight coupling between components.

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

The cloud-controller-manager is structured using a plugin mechanism that allows different cloud providers to integrate their platforms with Kubernetes.

Design



The cloud controller manager runs in the control plane as a replicated set of processes (usually, these are containers in Pods). Each cloud-controller-manager implements multiple controllers in a single process.

> **Note:** You can also run the cloud controller manager as a Kubernetes addon rather than as part of the control plane.

Cloud controller manager functions

The controllers inside the cloud controller manager include:

Node controller

The node controller is responsible for updating Node objects when new servers are created in your cloud infrastructure. The node controller obtains information about the hosts running inside your tenancy with the cloud provider. The node controller performs the following functions:

1. Update a Node object with the corresponding server's unique identifier obtained from the cloud provider API.
2. Annotating and labelling the Node object with cloud-specific information, such as the region the node is deployed into and the resources (CPU, memory, etc) that it has available.

3. Obtain the node's hostname and network addresses.
4. Verifying the node's health. In case a node becomes unresponsive, this controller checks with your cloud provider's API to see if the server has been deactivated / deleted / terminated. If the node has been deleted from the cloud, the controller deletes the Node object from your Kubernetes cluster.

Some cloud provider implementations split this into a node controller and a separate node lifecycle controller.

Route controller

The route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in your Kubernetes cluster can communicate with each other.

Depending on the cloud provider, the route controller might also allocate blocks of IP addresses for the Pod network.

Service controller

Services integrate with cloud infrastructure components such as managed load balancers, IP addresses, network packet filtering, and target health checking. The service controller interacts with your cloud provider's APIs to set up load balancers and other infrastructure components when you declare a Service resource that requires them.

Authorization

This section breaks down the access that the cloud controller manager requires on various API objects, in order to perform its operations.

Node controller

The Node controller only works with Node objects. It requires full access to read and modify Node objects.

`v1/Node`：

- get
- list
- create
- update
- patch
- watch
- delete

Route controller

The route controller listens to Node object creation and configures routes appropriately. It requires Get access to Node objects.

`v1/Node`：

- get

Service controller

The service controller watches for Service object **create**, **update** and **delete** events and then configures Endpoints for those Services appropriately (for EndpointSlices, the kube-controller-manager manages these on demand).

To access Services, it requires **list**, and **watch** access. To update Services, it requires **patch** and **update** access.

To set up Endpoints resources for the Services, it requires access to **create**, **list**, **get**, **watch**, and **update**.

`v1/Service`：

- list
- get
- watch
- patch
- update

Others

The implementation of the core of the cloud controller manager requires access to create Event objects, and to ensure secure operation, it requires access to create ServiceAccounts.

