# Local Files And Paths Used By The Kubelet

The kubelet is mostly a stateless process running on a Kubernetes node. This document outlines files that kubelet reads and writes.

**Note:**

This document is for informational purpose and not describing any guaranteed behaviors or APIs. It lists resources used by the kubelet, which is an implementation detail and a subject to change at any release.

The kubelet typically uses the control plane as the source of truth on what needs to run on the Node, and the container runtime to retrieve the current state of containers. So long as you provide a *kubeconfig* (API client configuration) to the kubelet, the kubelet does connect to your control plane; otherwise the node operates in *standalone mode*.

On Linux nodes, the kubelet also relies on reading cgroups and various system files to collect metrics.

On Windows nodes, the kubelet collects metrics via a different mechanism that does not rely on paths.

There are also a few other files that are used by the kubelet as well, as kubelet communicates using local Unix-domain sockets. Some are sockets that the kubelet listens on, and for other sockets the kubelet discovers them and then connects as a client.

**Note:**

This page lists paths as Linux paths, which map to the Windows paths by adding a root disk `C:\` in place of `/` (unless specified otherwise). For example, `/var/lib/kubelet/device-plugins` maps to `C:\var\lib\kubelet\device-plugins`.

## Configuration

### Kubelet configuration files

The path to the kubelet configuration file can be configured using the command line argument `--config`. The kubelet also supports drop-in configuration files to enhance configuration.

### Certificates

Certificates and private keys are typically located at `/var/lib/kubelet/pki`, but can be configured using the `--cert-dir` kubelet command line argument. Names of certificate files are also configurable.

### Manifests

Manifests for static pods are typically located in `/etc/kubernetes/manifests`. Location can be configured using the `staticPodPath` kubelet configuration option.

### Systemd unit settings

When kubelet is running as a systemd unit, some kubelet configuration may be declared in systemd unit settings file. Typically it includes:

- command line arguments to run kubelet
- environment variables, used by kubelet or configuring golang runtime

## State

### Checkpoint files for resource managers

All resource managers keep the mapping of Pods to allocated resources in state files. State files are located in the kubelet's base directory, also termed the *root directory* (but not the same as `/`, the node root directory). You can configure the base directory for the kubelet using the kubelet command line argument `--root-dir`.

Names of files:

- `memory_manager_state` for the Memory Manager
- `cpu_manager_state` for the CPU Manager
- `dra_manager_state` for DRA

### Checkpoint file for device manager

Device manager creates checkpoints in the same directory with socket files: `/var/lib/kubelet/device-plugins/`. The name of a checkpoint file is `kubelet_internal_checkpoint` for Device Manager

### Pod resource checkpoints

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: true)

If a node has enabled the `InPlacePodVerticalScaling`feature gate, the kubelet stores a local record of *allocated* and *actuated* Pod resources. See Resize CPU and Memory Resources assigned to Containers for more details on how these records are used.

Names of files:

- `allocated_pods_state` records the resources allocated to each pod running on the node
- `actuated_pods_state` records the resources that have been accepted by the runtime for each pod pod running on the node

The files are located within the kubelet base directory (`/var/lib/kubelet` by default on Linux; configurable using `--root-dir`).

### Container runtime

Kubelet communicates with the container runtime using socket configured via the configuration parameters:

- `containerRuntimeEndpoint` for runtime operations
- `imageServiceEndpoint` for image management operations

The actual values of those endpoints depend on the container runtime being used.

### Device plugins

The kubelet exposes a socket at the path `/var/lib/kubelet/device-plugins/kubelet.sock` for various Device Plugins to register.

When a device plugin registers itself, it provides its socket path for the kubelet to connect.

The device plugin socket should be in the directory `device-plugins` within the kubelet base directory. On a typical Linux node, this means `/var/lib/kubelet/device-plugins`.

### Pod resources API

Pod Resources API will be exposed at the path `/var/lib/kubelet/pod-resources`.

### DRA, CSI, and Device plugins

The kubelet looks for socket files created by device plugins managed via DRA, device manager, or storage plugins, and then attempts to connect to these sockets. The directory that the kubelet looks in is `plugins_registry` within the kubelet base directory, so on a typical Linux node this means `/var/lib/kubelet/plugins_registry`.

Note, for the device plugins there are two alternative registration mechanisms Only one should be used for a given plugin.

The types of plugins that can place socket files into that directory are:

- CSI plugins
- DRA plugins
- Device Manager plugins

(typically `/var/lib/kubelet/plugins_registry`).

### Graceful node shutdown

FEATURE STATE: `Kubernetes v1.21 [beta]` (enabled by default: true)

Graceful node shutdown stores state locally at `/var/lib/kubelet/graceful_node_shutdown_state`.

### Image Pull Records

FEATURE STATE: `Kubernetes v1.33 [alpha]` (enabled by default: false)

The kubelet stores records of attempted and successful image pulls, and uses it to verify that the image was previously successfully pulled with the same credentials.

These records are cached as files in the `image_registry` directory within the kubelet base directory. On a typical Linux node, this means `/var/lib/kubelet/image_manager`. There are two subdirectories to `image_manager`:

- `pulling` - stores records about images the Kubelet is attempting to pull.
- `pulled` - stores records about images that were successfully pulled by the Kubelet, along with metadata about the credentials used for the pulls.

See Ensure Image Pull Credential Verification for details.

## Security profiles & configuration

### Seccomp

Seccomp profile files referenced from Pods should be placed in `/var/lib/kubelet/seccomp`. See the seccomp reference for details.

### AppArmor

The kubelet does not load or refer to AppArmor profiles by a Kubernetes-specific path. AppArmor profiles are loaded via the node operating system rather then referenced by their path.

## Locking

FEATURE STATE: `Kubernetes v1.2 [alpha]`

A lock file for the kubelet; typically `/var/run/kubelet.lock`. The kubelet uses this to ensure that two different kubelets don't try to run in conflict with each other. You can configure the path to the lock file using the the `--lock-file` kubelet command line argument.

If two kubelets on the same node use a different value for the lock file path, they will not be able to detect a conflict when both are running.

## What's next

- Learn about the kubelet command line arguments.
- Review the Kubelet Configuration (v1beta1) reference

---

# Ports and Protocols

When running Kubernetes in an environment with strict network boundaries, such as on-premises datacenter with physical network firewalls or Virtual Networks in Public Cloud, it is useful to be aware of the ports and protocols used by Kubernetes components.

## Control plane

| Protocol | Direction | Port Range | Purpose | Used By |
|---|---|---|---|---|
| TCP | Inbound | 6443 | Kubernetes API server | All |
| TCP | Inbound | 2379-2380 | etcd server client API | kube-apiserver, etcd |
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 10259 | kube-scheduler | Self |
| TCP | Inbound | 10257 | kube-controller-manager | Self |

Although etcd ports are included in control plane section, you can also host your own etcd cluster externally or on custom ports.

## Worker node(s)

| Protocol | Direction | Port Range | Purpose | Used By |
|---|---|---|---|---|
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 10256 | kube-proxy | Self, Load balancers |
| TCP | Inbound | 30000-32767 | NodePort Services† | All |
| UDP | Inbound | 30000-32767 | NodePort Services† | All |

† Default port range for NodePort Services.

All default port numbers can be overridden. When custom ports are used those ports need to be open instead of defaults mentioned here.

One common example is API server port that is sometimes switched to 443. Alternatively, the default port is kept as is and API server is put behind a load balancer that listens on 443 and routes the requests to API server on the default port.

---

# Seccomp and Kubernetes

Seccomp stands for secure computing mode and has been a feature of the Linux kernel since version 2.6.12. It can be used to sandbox the privileges of a process, restricting the calls it is able to make from userspace into the kernel. Kubernetes lets you automatically apply seccomp profiles loaded onto a node to your Pods and containers.

