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

Kubernetes components

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 load balancers for those Services appropriately.

To access Services, it requires **list**, and **watch** access. To update Services, it requires **patch** and **update** access to the `status` subresource.

`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:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: cloud-controller-managerrules:- apiGroups:  - ""  resources:  - events  verbs:  - create  - patc
```

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.

Mixed Version Proxy

FEATURE STATE: `Kubernetes v1.28 [alpha]` (enabled by default: false)

Kubernetes 1.34 includes an alpha feature that lets an API Server proxy a resource requests to other *peer* API servers. This is useful when there are multiple API servers running different versions of Kubernetes in one cluster (for example, during a long-lived rollout to a new release of Kubernetes).

This enables cluster administrators to configure highly available clusters that can be upgraded more safely, by directing resource requests (made during the upgrade) to the correct kube-apiserver. That proxying prevents users from seeing unexpected 404 Not Found errors that stem from the upgrade process.

This mechanism is called the *Mixed Version Proxy*.

Enabling the Mixed Version Proxy

Ensure that `UnknownVersionInteroperabilityProxy` feature gate is enabled when you start the API Server:

```
kube-apiserver \
--feature-gates=UnknownVersionInteroperabilityProxy=true \# required command line arguments for this feature--peer-ca-file=<path t
```

Proxy transport and authentication between API servers

- The source kube-apiserver reuses the existing APIserver client authentication flags `--proxy-client-cert-file` and `--proxy-client-key-file` to present its identity that will be verified by its peer (the destination kube-apiserver). The destination API server verifies that peer connection based on the configuration you specify using the `--requestheader-client-ca-file` command line argument.

- To authenticate the destination server's serving certs, you must configure a certificate authority bundle by specifying the `--peer-ca-file` command line argument to the **source** API server.

Configuration for peer API server connectivity

To set the network location of a kube-apiserver that peers will use to proxy requests, use the `--peer-advertise-ip` and `--peer-advertise-port` command line arguments to kube-apiserver or specify these fields in the API server configuration file. If these flags are unspecified, peers will use the value from either `--advertise-address` or `--bind-address` command line argument to the kube-apiserver. If those too, are unset, the host's default interface is used.

Mixed version proxying

When you enable mixed version proxying, the aggregation layer loads a special filter that does the following:

- When a resource request reaches an API server that cannot serve that API (either because it is at a version pre-dating the introduction of the API or the API is turned off on the API server) the API server attempts to send the request to a peer API server that can serve the requested API. It does so by identifying API groups / versions / resources that the local server doesn't recognise, and tries to proxy those requests to a peer API server that is capable of handling the request.
- If the peer API server fails to respond, the *source* API server responds with 503 ("Service Unavailable") error.

How it works under the hood

When an API Server receives a resource request, it first checks which API servers can serve the requested resource. This check happens using the internal `StorageVersion` API.

- If the resource is known to the API server that received the request (for example, `GET /api/v1/pods/some-pod`), the request is handled locally.

- If there is no internal `StorageVersion` object found for the requested resource (for example, `GET /my-api/v1/my-resource`) and the configured APIService specifies proxying to an extension API server, that proxying happens following the usual flow for extension APIs.

- If a valid internal `StorageVersion` object is found for the requested resource (for example, `GET /batch/v1/jobs`) and the API server trying to handle the request (the *handling API server*) has the `batch` API disabled, then the *handling API server* fetches the peer API servers that do serve the relevant API group / version / resource (`api/v1/batch` in this case) using the information in the fetched `StorageVersion` object. The *handling API server* then proxies the request to one of the matching peer kube-apiservers that are aware of the requested resource.

    - If there is no peer known for that API group / version / resource, the handling API server passes the request to its own handler chain which should eventually return a 404 ("Not Found") response.

    - If the handling API server has identified and selected a peer API server, but that peer fails to respond (for reasons such as network connectivity issues, or a data race between the request being received and a controller registering the peer's info into the control plane), then the handling API server responds with a 503 ("Service Unavailable") error.

Cluster Architecture

The architectural concepts behind Kubernetes.

A Kubernetes cluster consists of a control plane plus a set of worker machines, called nodes, that run containerized applications. Every cluster needs at least one worker node in order to run Pods.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

This document outlines the various components you need to have for a complete and working Kubernetes cluster.

The control plane (kube-apiserver, etcd, kube-controller-manager, kube-scheduler) and several nodes. Each node is running a kubelet and kube-proxy.

Figure 1. Kubernetes cluster components.

▶ About this architecture

Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a Deployment's `replicas` field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, setup scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple machines.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for the data.

You can find in-depth information about etcd in the official documentation.

kube-scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

Control plane component that runs controller processes.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

There are many different types of controllers. Some examples of them are:

- Node controller: Responsible for noticing and responding when nodes go down.
- Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
- ServiceAccount controller: Create default ServiceAccounts for new namespaces.

The above is not an exhaustive list.

cloud-controller-manager

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.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

- Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route controller: For setting up routes in the underlying cloud infrastructure
- Service controller: For creating, updating and deleting cloud provider load balancers

---

Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy (optional)

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

If you use a network plugin that implements packet forwarding for Services by itself, and providing equivalent behavior to kube-proxy, then you do not need to run kube-proxy on the nodes in your cluster.

Container runtime

A fundamental component that empowers Kubernetes to run containers effectively. It is responsible for managing the execution and lifecycle of containers within the Kubernetes environment.

Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

Addons

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the `kube-system` namespace.