`v1/Event`：

- create
- patch
- update

`v1/ServiceAccount`：

- create

The RBAC ClusterRole for the cloud controller manager looks like:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cloud-controller-manager
rules:
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - '*'
- apiGroups:
  - ""
  resources:
  - nodes/status
  verbs:
  - patch
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - patch
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - persistentvolumes
  verbs:
  - get
  - list
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - create
  - get
  - list
  - watch
  - update
```

What's next

- Cloud Controller Manager Administration has instructions on running and managing the cloud controller manager.

- To upgrade a HA control plane to use the cloud controller manager, see Migrate Replicated Control Plane To Use Cloud Controller Manager.

- Want to know how to implement your own cloud controller manager, or extend an existing project?

    - The cloud controller manager uses Go interfaces, specifically, `CloudProvider` interface defined in `cloud.go` from kubernetes/cloud-provider to allow implementations from any cloud to be plugged in.
    - The implementation of the shared controllers highlighted in this document (Node, Route, and Service), and some scaffolding along with the shared cloudprovider interface, is part of the Kubernetes core. Implementations specific to cloud providers are outside the core of Kubernetes and implement the `CloudProvider` interface.
    - For more information about developing plugins, see Developing Cloud Controller Manager.

6 - About cgroup v2

On Linux, control groups constrain resources that are allocated to processes.

The kubelet and the underlying container runtime need to interface with cgroups to enforce resource management for pods and containers which includes cpu/memory requests and limits for containerized workloads.

There are two versions of cgroups in Linux: cgroup v1 and cgroup v2. cgroup v2 is the new generation of the `cgroup` API.

What is cgroup v2?

FEATURE STATE: `Kubernetes v1.25 [stable]`

cgroup v2 is the next version of the Linux `cgroup` API. cgroup v2 provides a unified control system with enhanced resource management capabilities.

cgroup v2 offers several improvements over cgroup v1, such as the following:

- Single unified hierarchy design in API
- Safer sub-tree delegation to containers
- Newer features like Pressure Stall Information
- Enhanced resource allocation management and isolation across multiple resources
  - Unified accounting for different types of memory allocations (network memory, kernel memory, etc)
  - Accounting for non-immediate resource changes such as page cache write backs

Some Kubernetes features exclusively use cgroup v2 for enhanced resource management and isolation. For example, the MemoryQoS feature improves memory QoS and relies on cgroup v2 primitives.

Using cgroup v2

The recommended way to use cgroup v2 is to use a Linux distribution that enables and uses cgroup v2 by default.

To check if your distribution uses cgroup v2, refer to Identify cgroup version on Linux nodes.

Requirements

cgroup v2 has the following requirements:

- OS distribution enables cgroup v2
- Linux Kernel version is 5.8 or later
- Container runtime supports cgroup v2. For example:
  - containerd v1.4 and later
  - cri-o v1.20 and later
- The kubelet and the container runtime are configured to use the systemd cgroup driver

Linux Distribution cgroup v2 support

For a list of Linux distributions that use cgroup v2, refer to the cgroup v2 documentation

- Container Optimized OS (since M97)
- Ubuntu (since 21.10, 22.04+ recommended)
- Debian GNU/Linux (since Debian 11 bullseye)
- Fedora (since 31)
- Arch Linux (since April 2021)
- RHEL and RHEL-like distributions (since 9)

To check if your distribution is using cgroup v2, refer to your distribution's documentation or follow the instructions in Identify the cgroup version on Linux nodes.

You can also enable cgroup v2 manually on your Linux distribution by modifying the kernel cmdline boot arguments. If your distribution uses GRUB, `systemd.unified_cgroup_hierarchy=1` should be added in `GRUB_CMDLINE_LINUX` under `/etc/default/grub`, followed by `sudo update-grub`. However, the recommended approach is to use a distribution that already enables cgroup v2 by default.

Migrating to cgroup v2

To migrate to cgroup v2, ensure that you meet the requirements, then upgrade to a kernel version that enables cgroup v2 by default.

The kubelet automatically detects that the OS is running on cgroup v2 and performs accordingly with no additional configuration required.

There should not be any noticeable difference in the user experience when switching to cgroup v2, unless users are accessing the cgroup file system directly, either on the node or from within the containers.

cgroup v2 uses a different API than cgroup v1, so if there are any applications that directly access the cgroup file system, they need to be updated to newer versions that support cgroup v2. For example:

- Some third-party monitoring and security agents may depend on the cgroup filesystem. Update these agents to versions that support cgroup v2.
- If you run cAdvisor as a stand-alone DaemonSet for monitoring pods and containers, update it to v0.43.0 or later.
- If you deploy Java applications, prefer to use versions which fully support cgroup v2:
  - OpenJDK / HotSpot: jdk8u372, 11.0.16, 15 and later
  - IBM Semeru Runtimes: jdk8u345-b01, 11.0.16.0, 17.0.4.0, 18.0.2.0 and later
  - IBM Java: 8.0.7.15 and later
- If you are using the uber-go/automaxprocs package, make sure the version you use is v1.5.1 or higher.

Identify the cgroup version on Linux Nodes

The cgroup version depends on the Linux distribution being used and the default cgroup version configured on the OS. To check which cgroup version your distribution uses, run the `stat -fc %T /sys/fs/cgroup/` command on the node:

```
stat -fc %T /sys/fs/cgroup/
```

For cgroup v2, the output is `cgroup2fs`.

For cgroup v1, the output is `tmpfs`.

What's next

- Learn more about cgroups
- Learn more about container runtime
- Learn more about cgroup drivers

7 - Container Runtime Interface (CRI)

The CRI is a plugin interface which enables the kubelet to use a wide variety of container runtimes, without having a need to recompile the cluster components.

You need a working container runtime on each Node in your cluster, so that the kubelet can launch Pods and their containers.

The Container Runtime Interface (CRI) is the main protocol for the communication between the kubelet and Container Runtime.

The Kubernetes Container Runtime Interface (CRI) defines the main gRPC protocol for the communication between the cluster components kubelet and container runtime.

The API

FEATURE STATE: `Kubernetes v1.23 [stable]`

The kubelet acts as a client when connecting to the container runtime via gRPC. The runtime and image service endpoints have to be available in the container runtime, which can be configured separately within the kubelet by using the `--image-service-endpoint` command line flags.

For Kubernetes v1.27, the kubelet prefers to use CRI `v1`. If a container runtime does not support `v1` of the CRI, then the kubelet tries to negotiate any older supported version. The v1.27 kubelet can also negotiate CRI `v1alpha2`, but this version is considered as deprecated. If the kubelet cannot negotiate a supported CRI version, the kubelet gives up and doesn't register as a node.

Upgrading

When upgrading Kubernetes, the kubelet tries to automatically select the latest CRI version on restart of the component. If that fails, then the fallback will take place as mentioned above. If a gRPC re-dial was required because the container runtime has been upgraded, then the container runtime must also support the initially selected version or the redial is expected to fail. This requires a restart of the kubelet.

What's next

- Learn more about the CRI protocol definition

8 - Garbage Collection

Garbage collection is a collective term for the various mechanisms Kubernetes uses to clean up cluster resources. This allows the clean up of resources like the following:

- Terminated pods
- Completed Jobs
- Objects without owner references
- Unused containers and container images
- Dynamically provisioned PersistentVolumes with a StorageClass reclaim policy of Delete
- Stale or expired CertificateSigningRequests (CSRs)
- Nodes deleted in the following scenarios:
  - On a cloud when the cluster uses a cloud controller manager
  - On-premises when the cluster uses an addon similar to a cloud controller manager
- Node Lease objects

Owners and dependents

Many objects in Kubernetes link to each other through *owner references*. Owner references tell the control plane which objects are dependent on others. Kubernetes uses owner references to give the control plane, and other API clients, the opportunity to clean up related resources before deleting an object. In most cases, Kubernetes manages owner references automatically.

Ownership is different from the labels and selectors mechanism that some resources also use. For example, consider a Service that creates `EndpointSlice` objects. The Service uses *labels* to allow the control plane to determine which `EndpointSlice` objects are used for that Service. In addition to the labels, each `EndpointSlice` that is managed on behalf of a Service has an owner reference. Owner references help different parts of Kubernetes avoid interfering with objects they don't control.

> **Note:**
> Cross-namespace owner references are disallowed by design. Namespaced dependents can specify cluster-scoped or namespaced owners. A namespaced owner **must** exist in the same namespace as the dependent. If it does not, the owner reference is treated as absent, and the dependent is subject to deletion once all owners are verified absent.
>
> Cluster-scoped dependents can only specify cluster-scoped owners. In v1.20+, if a cluster-scoped dependent specifies a namespaced kind as an owner, it is treated as having an unresolvable owner reference, and is not able to be garbage collected.
>
> In v1.20+, if the garbage collector detects an invalid cross-namespace `ownerReference`, or a cluster-scoped dependent with an `ownerReference` referencing a namespaced kind, a warning Event with a reason of `OwnerRefInvalidNamespace` and an `involvedObject` of the invalid dependent is reported. You can check for that kind of Event by running `kubectl get events -A --field-selector=reason=OwnerRefInvalidNamespace`.

Cascading deletion

Kubernetes checks for and deletes objects that no longer have owner references, like the pods left behind when you delete a ReplicaSet. When you delete an object, you can control whether Kubernetes deletes the object's dependents automatically, in a process called *cascading deletion*. There are two types of cascading deletion, as follows:

- Foreground cascading deletion
- Background cascading deletion

You can also control how and when garbage collection deletes resources that have owner references using Kubernetes finalizers.

Foreground cascading deletion

In foreground cascading deletion, the owner object you're deleting first enters a *deletion in progress* state. In this state, the following happens to the owner object:

- The Kubernetes API server sets the object's `metadata.deletionTimestamp` field to the time the object was marked for deletion.
- The Kubernetes API server also sets the `metadata.finalizers` field to `foregroundDeletion`.
- The object remains visible through the Kubernetes API until the deletion process is complete.

After the owner object enters the deletion in progress state, the controller deletes the dependents. After deleting all the dependent objects, the controller deletes the owner object. At this point, the object is no longer visible in the Kubernetes API.

During foreground cascading deletion, the only dependents that block owner deletion are those that have the `ownerReference.blockOwnerDeletion=true` field. See Use foreground cascading deletion to learn more.

Background cascading deletion

In background cascading deletion, the Kubernetes API server deletes the owner object immediately and the controller cleans up the dependent objects in the background. By default, Kubernetes uses background cascading deletion unless you manually use foreground deletion or choose to orphan the dependent objects.

See Use background cascading deletion to learn more.

Orphaned dependents

When Kubernetes deletes an owner object, the dependents left behind are called *orphan* objects. By default, Kubernetes deletes dependent objects. To learn how to override this behaviour, see Delete owner objects and orphan dependents.

Garbage collection of unused containers and images

The kubelet performs garbage collection on unused images every five minutes and on unused containers every minute. You should avoid using external garbage collection tools, as these can break the kubelet behavior and remove containers that should exist.

To configure options for unused container and image garbage collection, tune the kubelet using a configuration file and change the parameters related to garbage collection using the `KubeletConfiguration` resource type.

Container image lifecycle

Kubernetes manages the lifecycle of all images through its *image manager*, which is part of the kubelet, with the cooperation of cadvisor. The kubelet considers the following disk usage limits when making garbage collection decisions:

- `HighThresholdPercent`
- `LowThresholdPercent`

Disk usage above the configured `HighThresholdPercent` value triggers garbage collection, which deletes images in order based on the last time they were used, starting with the oldest first. The kubelet deletes images until disk usage reaches the `LowThresholdPercent` value.

Container garbage collection

The kubelet garbage collects unused containers based on the following variables, which you can define:

- `MinAge` : the minimum age at which the kubelet can garbage collect a container. Disable by setting to `0` .
- `MaxPerPodContainer` : the maximum number of dead containers each Pod can have. Disable by setting to less than `0` .
- `MaxContainers` : the maximum number of dead containers the cluster can have. Disable by setting to less than `0` .

In addition to these variables, the kubelet garbage collects unidentified and deleted containers, typically starting with the oldest first.

`MaxPerPodContainer` and `MaxContainers` may potentially conflict with each other in situations where retaining the maximum number of containers per Pod (`MaxPerPodContainer`) would go outside the allowable total of global dead containers (`MaxContainers`). In this situation, the kubelet adjusts `MaxPerPodContainer` to address the conflict. A worst-case scenario would be to downgrade `MaxPerPodContainer` to `1` and evict the oldest containers. Additionally, containers owned by pods that have been deleted are removed once they are older than `MinAge` .

> **Note:** The kubelet only garbage collects the containers it manages.

Configuring garbage collection

You can tune garbage collection of resources by configuring options specific to the controllers managing those resources. The following pages show you how to configure garbage collection:

- Configuring cascading deletion of Kubernetes objects
- Configuring cleanup of finished Jobs

What's next

- Learn more about ownership of Kubernetes objects.
- Learn more about Kubernetes finalizers.
- Learn about the TTL controller that cleans up finished Jobs.