## Seccomp fields

FEATURE STATE: `Kubernetes v1.19 [stable]`

There are four ways to specify a seccomp profile for a pod:

- for the whole Pod using `spec.securityContext.seccompProfile`
- for a single container using `spec.containers[*].securityContext.seccompProfile`
- for an (restartable / sidecar) init container using `spec.initContainers[*].securityContext.seccompProfile`
- for an ephemeral container using `spec.ephemeralContainers[*].securityContext.seccompProfile`

pods/security/seccomp/fields.yaml  Copy pods/security/seccomp/fields.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: podspec:  securityContext:    seccompProfile:      type: Unconfined  ephemeralContainers:  - name: ephem
```

The Pod in the example above runs as `Unconfined`, while the `ephemeral-container` and `init-container` specifically defines `RuntimeDefault`. If the ephemeral or init container would not have set the `securityContext.seccompProfile` field explicitly, then the value would be inherited from the Pod. The same applies to the container, which runs a `Localhost` profile `my-profile.json`.

Generally speaking, fields from (ephemeral) containers have a higher priority than the Pod level value, while containers which do not set the seccomp field inherit the profile from the Pod.

**Note:**

It is not possible to apply a seccomp profile to a Pod or container running with `privileged: true` set in the container's `securityContext`. Privileged containers always run as `Unconfined`.

The following values are possible for the `seccompProfile.type`:

`Unconfined`
: The workload runs without any seccomp restrictions.

`RuntimeDefault`
: A default seccomp profile defined by the container runtime is applied. The default profiles aim to provide a strong set of security defaults while preserving the functionality of the workload. It is possible that the default profiles differ between container runtimes and their release versions, for example when comparing those from CRI-O and containerd.

`Localhost`
: The `localhostProfile` will be applied, which has to be available on the node disk (on Linux it's `/var/lib/kubelet/seccomp`). The availability of the seccomp profile is verified by the container runtime on container creation. If the profile does not exist, then the container creation will fail with a `CreateContainerError`.

### `Localhost` profiles

Seccomp profiles are JSON files following the scheme defined by the OCI runtime specification. A profile basically defines actions based on matched syscalls, but also allows to pass specific values as arguments to syscalls. For example:

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "defaultErrnoRet": 38,
  "syscalls": [
    {
      "names": [
        "adjtimex",
        "alarm",
        "bind",
        "waitid",
        "waitpid",
        "write",
        "writev"
      ],
      "action": "SCMP_ACT_ALLOW"
    }
  ]
}
```

The `defaultAction` in the profile above is defined as `SCMP_ACT_ERRNO` and will return as fallback to the actions defined in `syscalls`. The error is defined as code `38` via the `defaultErrnoRet` field.

The following actions are generally possible:

`SCMP_ACT_ERRNO`
: Return the specified error code.

`SCMP_ACT_ALLOW`
: Allow the syscall to be executed.

`SCMP_ACT_KILL_PROCESS`
: Kill the process.

`SCMP_ACT_KILL_THREAD` and `SCMP_ACT_KILL`
: Kill only the thread.

`SCMP_ACT_TRAP`
: Throw a `SIGSYS` signal.

`SCMP_ACT_NOTIFY` and `SECCOMP_RET_USER_NOTIF`.
: Notify the user space.

`SCMP_ACT_TRACE`
: Notify a tracing process with the specified value.

`SCMP_ACT_LOG`
: Allow the syscall to be executed after the action has been logged to syslog or auditd.

Some actions like `SCMP_ACT_NOTIFY` or `SECCOMP_RET_USER_NOTIF` may be not supported depending on the container runtime, OCI runtime or Linux kernel version being used. There may be also further limitations, for example that `SCMP_ACT_NOTIFY` cannot be used as `defaultAction` or for certain syscalls like `write`. All those limitations are defined by either the OCI runtime (runc, crun) or libseccomp.

The `syscalls` JSON array contains a list of objects referencing syscalls by their respective `names`. For example, the action `SCMP_ACT_ALLOW` can be used to create a whitelist of allowed syscalls as outlined in the example above. It would also be possible to define another list using the action `SCMP_ACT_ERRNO` but a different return (`errnoRet`) value.

It is also possible to specify the arguments (`args`) passed to certain syscalls. More information about those advanced use cases can be found in the OCI runtime spec and the Seccomp Linux kernel documentation.

## Further reading

- Restrict a Container's Syscalls with seccomp
- Pod Security Standards

---

# Node Status

The status of a node in Kubernetes is a critical aspect of managing a Kubernetes cluster. In this article, we'll cover the basics of monitoring and maintaining node status to ensure a healthy and stable cluster.

## Node status fields

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 50 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 50 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.

---

# Linux Node Swap Behaviors

To allow Kubernetes workloads to use swap, on a Linux node, you must disable the kubelet's default behavior of failing when swap is detected, and specify memory-swap behavior as `LimitedSwap`:

The available choices for swap behavior are:

`NoSwap`
: (default) Workloads running as Pods on this node do not and cannot use swap. However, processes outside of Kubernetes' scope, such as system daemons (including the kubelet itself!) **can** utilize swap. This behavior is beneficial for protecting the node from system-level memory spikes, but it does not safeguard the workloads themselves from such spikes.

`LimitedSwap`
: Kubernetes workloads can utilize swap memory. The amount of swap available to a Pod is determined automatically.

To learn more, read swap memory management.

---

# Kubelet Systemd Watchdog

FEATURE STATE: `Kubernetes v1.32 [beta]` (enabled by default: true)

On Linux nodes, Kubernetes 1.34 supports integrating with systemd to allow the operating system supervisor to recover a failed kubelet. This integration is not enabled by default. It can be used as an alternative to periodically requesting the kubelet's `/healthz` endpoint for health checks. If the kubelet does not respond to the watchdog within the timeout period, the watchdog will kill the kubelet.

The systemd watchdog works by requiring the service to periodically send a *keep-alive* signal to the systemd process. If the signal is not received within a specified timeout period, the service is considered unresponsive and is terminated. The service can then be restarted according to the configuration.

## Configuration

Using the systemd watchdog requires configuring the `WatchdogSec` parameter in the `[Service]` section of the kubelet service unit file:

```
[Service]
WatchdogSec=30s
```

Setting `WatchdogSec=30s` indicates a service watchdog timeout of 30 seconds. Within the kubelet, the `sd_notify()` function is invoked, at intervals of \( WatchdogSec \div 2\). to send `WATCHDOG=1` (a keep-alive message). If the watchdog is not fed within the timeout period, the kubelet will be killed. Setting `Restart` to "always", "on-failure", "on-watchdog", or "on-abnormal" will ensure that the service is automatically restarted.

Some details about the systemd configuration:

1. If you set the systemd value for `WatchdogSec` to 0, or omit setting it, the systemd watchdog is not enabled for this unit.
2. The kubelet supports a minimum watchdog period of 1.0 seconds; this is to prevent the kubelet from being killed unexpectedly. You can set the value of `WatchdogSec` in a systemd unit definition to a period shorter than 1 second, but Kubernetes does not support any shorter interval. The timeout does not have to be a whole integer number of seconds.
3. The Kubernetes project suggests setting `WatchdogSec` to approximately a 15s period. Periods longer than 10 minutes are supported but explicitly **not** recommended.