Selected addons are described below; for an extended list of available addons, please see Addons.

DNS

While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container resource monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

A cluster-level logging mechanism is responsible for saving container logs to a central log store with a search/browsing interface.

Network plugins

Network plugins are software components that implement the container network interface (CNI) specification. They are responsible for allocating IP addresses to pods and enabling them to communicate with each other within the cluster.

Architecture variations

While the core components of Kubernetes remain consistent, the way they are deployed and managed can vary. Understanding these variations is crucial for designing and maintaining Kubernetes clusters that meet specific operational needs.

Control plane deployment options

The control plane components can be deployed in several ways:

Traditional deployment
: Control plane components run directly on dedicated machines or VMs, often managed as systemd services.

Static Pods
: Control plane components are deployed as static Pods, managed by the kubelet on specific nodes. This is a common approach used by tools like kubeadm.

Self-hosted
: The control plane runs as Pods within the Kubernetes cluster itself, managed by Deployments and StatefulSets or other Kubernetes primitives.

Managed Kubernetes services
: Cloud providers often abstract away the control plane, managing its components as part of their service offering.

Workload placement considerations

The placement of workloads, including the control plane components, can vary based on cluster size, performance requirements, and operational policies:

- In smaller or development clusters, control plane components and user workloads might run on the same nodes.
- Larger production clusters often dedicate specific nodes to control plane components, separating them from user workloads.
- Some organizations run critical add-ons or monitoring tools on control plane nodes.

Cluster management tools

Tools like kubeadm, kops, and Kubespray offer different approaches to deploying and managing clusters, each with its own method of component layout and management.

The flexibility of Kubernetes architecture allows organizations to tailor their clusters to specific needs, balancing factors such as operational complexity, performance, and management overhead.

Customization and extensibility

Kubernetes architecture allows for significant customization:

- Custom schedulers can be deployed to work alongside the default Kubernetes scheduler or to replace it entirely.
- API servers can be extended with CustomResourceDefinitions and API Aggregation.
- Cloud providers can integrate deeply with Kubernetes using the cloud-controller-manager.

The flexibility of Kubernetes architecture allows organizations to tailor their clusters to specific needs, balancing factors such as operational complexity, performance, and management overhead.

What's next

Learn more about the following:

- Nodes and their communication with the control plane.
- Kubernetes controllers.
- kube-scheduler which is the default scheduler for Kubernetes.
- Etcd's official documentation.
- Several container runtimes in Kubernetes.
- Integrating with cloud providers using cloud-controller-manager.
- kubectl commands.

Kubernetes Self-Healing

Kubernetes is designed with self-healing capabilities that help maintain the health and availability of workloads. It automatically replaces failed containers, reschedules workloads when nodes become unavailable, and ensures that the desired state of the system is maintained.

Self-Healing capabilities

- **Container-level restarts:** If a container inside a Pod fails, Kubernetes restarts it based on the `restartPolicy`.

- **Replica replacement:** If a Pod in a Deployment or StatefulSet fails, Kubernetes creates a replacement Pod to maintain the specified number of replicas. If a Pod fails that is part of a DaemonSet fails, the control plane creates a replacement Pod to run on the same node.

- **Persistent storage recovery:** If a node is running a Pod with a PersistentVolume (PV) attached, and the node fails, Kubernetes can reattach the volume to a new Pod on a different node.

- **Load balancing for Services:** If a Pod behind a Service fails, Kubernetes automatically removes it from the Service's endpoints to route traffic only to healthy Pods.

Here are some of the key components that provide Kubernetes self-healing:

- **kubelet:** Ensures that containers are running, and restarts those that fail.

- **ReplicaSet, StatefulSet and DaemonSet controller:** Maintains the desired number of Pod replicas.

- **PersistentVolume controller:** Manages volume attachment and detachment for stateful workloads.

Considerations

- **Storage Failures:** If a persistent volume becomes unavailable, recovery steps may be required.

- **Application Errors:** Kubernetes can restart containers, but underlying application issues must be addressed separately.

What's next

- Read more about Pods
- Learn about Kubernetes Controllers
- Explore PersistentVolumes
- Read about node autoscaling. Node autoscaling also provides automatic healing if or when nodes fail in your cluster.

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 Kubernetes extension patterns and the sample-controller repository.

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

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 dependents it knows about. After deleting all the dependent objects it knows about, 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 and are in the garbage collection controller cache. The garbage collection controller cache may not contain objects whose resource type cannot be listed / watched successfully, or objects that are created concurrent with deletion of an owner object. 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 garbage collector controller (custom or default) cleans up the dependent objects in the background. If a finalizer exists, it ensures that objects are not deleted until all necessary clean-up tasks are completed. 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.

Garbage collection for unused container images

FEATURE STATE: `Kubernetes v1.30 [beta]` (enabled by default: true)

As a beta feature, you can specify the maximum time a local image can be unused for, regardless of disk usage. This is a kubelet setting that you configure for each node.

To configure the setting, you need to set a value for the `imageMaximumGCAge` field in the kubelet configuration file.

The value is specified as a Kubernetes duration. See duration in the glossary for more details.

For example, you can set the configuration field to `12h45m`, which means 12 hours and 45 minutes.

Note:

This feature does not track image usage across kubelet restarts. If the kubelet is restarted, the tracked image age is reset, causing the kubelet to wait the full `imageMaximumGCAge` duration before qualifying images for garbage collection based on image age.

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.

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: 8.0.382.0, 11.0.20.0, 17.0.8.0, and later
  - IBM Java: 8.0.8.6 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

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.

Read coordinated leader election to learn about how Kubernetes builds on the Lease API to select which component instance acts as leader.

API server identity

FEATURE STATE: `Kubernetes v1.26 [beta]` (enabled by default: true)

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 `apiserver-<sha256-hash>`. Alternatively you can use the label selector `apiserver.kubernetes.io/identity=kube-apiserver`:

```
kubectl -n kube-system get lease -l apiserver.kubernetes.io/identity=kube-apiserver
```

```
NAME                                      HOLDER                                                                    AGE
apiserver-07a5ea9b9b072c4a5f3d1c3702      apiserver-07a5ea9b9b072c4a5f3d1c3702_0c8914f7-0f35-440e-8676-7844977d3a05   5m33s
apiserver-7be9e061c59d368b3ddaf1376e      apiserver-7be9e061c59d368b3ddaf1376e_84f2a85d-37c1-4b14-b6b9-603e62e4896f   4m23s
apiserver-1dfef752bcb36637d2763d1868      apiserver-1dfef752bcb36637d2763d1868_c5ffa286-8a9a-45d4-91e7-61118ed58d2e   4m43s
```

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-apiserver by checking the value of the `kubernetes.io/hostname` label:

```
kubectl -n kube-system get lease apiserver-07a5ea9b9b072c4a5f3d1c3702 -o yaml
```

```
apiVersion: coordination.k8s.io/v1
kind: Leasemetadata:  creationTimestamp: "2023-07-02T13:16:48Z"  labels:    apiserver.kubernetes.io/identity: kube-apiserver    kul
```

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.

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 is being restarted with a new set of `--node-labels`, but the same Node name is used, the change will not take effect, as labels are only set (or modified) upon Node registration with the API server.

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 set optional node role(s) for nodes by adding one or more `node-role.kubernetes.io/<role>: <role>` labels to the node where characters of `<role>` are limited by the syntax rules for labels.

Kubernetes ignores the label value for node roles; by convention, you can set it to the same string you used for the node role in the label key.

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>
```

See Node Status for more details.

Node 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.

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 v1.27 [stable]` (enabled by default: true)

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.

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.
- Graceful/non-graceful node shutdown.
- Node autoscaling to manage the number and size of nodes in your cluster.
- Taints and Tolerations.
- Node Resource Managers.
- Resource Management for Windows nodes.