### Example Configuration

```
[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=https://kubernetes.io/docs/home/
Wants=network-online.target
After=network-online.target

[Service]
```

```
ExecStart=/usr/bin/kubelet
# Configures the watchdog timeout
WatchdogSec=30s
Restart=on-failure
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target
```

### What's next

For more details about systemd configuration, refer to the systemd documentation

---

# Node Reference Information

This section contains the following reference topics about nodes:

- the kubelet's checkpoint API
- a list of Articles on dockershim Removal and on Using CRI-compatible Runtimes
- Kubelet Device Manager API Versions
- Node Labels Populated By The Kubelet
- Local Files And Paths Used By The Kubelet
- Node `.status` information
- Linux Node Swap Behaviors
- Seccomp information

You can also read node reference details from elsewhere in the Kubernetes documentation, including:

- Node Metrics Data.
- CRI Pod & Container Metrics.
- Understand Pressure Stall Information (PSI) Metrics.

---

# Protocols for Services

If you configure a Service, you can select from any network protocol that Kubernetes supports.

Kubernetes supports the following protocols with Services:

- `SCTP`
- `TCP` *(the default)*
- `UDP`

When you define a Service, you can also specify the application protocol that it uses.

This document details some special cases, all of them typically using TCP as a transport protocol:

- HTTP and HTTPS
- PROXY protocol
- TLS termination at the load balancer

## Supported protocols

There are 3 valid values for the `protocol` of a port for a Service:

#### `SCTP`

FEATURE STATE: `Kubernetes v1.20 [stable]`

When using a network plugin that supports SCTP traffic, you can use SCTP for most Services. For `type: LoadBalancer` Services, SCTP support depends on the cloud provider offering this facility. (Most do not).

SCTP is not supported on nodes that run Windows.

#### Support for multihomed SCTP associations

The support of multihomed SCTP associations requires that the CNI plugin can support the assignment of multiple interfaces and IP addresses to a Pod.

NAT for multihomed SCTP associations requires special logic in the corresponding kernel modules.

#### `TCP`

You can use TCP for any kind of Service, and it's the default network protocol.

#### `UDP`

You can use UDP for most Services. For `type: LoadBalancer` Services, UDP support depends on the cloud provider offering this facility.

## Special cases

### HTTP

If your cloud provider supports it, you can use a Service in LoadBalancer mode to configure a load balancer outside of your Kubernetes cluster, in a special mode where your cloud provider's load balancer implements HTTP / HTTPS reverse proxying, with traffic forwarded to the backend endpoints for that Service.

Typically, you set the protocol for the Service to `TCP` and add an annotation (usually specific to your cloud provider) that configures the load balancer to handle traffic at the HTTP level. This configuration might also include serving HTTPS (HTTP over TLS) and reverse-proxying plain HTTP to your workload.

**Note:**

You can also use an Ingress to expose HTTP/HTTPS Services.

You might additionally want to specify that the application protocol of the connection is `http` or `https`. Use `http` if the session from the load balancer to your workload is HTTP without TLS, and use `https` if the session from the load balancer to your workload uses TLS encryption.

### PROXY protocol

If your cloud provider supports it, you can use a Service set to `type: LoadBalancer` to configure a load balancer outside of Kubernetes itself, that will forward connections wrapped with the PROXY protocol.

The load balancer then sends an initial series of octets describing the incoming connection, similar to this example (PROXY protocol v1):

```
PROXY TCP4 192.0.2.202 10.0.42.7 12345 7\r\n
```

The data after the proxy protocol preamble are the original data from the client. When either side closes the connection, the load balancer also triggers a connection close and sends any remaining data where feasible.

Typically, you define a Service with the protocol to `TCP`. You also set an annotation, specific to your cloud provider, that configures the load balancer to wrap each incoming connection in the PROXY protocol.

### TLS

If your cloud provider supports it, you can use a Service set to `type: LoadBalancer` as a way to set up external reverse proxying, where the connection from client to load balancer is TLS encrypted and the load balancer is the TLS server peer. The connection from the load balancer to your workload can also be TLS, or might be plain text. The exact options available to you depend on your cloud provider or custom Service implementation.

Typically, you set the protocol to `TCP` and set an annotation (usually specific to your cloud provider) that configures the load balancer to act as a TLS server. You would configure the TLS identity (as server, and possibly also as a client that connects to your workload) using mechanisms that are specific to your cloud provider.

---

# Node Labels Populated By The Kubelet

Kubernetes nodes come pre-populated with a standard set of labels.

You can also set your own labels on nodes, either through the kubelet configuration or using the Kubernetes API.

## Preset labels

The preset labels that Kubernetes sets on nodes are:

- `kubernetes.io/arch`
- `kubernetes.io/hostname`
- `kubernetes.io/os`
- `node.kubernetes.io/instance-type` (if known to the kubelet – Kubernetes may not have this information to set the label)
- `topology.kubernetes.io/region` (if known to the kubelet – Kubernetes may not have this information to set the label)
- `topology.kubernetes.io/zone` (if known to the kubelet – Kubernetes may not have this information to set the label)

**Note:**

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

## What's next

- See Well-Known Labels, Annotations and Taints for a list of common labels.
- Learn how to add a label to a node.

---

# Linux Kernel Version Requirements

**Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Many features rely on specific kernel functionalities and have minimum kernel version requirements. However, relying solely on kernel version numbers may not be sufficient for certain operating system distributions, as maintainers for distributions such as RHEL, Ubuntu and SUSE often backport selected features to older kernel releases (retaining the older kernel version).

## Pod sysctls

On Linux, the `sysctl()` system call configures kernel parameters at run time. There is a command line tool named `sysctl` that you can use to configure these parameters, and many are exposed via the `proc` filesystem.

Some sysctls are only available if you have a modern enough kernel.

The following sysctls have a minimal kernel version requirement, and are supported in the safe set:

- `net.ipv4.ip_local_reserved_ports` (since Kubernetes 1.27, needs kernel 3.16+);
- `net.ipv4.tcp_keepalive_time` (since Kubernetes 1.29, needs kernel 4.5+);
- `net.ipv4.tcp_fin_timeout` (since Kubernetes 1.29, needs kernel 4.6+);
- `net.ipv4.tcp_keepalive_intvl` (since Kubernetes 1.29, needs kernel 4.5+);
- `net.ipv4.tcp_keepalive_probes` (since Kubernetes 1.29, needs kernel 4.5+);
- `net.ipv4.tcp_syncookies` (namespaced since kernel 4.6+).
- `net.ipv4.tcp_rmem` (since Kubernetes 1.32, needs kernel 4.15+).
- `net.ipv4.tcp_wmem` (since Kubernetes 1.32, needs kernel 4.15+).
- `net.ipv4.vs.conn_reuse_mode` (used in `ipvs` proxy mode, needs kernel 4.1+);

### kube proxy `nftables` proxy mode

For Kubernetes 1.34, the `nftables` mode of kube-proxy requires version 1.0.1 or later of the nft command-line tool, as well as kernel 5.13 or later.

For testing/development purposes, you can use older kernels, as far back as 5.4 if you set the `nftables.skipKernelVersionCheck` option in the kube-proxy config. But this is not recommended in production since it may cause problems with other nftables users on the system.

## Version 2 control groups

Kubernetes cgroup v1 support is in maintained mode starting from Kubernetes v1.31; using cgroup v2 is recommended. In Linux 5.8, the system-level `cpu.stat` file was added to the root cgroup for convenience.

In runc document, Kernel older than 5.2 is not recommended due to lack of freezer.

## Pressure Stall Information (PSI)

Pressure Stall Information is supported in Linux kernel versions 4.20 and up, but requires the following configuration:

- The kernel must be compiled with the `CONFIG_PSI=y` option. Most modern distributions enable this by default. You can check your kernel's configuration by running `zgrep CONFIG_PSI /proc/config.gz`.
- Some Linux distributions may compile PSI into the kernel but disable it by default. If so, you need to enable it at boot time by adding the `psi=1` parameter to the kernel command line.

## Other kernel requirements

Some features may depend on new kernel functionalities and have specific kernel requirements:

1. Recursive read only mount: This is implemented by applying the `MOUNT_ATTR_RDONLY` attribute with the `AT_RECURSIVE` flag using `mount_setattr`(2) added in Linux kernel v5.12.
2. Pod user namespace support requires minimal kernel version 6.5+, according to KEP-127.
3. For node system swap, tmpfs set to `noswap` is not supported until kernel 6.3.

## Linux kernel long term maintenance

Active kernel releases can be found in kernel.org.

There are usually several *long term maintenance* kernel releases provided for the purposes of backporting bug fixes for older kernel trees. Only important bug fixes are applied to such kernels and they don't usually see very frequent releases, especially for older trees. See the Linux kernel website for the list of releases in the *Longterm* category.

## What's next

- See sysctls for more details.
- Allow running kube-proxy with in nftables mode.

- Read more information in cgroups v2.

---

# Kubelet Configuration Directory Merging

When using the kubelet's `--config-dir` flag to specify a drop-in directory for configuration, there is some specific behavior on how different types are merged.

Here are some examples of how different data types behave during configuration merging:

### Structure Fields

There are two types of structure fields in a YAML structure: singular (or a scalar type) and embedded (structures that contain scalar types). The configuration merging process handles the overriding of singular and embedded struct fields to create a resulting kubelet configuration.

For instance, you may want a baseline kubelet configuration for all nodes, but you may want to customize the `address` and `authorization` fields. This can be done as follows:

Main kubelet configuration file contents:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250authorization:  mode: Webhook  webhook:    cacheAuthorizedTTL: "5m"    cacheUnauthorizedTTL:
```

Contents of a file in `--config-dir` directory:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationauthorization:  mode: AlwaysAllow  webhook:    cacheAuthorizedTTL: "8m"    cacheUnauthorizedTTL: "45s"ad
```

The resulting configuration will be as follows:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250serializeImagePulls: falseauthorization:  mode: AlwaysAllow  webhook:    cacheAuthorizedTTL:
```

### Lists

You can override the slices/lists values of the kubelet configuration. However, the entire list gets overridden during the merging process. For example, you can override the `clusterDNS` list as follows:

Main kubelet configuration file contents:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250serializeImagePulls: falseclusterDNS:  - "192.168.0.9"  - "192.168.0.8"
```

Contents of a file in `--config-dir` directory:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationclusterDNS:  - "192.168.0.2"  - "192.168.0.3"  - "192.168.0.5"
```

The resulting configuration will be as follows:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250serializeImagePulls: falseclusterDNS:  - "192.168.0.2"  - "192.168.0.3"  - "192.168.0.5"
```

### Maps, including Nested Structures

Individual fields in maps, regardless of their value types (boolean, string, etc.), can be selectively overridden. However, for `map[string][]string`, the entire list associated with a specific field gets overridden. Let's understand this better with an example, particularly on fields like `featureGates` and `staticPodURLHeader`:

Main kubelet configuration file contents:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250serializeImagePulls: falsefeatureGates:  AllAlpha: false  MemoryQoS: truestaticPodURLHeader:
```

Contents of a file in `--config-dir` directory:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationfeatureGates:  MemoryQoS: false  KubeletTracing: true  DynamicResourceAllocation: truestaticPodURLHeader
```

The resulting configuration will be as follows:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationport: 20250serializeImagePulls: falsefeatureGates:  AllAlpha: false  MemoryQoS: false  KubeletTracing: t
```

---

# Kubelet Device Manager API Versions

This page provides details of version compatibility between the Kubernetes device plugin API, and different versions of Kubernetes itself.

## Compatibility matrix

| | v1alpha1 | v1beta1 |
|---|---|---|
| Kubernetes 1.21 | - | ✓ |
| Kubernetes 1.22 | - | ✓ |
| Kubernetes 1.23 | - | ✓ |
| Kubernetes 1.24 | - | ✓ |
| Kubernetes 1.25 | - | ✓ |
| Kubernetes 1.26 | - | ✓ |

Key:

- ✓ Exactly the same features / API objects in both device plugin API and the Kubernetes version.
- + The device plugin API has features or API objects that may not be present in the Kubernetes cluster, either because the device plugin API has added additional new API calls, or that the server has removed an old API call. However, everything they have in common (most other APIs) will work. Note that alpha APIs may vanish or change significantly between one minor release and the next.
- – The Kubernetes cluster has features the device plugin API can't use, either because server has added additional API calls, or that device plugin API has removed an old API call. However, everything they share in common (most APIs) will work.

---

# Kubelet Checkpoint API

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

Checkpointing a container is the functionality to create a stateful copy of a running container. Once you have a stateful copy of a container, you could move it to a different computer for debugging or similar purposes.

If you move the checkpointed container data to a computer that's able to restore it, that restored container continues to run at exactly the same point it was checkpointed. You can also inspect the saved data, provided that you have suitable tools for doing so.

Creating a checkpoint of a container might have security implications. Typically a checkpoint contains all memory pages of all processes in the checkpointed container. This means that everything that used to be in memory is now available on the local disk. This includes all private data and possibly keys used for encryption. The underlying CRI implementations (the container runtime on that node) should create the checkpoint archive to be only accessible by the `root` user. It is still important to remember if the checkpoint archive is transferred to another system all memory pages will be readable by the owner of the checkpoint archive.

## Operations

### `post` checkpoint the specified container

Tell the kubelet to checkpoint a specific container from the specified Pod.

Consult the Kubelet authentication/authorization reference for more information about how access to the kubelet checkpoint interface is controlled.

The kubelet will request a checkpoint from the underlying CRI implementation. In the checkpoint request the kubelet will specify the name of the checkpoint archive as `checkpoint-<podFullName>-<containerName>-<timestamp>.tar` and also request to store the checkpoint archive in the `checkpoints` directory below its root directory (as defined by `--root-dir`). This defaults to `/var/lib/kubelet/checkpoints`.

The checkpoint archive is in *tar* format, and could be listed using an implementation of `tar`. The contents of the archive depend on the underlying CRI implementation (the container runtime on that node).

#### HTTP Request

POST /checkpoint/{namespace}/{pod}/{container}

#### Parameters

- **namespace** (*in path*): string, required

  Namespace

- **pod** (*in path*): string, required

  Pod

- **container** (*in path*): string, required

  Container

- **timeout** (*in query*): integer

  Timeout in seconds to wait until the checkpoint creation is finished. If zero or no timeout is specified the default CRI timeout value will be used. Checkpoint creation time depends directly on the used memory of the container. The more memory a container uses the more time is required to create the corresponding checkpoint.

#### Response

200: OK

401: Unauthorized

404: Not Found (if the `ContainerCheckpoint` feature gate is disabled)

404: Not Found (if the specified `namespace`, `pod` or `container` cannot be found)

500: Internal Server Error (if the CRI implementation encounter an error during checkpointing (see error message for further details))

500: Internal Server Error (if the CRI implementation does not implement the checkpoint CRI API (see error message for further details))

---

# Networking Reference

This section of the Kubernetes documentation provides reference details of Kubernetes networking.

---



---

# Virtual IPs and Service Proxies

Every node in a Kubernetes cluster runs a kube-proxy (unless you have deployed your own alternative component in place of `kube-proxy`).

The `kube-proxy` component is responsible for implementing a *virtual IP* mechanism for Services of `type` other than `ExternalName`. Each instance of kube-proxy watches the Kubernetes control plane for the addition and removal of Service and EndpointSlice objects. For each Service, kube-proxy calls appropriate APIs (depending on the kube-proxy mode) to configure the node to capture traffic to the Service's `clusterIP` and `port`, and redirect that traffic to one of the Service's endpoints (usually a Pod, but possibly an arbitrary user-provided IP address). A control loop ensures that the rules on each node are reliably synchronized with the Service and EndpointSlice state as indicated by the API server.

**Virtual IP mechanism for Services, using iptables mode**

A question that pops up every now and then is why Kubernetes relies on proxying to forward inbound traffic to backends. What about other approaches? For example, would it be possible to configure DNS records that have multiple A values (or AAAA for IPv6), and rely on round-robin name resolution?

There are a few reasons for using proxying for Services:

- There is a long history of DNS implementations not respecting record TTLs, and caching the results of name lookups after they should have expired.
- Some apps do DNS lookups only once and cache the results indefinitely.
- Even if apps and libraries did proper re-resolution, the low or zero TTLs on the DNS records could impose a high load on DNS that then becomes difficult to manage.

Later in this page you can read about how various kube-proxy implementations work. Overall, you should note that, when running `kube-proxy`, kernel level rules may be modified (for example, iptables rules might get created), which won't get cleaned up, in some cases until you reboot. Thus, running kube-proxy is something that should only be done by an administrator who understands the consequences of having a low level, privileged network proxying service on a computer. Although the `kube-proxy` executable supports a `cleanup` function, this function is not an official feature and thus is only available to use as-is.

Some of the details in this reference refer to an example: the backend Pods for a stateless image-processing workloads, running with three replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

## Proxy modes

The kube-proxy starts up in different modes, which are determined by its configuration.

On Linux nodes, the available modes for kube-proxy are:

`iptables`
: A mode where the kube-proxy configures packet forwarding rules using iptables.

`ipvs`
: a mode where the kube-proxy configures packet forwarding rules using ipvs.

`nftables`
: a mode where the kube-proxy configures packet forwarding rules using nftables.

There is only one mode available for kube-proxy on Windows:

`kernelspace`
: a mode where the kube-proxy configures packet forwarding rules in the Windows kernel

### `iptables` proxy mode

*This proxy mode is only available on Linux nodes.*

In this mode, kube-proxy configures packet forwarding rules using the iptables API of the kernel netfilter subsystem. For each endpoint, it installs iptables rules which, by default, select a backend Pod at random.

#### Example

As an example, consider the image processing application described earlier in the page. When the backend Service is created, the Kubernetes control plane assigns a virtual IP address, for example 10.0.0.1. For this example, assume that the Service port is 1234. All of the kube-proxy instances in the cluster observe the creation of the new Service.

When kube-proxy on a node sees a new Service, it installs a series of iptables rules which redirect from the virtual IP address to more iptables rules, defined per Service. The per-Service rules link to further rules for each backend endpoint, and the per- endpoint rules redirect traffic (using destination NAT) to the backends.

When a client connects to the Service's virtual IP address the iptables rule kicks in. A backend is chosen (either based on session affinity or randomly) and packets are redirected to the backend without rewriting the client IP address.

This same basic flow executes when traffic comes in through a `type: NodePort` Service, or through a load-balancer, though in those cases the client IP address does get altered.

#### Optimizing iptables mode performance

In iptables mode, kube-proxy creates a few iptables rules for every Service, and a few iptables rules for each endpoint IP address. In clusters with tens of thousands of Pods and Services, this means tens of thousands of iptables rules, and kube-proxy may take a long time to update the rules in the kernel when Services (or their EndpointSlices) change. You can adjust the syncing behavior of kube-proxy via options in the `iptables` section of the kube-proxy configuration file (which you specify via `kube-proxy --config <path>`):

```
...
iptables:  minSyncPeriod: 1s  syncPeriod: 30s...
```

##### `minSyncPeriod`

The `minSyncPeriod` parameter sets the minimum duration between attempts to resynchronize iptables rules with the kernel. If it is `0s`, then kube-proxy will always immediately synchronize the rules every time any Service or EndpointSlice changes. This works fine in very small clusters, but it results in a lot of redundant work when lots of things change in a small time period. For example, if you have a Service backed by a Deployment with 100 pods, and you delete the Deployment, then with `minSyncPeriod: 0s`, kube-proxy would end up removing the Service's endpoints from the iptables rules one by one, resulting in a total of 100 updates. With a larger `minSyncPeriod`, multiple Pod deletion events would get aggregated together, so kube-proxy might instead end up making, say, 5 updates, each removing 20 endpoints, which will be much more efficient in terms of CPU, and result in the full set of changes being synchronized faster.

The larger the value of `minSyncPeriod`, the more work that can be aggregated, but the downside is that each individual change may end up waiting up to the full `minSyncPeriod` before being processed, meaning that the iptables rules spend more time being out-of-sync with the current API server state.

The default value of `1s` should work well in most clusters, but in very large clusters it may be necessary to set it to a larger value. Especially, if kube-proxy's `sync_proxy_rules_duration_seconds` metric indicates an average time much larger than 1 second, then bumping up `minSyncPeriod` may make updates more efficient.

##### Updating legacy `minSyncPeriod` configuration

Older versions of kube-proxy updated all the rules for all Services on every sync; this led to performance issues (update lag) in large clusters, and the recommended solution was to set a larger `minSyncPeriod`. Since Kubernetes v1.28, the iptables mode of kube-proxy uses a more minimal approach, only making updates where Services or EndpointSlices have actually changed.

If you were previously overriding `minSyncPeriod`, you should try removing that override and letting kube-proxy use the default value (`1s`) or at least a smaller value than you were using before upgrading.

If you are not running kube-proxy from Kubernetes 1.34, check the behavior and associated advice for the version that you are actually running.

##### `syncPeriod`

The `syncPeriod` parameter controls a handful of synchronization operations that are not directly related to changes in individual Services and EndpointSlices. In particular, it controls how quickly kube-proxy notices if an external component has interfered with kube-proxy's iptables rules. In large clusters, kube-proxy also only performs certain cleanup operations once every `syncPeriod` to avoid unnecessary work.

For the most part, increasing `syncPeriod` is not expected to have much impact on performance, but in the past, it was sometimes useful to set it to a very large value (eg, `1h`). This is no longer recommended, and is likely to hurt functionality more than it improves performance.

### IPVS proxy mode

*This proxy mode is only available on Linux nodes.*

In `ipvs` mode, kube-proxy uses the kernel IPVS and iptables APIs to create rules to redirect traffic from Service IPs to endpoint IPs.

The IPVS proxy mode is based on netfilter hook function that is similar to iptables mode, but uses a hash table as the underlying data structure and works in the kernel space.

#### Note:

The `ipvs` proxy mode was an experiment in providing a Linux kube-proxy backend with better rule-synchronizing performance and higher network-traffic throughput than the `iptables` mode. While it succeeded in those goals, the kernel IPVS API turned out to be a bad match for the Kubernetes Services API, and the `ipvs` backend was never able to implement all of the edge cases of Kubernetes Service functionality correctly. At some point in the future, it is expected to be formally deprecated as a feature.

The `nftables` proxy mode (described below) is essentially a replacement for both the `iptables` and `ipvs` modes, with better performance than either of them, and is recommended as a replacement for `ipvs`. If you are deploying onto Linux systems that are too old to run the `nftables` proxy mode, you should

also consider trying the `iptables` mode rather than `ipvs`, since the performance of `iptables` mode has improved greatly since the `ipvs` mode was first introduced.

IPVS provides more options for balancing traffic to backend Pods; these are:

- `rr` (Round Robin): Traffic is equally distributed amongst the backing servers.
- `wrr` (Weighted Round Robin): Traffic is routed to the backing servers based on the weights of the servers. Servers with higher weights receive new connections and get more requests than servers with lower weights.
- `lc` (Least Connection): More traffic is assigned to servers with fewer active connections.
- `wlc` (Weighted Least Connection): More traffic is routed to servers with fewer connections relative to their weights, that is, connections divided by weight.
- `lblc` (Locality based Least Connection): Traffic for the same IP address is sent to the same backing server if the server is not overloaded and available; otherwise the traffic is sent to servers with fewer connections, and keep it for future assignment.
- `lblcr` (Locality Based Least Connection with Replication): Traffic for the same IP address is sent to the server with least connections. If all the backing servers are overloaded, it picks up one with fewer connections and adds it to the target set. If the target set has not changed for the specified time, the server with the highest load is removed from the set, in order to avoid a high degree of replication.
- `sh` (Source Hashing): Traffic is sent to a backing server by looking up a statically assigned hash table based on the source IP addresses.
- `dh` (Destination Hashing): Traffic is sent to a backing server by looking up a statically assigned hash table based on their destination addresses.
- `sed` (Shortest Expected Delay): Traffic forwarded to a backing server with the shortest expected delay. The expected delay is `(C + 1) / U` if sent to a server, where `C` is the number of connections on the server and `U` is the fixed service rate (weight) of the server.
- `nq` (Never Queue): Traffic is sent to an idle server if there is one, instead of waiting for a fast one; if all servers are busy, the algorithm falls back to the `sed` behavior.
- `mh` (Maglev Hashing): Assigns incoming jobs based on Google's Maglev hashing algorithm, This scheduler has two flags: `mh-fallback`, which enables fallback to a different server if the selected server is unavailable, and `mh-port`, which adds the source port number to the hash computation. When using `mh`, kube-proxy always sets the `mh-port` flag and does not enable the `mh-fallback` flag. In proxy-mode=ipvs `mh` will work as source-hashing (`sh`), but with ports.

These scheduling algorithms are configured through the `ipvs.scheduler` field in the kube-proxy configuration.

**Note:**

To run kube-proxy in IPVS mode, you must make IPVS available on the node before starting kube-proxy.

When kube-proxy starts in IPVS proxy mode, it verifies whether IPVS kernel modules are available. If the IPVS kernel modules are not detected, then kube-proxy exits with an error.

**Virtual IP address mechanism for Services, using IPVS mode**

### `nftables` proxy mode

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

*This proxy mode is only available on Linux nodes, and requires kernel 5.13 or later.*

In this mode, kube-proxy configures packet forwarding rules using the nftables API of the kernel netfilter subsystem. For each endpoint, it installs nftables rules which, by default, select a backend Pod at random.

The nftables API is the successor to the iptables API and is designed to provide better performance and scalability than iptables. The `nftables` proxy mode is able to process changes to service endpoints faster and more efficiently than the `iptables` mode, and is also able to more efficiently process packets in the kernel (though this only becomes noticeable in clusters with tens of thousands of services).

As of Kubernetes 1.34, the `nftables` mode is still relatively new, and may not be compatible with all network plugins; consult the documentation for your network plugin.

#### Migrating from `iptables` mode to `nftables`

Users who want to switch from the default `iptables` mode to the `nftables` mode should be aware that some features work slightly differently the `nftables` mode:

- **NodePort interfaces**: In `iptables` mode, by default, NodePort services are reachable on all local IP addresses. This is usually not what users want, so the `nftables` mode defaults to `--nodeport-addresses primary`, meaning Services using `type: NodePort` are only reachable on the node's primary IPv4 and/or IPv6 addresses. You can override this by specifying an explicit value for that option: e.g., `--nodeport-addresses 0.0.0.0/0` to listen on all (local) IPv4 IPs.
- `type: NodePort` **Services on `127.0.0.1`**: In `iptables` mode, if the `--nodeport-addresses` range includes `127.0.0.1` (and the option `--iptables-localhost-nodeports false` option is not passed), then Services of `type: NodePort` are reachable even on "localhost" (`127.0.0.1`). In `nftables` mode (and `ipvs` mode), this will not work. If you are not sure if you are depending on this functionality, you can check kube-proxy's `iptables_localhost_nodeports_accepted_packets_total` metric; if it is non-0, that means that some client has connected to a `type: NodePort` Service via localhost/loopback.

- **NodePort interaction with firewalls**: The `iptables` mode of kube-proxy tries to be compatible with overly-aggressive firewalls; for each `type: NodePort` service, it will add rules to accept inbound traffic on that port, in case that traffic would otherwise be blocked by a firewall. This approach will not work with firewalls based on nftables, so kube-proxy's `nftables` mode does not do anything here; if you have a local firewall, you must ensure that it is properly configured to allow Kubernetes traffic through (e.g., by allowing inbound traffic on the entire NodePort range).
- **Conntrack bug workarounds**: Linux kernels prior to 6.1 have a bug that can result in long-lived TCP connections to service IPs being closed with the error "Connection reset by peer". The `iptables` mode of kube-proxy installs a workaround for this bug, but this workaround was later found to cause other problems in some clusters. The `nftables` mode does not install any workaround by default, but you can check kube-proxy's `iptables_ct_state_invalid_dropped_packets_total` metric to see if your cluster is depending on the workaround, and if so, you can run kube-proxy with the option `--conntrack-tcp-be-liberal` to work around the problem in `nftables` mode.

#### `kernelspace` proxy mode

*This proxy mode is only available on Windows nodes*.

The kube-proxy configures packet filtering rules in the Windows *Virtual Filtering Platform* (VFP), an extension to Windows vSwitch. These rules process encapsulated packets within the node-level virtual networks, and rewrite packets so that the destination IP address (and layer 2 information) is correct for getting the packet routed to the correct destination. The Windows VFP is analogous to tools such as Linux `nftables` or `iptables`. The Windows VFP extends the *Hyper-V Switch*, which was initially implemented to support virtual machine networking.

When a Pod on a node sends traffic to a virtual IP address, and the kube-proxy selects a Pod on a different node as the load balancing target, the `kernelspace` proxy mode rewrites that packet to be destined to the target backend Pod. The Windows *Host Networking Service* (HNS) ensures that packet rewriting rules are configured so that the return traffic appears to come from the virtual IP address and not the specific backend Pod.

##### Direct server return for `kernelspace` mode

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

As an alternative to the basic operation, a node that hosts the backend Pod for a Service can apply the packet rewriting directly, rather than placing this burden on the node where the client Pod is running. This is called *direct server return*.

To use this, you must run kube-proxy with the `--enable-dsr` command line argument **and** enable the `WinDSR` feature gate.

Direct server return also optimizes the case for Pod return traffic even when both Pods are running on the same node.

## Session affinity

In these proxy models, the traffic bound for the Service's IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or Services or Pods.

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can select the session affinity based on the client's IP addresses by setting `.spec.sessionAffinity` to `ClientIP` for a Service (the default is `None`).

### Session stickiness timeout

You can also set the maximum session sticky time by setting `.spec.sessionAffinityConfig.clientIP.timeoutSeconds` appropriately for a Service. (the default value is 10800, which works out to be 3 hours).

**Note:**

On Windows, setting the maximum session sticky time for Services is not supported.

## IP address assignment to Services

Unlike Pod IP addresses, which actually route to a fixed destination, Service IPs are not actually answered by a single host. Instead, kube-proxy uses packet processing logic (such as Linux iptables) to define *virtual* IP addresses which are transparently redirected as needed.

When clients connect to the VIP, their traffic is automatically transported to an appropriate endpoint. The environment variables and DNS for Services are actually populated in terms of the Service's virtual IP address (and port).

### Avoiding collisions

One of the primary philosophies of Kubernetes is that you should not be exposed to situations that could cause your actions to fail through no fault of your own. For the design of the Service resource, this means not making you choose your own IP address if that choice might collide with someone else's choice. That is an isolation failure.

In order to allow you to choose an IP address for your Services, we must ensure that no two Services can collide. Kubernetes does that by allocating each Service its own IP address from within the `service-cluster-ip-range` CIDR range that is configured for the API Server.

### IP address allocation tracking

To ensure each Service receives a unique IP address, an internal allocator atomically updates a global allocation map in etcd prior to creating each Service. The map object must exist in the registry for Services to get IP address assignments, otherwise creations will fail with a message indicating an IP address could not be allocated.

In the control plane, a background controller is responsible for creating that map (needed to support migrating from older versions of Kubernetes that used in-memory locking). Kubernetes also uses controllers to check for invalid assignments (for example: due to administrator intervention) and for cleaning up allocated IP addresses that are no longer used by any Services.

**IP address allocation tracking using the Kubernetes API**

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

The control plane replaces the existing etcd allocator with a revised implementation that uses IPAddress and ServiceCIDR objects instead of an internal global allocation map. Each cluster IP address associated to a Service then references an IPAddress object.

Enabling the feature gate also replaces a background controller with an alternative that handles the IPAddress objects and supports migration from the old allocator model. Kubernetes 1.34 does not support migrating from IPAddress objects to the internal allocation map.

One of the main benefits of the revised allocator is that it removes the size limitations for the IP address range that can be used for the cluster IP address of Services. With `MultiCIDRServiceAllocator` enabled, there are no limitations for IPv4, and for IPv6 you can use IP address netmasks that are a /64 or smaller (as opposed to /108 with the legacy implementation).

Making IP address allocations available via the API means that you as a cluster administrator can allow users to inspect the IP addresses assigned to their Services. Kubernetes extensions, such as the Gateway API, can use the IPAddress API to extend Kubernetes' inherent networking capabilities.

Here is a brief example of a user querying for IP addresses:

```
kubectl get services
```

```
NAME         TYPE        CLUSTER-IP        EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   2001:db8:1:2::1   <none>        443/TCP   3d1h
```

```
kubectl get ipaddresses
```

```
NAME              PARENTREF
2001:db8:1:2::1   services/default/kubernetes
2001:db8:1:2::a   services/kube-system/kube-dns
```

Kubernetes also allow users to dynamically define the available IP ranges for Services using ServiceCIDR objects. During bootstrap, a default ServiceCIDR object named `kubernetes` is created from the value of the `--service-cluster-ip-range` command line argument to kube-apiserver:

```
kubectl get servicecidrs
```

```
NAME         CIDRS         AGE
kubernetes   10.96.0.0/28  17m
```

Users can create or delete new ServiceCIDR objects to manage the available IP ranges for Services:

```
cat <<'EOF' | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: ServiceCIDR
metadata:
  name: newservicecidr
spec:
  cidrs:
  - 10.96.0.0/24
EOF
```

```
servicecidr.networking.k8s.io/newcidr1 created
```

```
kubectl get servicecidrs
```

```
NAME             CIDRS         AGE
kubernetes       10.96.0.0/28  17m
newservicecidr   10.96.0.0/24  7m
```

Distributions or administrators of Kubernetes clusters may want to control that new Service CIDRs added to the cluster does not overlap with other networks on the cluster, that only belong to a specific range of IPs or just simple retain the existing behavior of only having one ServiceCIDR per cluster. An example of a Validation Admission Policy to achieve this is:

```
---
apiVersion: admissionregistration.k8s.io/v1kind: ValidatingAdmissionPolicymetadata:  name: "servicecidrs-default"spec:  failurePol
```

### IP address ranges for Service virtual IP addresses

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

Kubernetes divides the `ClusterIP` range into two bands, based on the size of the configured `service-cluster-ip-range` by using the following formula `min(max(16, cidrSize / 16), 256)`. That formula means the result is *never less than 16 or more than 256, with a graduated step function between them*.

Kubernetes prefers to allocate dynamic IP addresses to Services by choosing from the upper band, which means that if you want to assign a specific IP address to a `type: ClusterIP` Service, you should manually assign an IP address from the **lower** band. That approach reduces the risk of a conflict over allocation.

## Traffic policies

You can set the `.spec.internalTrafficPolicy` and `.spec.externalTrafficPolicy` fields to control how Kubernetes routes traffic to healthy (“ready”) backends.

### Internal traffic policy

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

You can set the `.spec.internalTrafficPolicy` field to control how traffic from internal sources is routed. Valid values are `Cluster` and `Local`. Set the field to `Cluster` to route internal traffic to all ready endpoints and `Local` to only route to ready node-local endpoints. If the traffic policy is `Local` and there are no

node-local endpoints, traffic is dropped by kube-proxy.

### External traffic policy

You can set the `.spec.externalTrafficPolicy` field to control how traffic from external sources is routed. Valid values are `Cluster` and `Local`. Set the field to `Cluster` to route external traffic to all ready endpoints and `Local` to only route to ready node-local endpoints. If the traffic policy is `Local` and there are no node-local endpoints, the kube-proxy does not forward any traffic for the relevant Service.

If `Cluster` is specified, all nodes are eligible load balancing targets *as long as* the node is not being deleted and kube-proxy is healthy. In this mode: load balancer health checks are configured to target the service proxy's readiness port and path. In the case of kube-proxy this evaluates to: `${NODE_IP}:10256/healthz`. kube-proxy will return either an HTTP code 200 or 503. kube-proxy's load balancer health check endpoint returns 200 if:

1. kube-proxy is healthy, meaning:

   it's able to progress programming the network and isn't timing out while doing so (the timeout is defined to be: **2 × `iptables.syncPeriod`**); and

2. the node is not being deleted (there is no deletion timestamp set for the Node).

kube-proxy returns 503 and marks the node as not eligible when it's being deleted because it supports connection draining for terminating nodes. A couple of important things occur from the point of view of a Kubernetes-managed load balancer when a node *is being* / *is* deleted.

While deleting:

- kube-proxy will start failing its readiness probe and essentially mark the node as not eligible for load balancer traffic. The load balancer health check failing causes load balancers which support connection draining to allow existing connections to terminate, and block new connections from establishing.

When deleted:

- The service controller in the Kubernetes cloud controller manager removes the node from the referenced set of eligible targets. Removing any instance from the load balancer's set of backend targets immediately terminates all connections. This is also the reason kube-proxy first fails the health check while the node is deleting.

It's important to note for Kubernetes vendors that if any vendor configures the kube-proxy readiness probe as a liveness probe: that kube-proxy will start restarting continuously when a node is deleting until it has been fully deleted. kube-proxy exposes a `/livez` path which, as opposed to the `/healthz` one, does **not** consider the Node's deleting state and only its progress programming the network. `/livez` is therefore the recommended path for anyone looking to define a livenessProbe for kube-proxy.

Users deploying kube-proxy can inspect both the readiness / liveness state by evaluating the metrics: `proxy_livez_total` / `proxy_healthz_total`. Both metrics publish two series, one with the 200 label and one with the 503 one.

For `Local` Services: kube-proxy will return 200 if

1. kube-proxy is healthy/ready, and
2. has a local endpoint on the node in question.

Node deletion does **not** have an impact on kube-proxy's return code for what concerns load balancer health checks. The reason for this is: deleting nodes could end up causing an ingress outage should all endpoints simultaneously be running on said nodes.

The Kubernetes project recommends that cloud provider integration code configures load balancer health checks that target the service proxy's healthz port. If you are using or implementing your own virtual IP implementation, that people can use instead of kube-proxy, you should set up a similar health checking port with logic that matches the kube-proxy implementation.

### Traffic to terminating endpoints

FEATURE STATE: `Kubernetes v1.28 [stable]`

If the `ProxyTerminatingEndpoints` feature gate is enabled in kube-proxy and the traffic policy is `Local`, that node's kube-proxy uses a more complicated algorithm to select endpoints for a Service. With the feature enabled, kube-proxy checks if the node has local endpoints and whether or not all the local endpoints are marked as terminating. If there are local endpoints and **all** of them are terminating, then kube-proxy will forward traffic to those terminating endpoints. Otherwise, kube-proxy will always prefer forwarding traffic to endpoints that are not terminating.

This forwarding behavior for terminating endpoints exists to allow `NodePort` and `LoadBalancer` Services to gracefully drain connections when using `externalTrafficPolicy: Local`.

As a deployment goes through a rolling update, nodes backing a load balancer may transition from N to 0 replicas of that deployment. In some cases, external load balancers can send traffic to a node with 0 replicas in between health check probes. Routing traffic to terminating endpoints ensures that Nodes that are scaling down Pods can gracefully receive and drain traffic to those terminating Pods. By the time the Pod completes termination, the external load balancer should have seen the node's health check failing and fully removed the node from the backend pool.

## Traffic Distribution

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

The `spec.trafficDistribution` field within a Kubernetes Service allows you to express preferences for how traffic should be routed to Service endpoints.

`PreferClose`
: This prioritizes sending traffic to endpoints in the same zone as the client. The EndpointSlice controller updates EndpointSlices with `hints` to communicate this preference, which kube-proxy then uses for routing decisions. If a client's zone does not have any available endpoints, traffic will be routed cluster-wide for that client.

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

In Kubernetes 1.34, two additional values are available (unless the `PreferSameTrafficDistribution` feature gate is disabled):

`PreferSameZone`
: This means the same thing as `PreferClose`, but is more explicit. (Originally, the intention was that `PreferClose` might later include functionality other than just "prefer same zone", but this is no longer planned. In the future, `PreferSameZone` will be the recommended value to use for this functionality, and `PreferClose` will be considered a deprecated alias for it.)

`PreferSameNode`
: This prioritizes sending traffic to endpoints on the same node as the client. As with `PreferClose`/`PreferSameZone`, the EndpointSlice controller updates EndpointSlices with `hints` indicating that a slice should be used for a particular node. If a client's node does not have any available endpoints, then the service proxy will fall back to "same zone" behavior, or cluster-wide if there are no same-zone endpoints either.

In the absence of any value for `trafficDistribution`, the default strategy is to distribute traffic evenly to all endpoints in the cluster.

### Comparison with `service.kubernetes.io/topology-mode: Auto`

The `trafficDistribution` field with `PreferClose`/`PreferSameZone`, and the older "Topology-Aware Routing" feature using the `service.kubernetes.io/topology-mode: Auto` annotation both aim to prioritize same-zone traffic. However, there is a key difference in their approaches:

- `service.kubernetes.io/topology-mode: Auto` attempts to distribute traffic proportionally across zones based on allocatable CPU resources. This heuristic includes safeguards (such as the fallback behavior for small numbers of endpoints), sacrificing some predictability in favor of potentially better load balancing.

- `trafficDistribution: PreferClose` aims to be simpler and more predictable: "If there are endpoints in the zone, they will receive all traffic for that zone, if there are no endpoints in a zone, the traffic will be distributed to other zones". This approach offers more predictability, but it means that you are responsible for avoiding endpoint overload.

If the `service.kubernetes.io/topology-mode` annotation is set to `Auto`, it will take precedence over `trafficDistribution`. The annotation may be deprecated in the future in favor of the `trafficDistribution` field.

### Interaction with Traffic Policies

When compared to the `trafficDistribution` field, the traffic policy fields (`externalTrafficPolicy` and `internalTrafficPolicy`) are meant to offer a stricter traffic locality requirements. Here's how `trafficDistribution` interacts with them:

- Precedence of Traffic Policies: For a given Service, if a traffic policy (`externalTrafficPolicy` or `internalTrafficPolicy`) is set to `Local`, it takes precedence over `trafficDistribution` for the corresponding traffic type (external or internal, respectively).

- `trafficDistribution` Influence: For a given Service, if a traffic policy (`externalTrafficPolicy` or `internalTrafficPolicy`) is set to `Cluster` (the default), or if the fields are not set, then `trafficDistribution` guides the routing behavior for the corresponding traffic type (external or internal, respectively). This means that an attempt will be made to route traffic to an endpoint that is in the same zone as the client.

### Considerations for using traffic distribution control

A Service using `trafficDistribution` will attempt to route traffic to (healthy) endpoints within the appropriate topology, even if this means that some endpoints receive much more traffic than other endpoints. If you do not have a sufficient number of endpoints within the same topology ("same zone", "same node", etc.) as the clients, then endpoints may become overloaded. This is especially likely if incoming traffic is not proportionally distributed across the topology. To mitigate this, consider the following strategies:

- Pod Topology Spread Constraints: Use Pod Topology Spread Constraints to distribute your pods evenly across zones or nodes.

- Zone-specific Deployments: If you are using "same zone" traffic distribution, but expect to see different traffic patterns in different zones, you can create a separate Deployment for each zone. This approach allows the separate workloads to scale independently. There are also workload management addons available from the ecosystem, outside the Kubernetes project itself, that can help here.

## What's next

To learn more about Services, read Connecting Applications with Services.

You can also:

- Read about Services as a concept
- Read about Ingresses as a concept
- Read the API reference for the Service API

---

# Articles on dockershim Removal and on Using CRI-compatible Runtimes

This is a list of articles and other pages that are either about the Kubernetes' deprecation and removal of *dockershim*, or about using CRI-compatible container runtimes, in connection with that removal.

## Kubernetes project

- Kubernetes blog: Dockershim Removal FAQ (originally published 2020/12/02)

- Kubernetes blog: Updated: Dockershim Removal FAQ (updated published 2022/02/17)

- Kubernetes blog: Kubernetes is Moving on From Dockershim: Commitments and Next Steps (published 2022/01/07)

- Kubernetes blog: Dockershim removal is coming. Are you ready? (published 2021/11/12)

- Kubernetes documentation: Migrating from dockershim
- Kubernetes documentation: Container Runtimes
- Kubernetes enhancement proposal: KEP-2221: Removing dockershim from kubelet
- Kubernetes enhancement proposal issue: Removing dockershim from kubelet (*k/enhancements#2221*)

You can provide feedback via the GitHub issue **Dockershim removal feedback & issues**. (*k/kubernetes/#106917*)

## External sources

- Amazon Web Services EKS documentation: Amazon EKS is ending support for Dockershim
- CNCF conference video: Lessons Learned Migrating Kubernetes from Docker to containerd Runtime (Ana Caylin, at KubeCon Europe 2019)
- Docker.com blog: What developers need to know about Docker, Docker Engine, and Kubernetes v1.20 (published 2020/12/04)
- "*Google Open Source*" channel on YouTube: Learn Kubernetes with Google - Migrating from Dockershim to Containerd
- Microsoft Apps on Azure blog: Dockershim deprecation and AKS (published 2022/01/21)
- Mirantis blog: The Future of Dockershim is cri-dockerd (published 2021/04/21)
- Mirantis: Mirantis/cri-dockerd Official Documentation
- Tripwire: How Dockershim's Forthcoming Deprecation Affects Your Kubernetes (published 2021/07/01)